

Contents

[IoT Hub Documentation](#)

[Overview](#)

[What is Azure IoT Hub?](#)

[Quickstarts](#)

[Send telemetry](#)

[C](#)

[Node.js](#)

[.NET](#)

[Java](#)

[Python](#)

[Android](#)

[iOS](#)

[Control a device](#)

[Node.js](#)

[.NET](#)

[Java](#)

[Python](#)

[Android](#)

[Communicate using device streams \(preview\)](#)

[C#](#)

[Node.js](#)

[C](#)

[SSH/RDP using device streams \(preview\)](#)

[C#](#)

[Node.js](#)

[C](#)

[Tutorials](#)

[Routing messages](#)

[Use metrics and diagnostic logs](#)

- Perform manual failover
- Configure your devices
- Manage firmware updates
- Test device connectivity

Concepts

- Overview of device management
- Compare IoT Hub and Event Hubs
- Choose the right tier
- High availability and disaster recovery
- Supporting additional protocols
- Compare message and event routing
- Device configuration best practices
- Azure IoT SDKs platform support
- Overview of IoT Hub device streams (preview)

Developer guide

- Device-to-cloud feature guide
 - Create and read IoT Hub messages
 - Read device-to-cloud messages from the built-in endpoint
 - Use custom endpoints and routing rules for device-to-cloud messages
 - Add queries to message routes
- React to IoT Hub events
 - Send cloud-to-device messages from IoT Hub
 - Choose a communication protocol
- Upload files from a device
- Manage device identities
- Control access to IoT Hub
- Understand device twins
- Understand module twins
- Invoke direct methods on a device

[Schedule jobs on multiple devices](#)

[IoT Hub endpoints](#)

[Query language](#)

[Quotas and throttling](#)

[Pricing examples](#)

[Device and service SDKs](#)

[MQTT support](#)

[Glossary](#)

[Security](#)

[Security from the ground up](#)

[Security best practices](#)

[Security architecture](#)

[Secure your IoT deployment](#)

[Secure using X.509 CA certificates](#)

[X.509 CA certificate security overview](#)

[X.509 CA certificate security concepts](#)

[How-to guides](#)

[Develop](#)

[Use device and service SDKs](#)

[Azure IoT SDKs platform support](#)

[Use the IoT device SDK for C](#)

[Use the IoTHubClient](#)

[Use the serializer](#)

[Develop for constrained devices](#)

[Develop for mobile devices](#)

[Manage connectivity and reliable messaging](#)

[Develop for Android Things platform](#)

[Query Avro data from a hub route](#)

[Order device connection state events from Event Grid](#)

[Send cloud-to-device messages](#)

[.NET](#)

[Java](#)

- [Node.js](#)
- [Python](#)
- [iOS](#)
- [Upload files from devices](#)
 - [.NET](#)
 - [Java](#)
 - [Node.js](#)
 - [Python](#)
- [Get started with device twins](#)
 - [Node.js](#)
 - [.NET](#)
 - [Java](#)
 - [Python](#)
- [Get started with module twins](#)
 - [Portal](#)
 - [.NET](#)
 - [Python](#)
 - [C](#)
 - [Node](#)
- [Get started with device management](#)
 - [Node.js](#)
 - [.NET](#)
 - [Java](#)
 - [Python](#)
- [Schedule and broadcast jobs](#)
 - [Node.js](#)
 - [.NET](#)
 - [Java](#)
 - [Python](#)
- [Manage](#)
 - [Create an IoT hub](#)
 - [Use Azure portal](#)

[Use Azure IoT Tools for VS Code](#)

[Use Azure PowerShell](#)

[Use Azure CLI](#)

[Use the REST API](#)

[Use a template from Azure PowerShell](#)

[Use a template from .NET](#)

[Configure file upload](#)

[Use Azure portal](#)

[Use Azure PowerShell](#)

[Use Azure CLI](#)

[Metrics in Azure Monitor](#)

[Set up diagnostic logs](#)

[Secure your hub with an X.509 certificate](#)

[Upgrade an IoT hub](#)

[Enable message tracing](#)

[Configure IP filtering](#)

[Configure devices at scale](#)

[Use Azure portal](#)

[Use Azure CLI](#)

[Bulk manage IoT devices](#)

[Use real devices](#)

[Use an online simulator](#)

[Use a physical device](#)

[Raspberry Pi with Node.js](#)

[Raspberry Pi with C](#)

[MXChip IoT DevKit with Arduino](#)

[Adafruit Feather HUZZAH ESP8266 with Arduino](#)

[Use MXChip IoT DevKit](#)

[Translate voice message with Azure Cognitive Services](#)

[Retrieve a Twitter message with Azure Functions](#)

[Send messages to an MQTT server using Eclipse Paho APIs](#)

[Monitor the magnetic sensor and send email notifications with Azure Functions](#)

Extended IoT scenarios

[Manage cloud device messaging with Azure IoT Tools for VS Code](#)

[Manage cloud device messaging with Cloud Explorer for Visual Studio](#)

[Data Visualization in Power BI](#)

[Data Visualization with Web Apps](#)

[Weather forecast using Azure Machine Learning](#)

[Device management with Azure IoT Tools for VS Code](#)

[Device management with Cloud Explorer for Visual Studio](#)

[Device management with IoT extension for Azure CLI](#)

[Remote monitoring and notifications with Logic Apps](#)

Troubleshoot

Device disconnections

Reference

[Azure CLI](#)

[.NET \(Service\)](#)

[.NET \(Devices\)](#)

[Java \(Service\)](#)

[Java \(Devices\)](#)

[Node.js \(Devices\)](#)

[Node.js \(Service\)](#)

[C device SDK](#)

[Azure IoT Edge](#)

[REST \(Device\)](#)

[REST \(Service\)](#)

[REST \(IoT Hub Resource\)](#)

[REST \(Certificates\)](#)

[Resource Manager template](#)

[Feature and API retirement](#)

[Operations monitoring](#)

[Migrate to diagnostics settings](#)

Related

[Solutions](#)

- [IoT solution accelerators](#)
- [IoT Central](#)
- [Platform Services](#)
 - [IoT Hub](#)
 - [IoT Hub Device Provisioning Service](#)
 - [IoT Service SDKs](#)
 - [Maps](#)
 - [Time Series Insights](#)
- [Edge](#)
 - [IoT Edge](#)
 - [IoT Device SDKs](#)
- [Resources](#)
 - [Azure IoT Samples for C# \(.NET\)](#)
 - [Azure IoT Samples for Node.js](#)
 - [Azure IoT Samples for Java](#)
 - [Azure IoT Samples for Python](#)
 - [Azure IoT Samples for iOS Platform](#)
 - [Azure Certified for IoT device catalog](#)
 - [Azure IoT Developer Center](#)
 - [Customer data requests](#)
 - [Azure Roadmap](#)
 - [Azure IoT Tools](#)
 - [DeviceExplorer tool](#)
 - [iothub-diagnostics tool](#)
 - [MSDN forum](#)
 - [Pricing](#)
 - [Pricing calculator](#)
 - [Service updates](#)
 - [Stack Overflow](#)
 - [Technical case studies](#)
 - [Videos](#)

What is Azure IoT Hub?

3/10/2019 • 3 minutes to read

IoT Hub is a managed service, hosted in the cloud, that acts as a central message hub for bi-directional communication between your IoT application and the devices it manages. You can use Azure IoT Hub to build IoT solutions with reliable and secure communications between millions of IoT devices and a cloud-hosted solution backend. You can connect virtually any device to IoT Hub.

IoT Hub supports communications both from the device to the cloud and from the cloud to the device. IoT Hub supports multiple messaging patterns such as device-to-cloud telemetry, file upload from devices, and request-reply methods to control your devices from the cloud. IoT Hub monitoring helps you maintain the health of your solution by tracking events such as device creation, device failures, and device connections.

IoT Hub's capabilities help you build scalable, full-featured IoT solutions such as managing industrial equipment used in manufacturing, tracking valuable assets in healthcare, and monitoring office building usage.

Scale your solution

IoT Hub scales to millions of simultaneously connected devices and millions of events per second to support your IoT workloads. IoT Hub offers several tiers of service to best fit your scalability needs. Learn more by checking out the [pricing page](#).

Secure your communications

IoT Hub gives you a secure communication channel for your devices to send data.

- Per-device authentication enables each device to connect securely to IoT Hub and for each device to be managed securely.
- You have complete control over device access and can control connections at the per-device level.
- The [IoT Hub Device Provisioning Service](#) automatically provisions devices to the right IoT hub when the device first boots up.
- Multiple authentication types support a variety of device capabilities:
 - SAS token-based authentication to quickly get started with your IoT solution.
 - Individual X.509 certificate authentication for secure, standards-based authentication.
 - X.509 CA authentication for simple, standards-based enrollment.

Route device data

Built-in message routing functionality gives you flexibility to set up automatic rules-based message fan-out:

- Use message routing to control where your hub sends device telemetry.
- There is no additional cost to route messages to multiple endpoints.
- No-code routing rules take the place of custom message dispatcher code.

Integrate with other services

You can integrate IoT Hub with other Azure services to build complete, end-to-end solutions. For example, use:

- [Azure Event Grid](#) to enable your business to react quickly to critical events in a reliable, scalable, and secure manner.
- [Azure Logic Apps](#) to automate business processes.
- [Azure Machine Learning](#) to add machine learning and AI models to your solution.
- [Azure Stream Analytics](#) to run real-time analytic computations on the data streaming from your devices.

Configure and control your devices

You can manage your devices connected to IoT Hub with an array of built-in functionality.

- Store, synchronize, and query device metadata and state information for all your devices.
- Set device state either per-device or based on common characteristics of devices.
- Automatically respond to a device-reported state change with message routing integration.

Make your solution highly available

There's a 99.9% [Service Level Agreement for IoT Hub](#). The full [Azure SLA](#) explains the guaranteed availability of Azure as a whole.

Connect your devices

Use the [Azure IoT device SDK](#) libraries to build applications that run on your devices and interact with IoT Hub. Supported platforms include multiple Linux distributions, Windows, and real-time operating systems. Supported languages include:

- C
- C#
- Java
- Python
- Node.js.

IoT Hub and the device SDKs support the following protocols for connecting devices:

- HTTPS
- AMQP
- AMQP over WebSockets
- MQTT
- MQTT over WebSockets

If your solution cannot use the device libraries, devices can use the MQTT v3.1.1, HTTPS 1.1, or AMQP 1.0 protocols to connect natively to your hub.

If your solution cannot use one of the supported protocols, you can extend IoT Hub to support custom protocols:

- Use [Azure IoT Edge](#) to create a field gateway to perform protocol translation on the edge.
- Customize the [Azure IoT protocol gateway](#) to perform protocol translation in the cloud.

Quotas and limits

Each Azure subscription has default quota limits in place to prevent service abuse, and these limits could impact

the scope of your IoT solution. The current limit on a per-subscription basis is 50 IoT hubs per subscription. You can request quota increases by contacting support. For more details on quota limits:

- [Azure subscription service limits](#)
- [IoT Hub throttling and you](#)

Next steps

To try out an end-to-end IoT solution, check out the IoT Hub quickstarts:

- [Quickstart: Send telemetry from a device to an IoT hub](#)

Quickstart: Send telemetry from a device to an IoT hub and read it with a back-end application (C)

2/26/2019 • 9 minutes to read

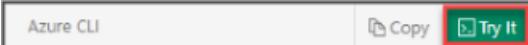
IoT Hub is an Azure service that enables you to ingest high volumes of telemetry from your IoT devices into the cloud for storage or processing. In this quickstart, you send telemetry from a simulated device application, through IoT Hub, to a back-end application for processing.

The quickstart uses a C sample application from the [Azure IoT device SDK for C](#) to send telemetry to an IoT hub. The Azure IoT device SDKs are written in [ANSI C \(C99\)](#) for portability and broad platform compatibility. Before running the sample code, you will create an IoT hub and register the simulated device with that hub.

This article written for Windows but you can complete this quickstart on Linux as well.

Open Azure Cloud Shell

Azure Cloud Shell is a free, interactive shell that you can use to run the steps in this article. Common Azure tools are preinstalled and configured in Cloud Shell for you to use with your account. Just select the **Copy** button to copy the code, paste it in Cloud Shell, and then press Enter to run it. There are a few ways to open Cloud Shell:

Select Try It in the upper-right corner of a code block.	
Open Cloud Shell in your browser.	
Select the Cloud Shell button on the menu in the upper-right corner of the Azure portal .	

If you don't have an Azure subscription, create a [free account](#) before you begin.

Prerequisites

- Install [Visual Studio 2017](#) with the 'Desktop development with C++' workload enabled.
- Install the latest version of [Git](#).

Prepare the development environment

For this quickstart, you will be using the [Azure IoT device SDK for C](#).

You can use the SDK by installing the packages and libraries for the following environments:

- **Linux:** apt-get packages are available for Ubuntu 16.04 and 18.04 using the following CPU architectures: amd64, arm64, armhf and i386. For more information, see [Using apt-get to create a C device client project on Ubuntu](#).
- **mbed:** For developers creating device applications on the mbed platform, we have published a library and samples that will get you started in minutes with Azure IoT Hub. For more information, see [Use the mbed library](#).

- **Arduino:** If you are developing on Arduino, you can leverage the Azure IoT library available in the Arduino IDE library manager. For more information, see [The Azure IoT Hub library for Arduino](#).
- **iOS:** The IoT Hub Device SDK is available as CocoaPods for Mac and iOS device development. For more information, see [iOS Samples for Microsoft Azure IoT](#).

However, in this quickstart, you will prepare a development environment used to clone and build the [Azure IoT C SDK](#) from GitHub. The SDK on GitHub includes the sample code used in this quickstart.

1. Download the version 3.13.4 of the [CMake build system](#). Verify the downloaded binary using the corresponding cryptographic hash value. The following example used Windows PowerShell to verify the cryptographic hash for version 3.11.4 of the x64 MSI distribution:

```
PS C:\Downloads> $hash = get-filehash ./cmake-3.13.4-win64-x64.msi
PS C:\Downloads> $hash.Hash -eq "64AC7DD5411B48C2717E15738B83EA0D4347CD51B940487DFF7F99A870656C09"
True
```

The following hash values for version 3.13.4 were listed on the CMake site at the time of this writing:

```
563a39e0a7c7368f81bfa1c3aff8b590a0617cdfe51177ddc808f66cc0866c76 cmake-3.13.4-Linux-x86_64.tar.gz
7c37235ece6ce85aab2ce169106e0e729504ad64707d56e4dbfc982cb4263847 cmake-3.13.4-win32-x86.msi
64ac7dd5411b48c2717e15738b83ea0d4347cd51b940487dff7f99a870656c09 cmake-3.13.4-win64-x64.msi
```

It is important that the Visual Studio prerequisites (Visual Studio and the 'Desktop development with C++' workload) are installed on your machine, **before** starting the `cmake` installation. Once the prerequisites are in place, and the download is verified, install the CMake build system.

2. Open a command prompt or Git Bash shell. Execute the following command to clone the [Azure IoT C SDK](#) GitHub repository:

```
git clone https://github.com/Azure/azure-iot-sdk-c.git --recursive
```

The size of this repository is currently around 220 MB. You should expect this operation to take several minutes to complete.

3. Create a `cmake` subdirectory in the root directory of the git repository, and navigate to that folder.

```
cd azure-iot-sdk-c
mkdir cmake
cd cmake
```

4. Run the following command that builds a version of the SDK specific to your development client platform. A Visual Studio solution for the simulated device will be generated in the `cmake` directory.

```
cmake ..
```

If `cmake` does not find your C++ compiler, you might get build errors while running the above command. If that happens, try running this command in the [Visual Studio command prompt](#).

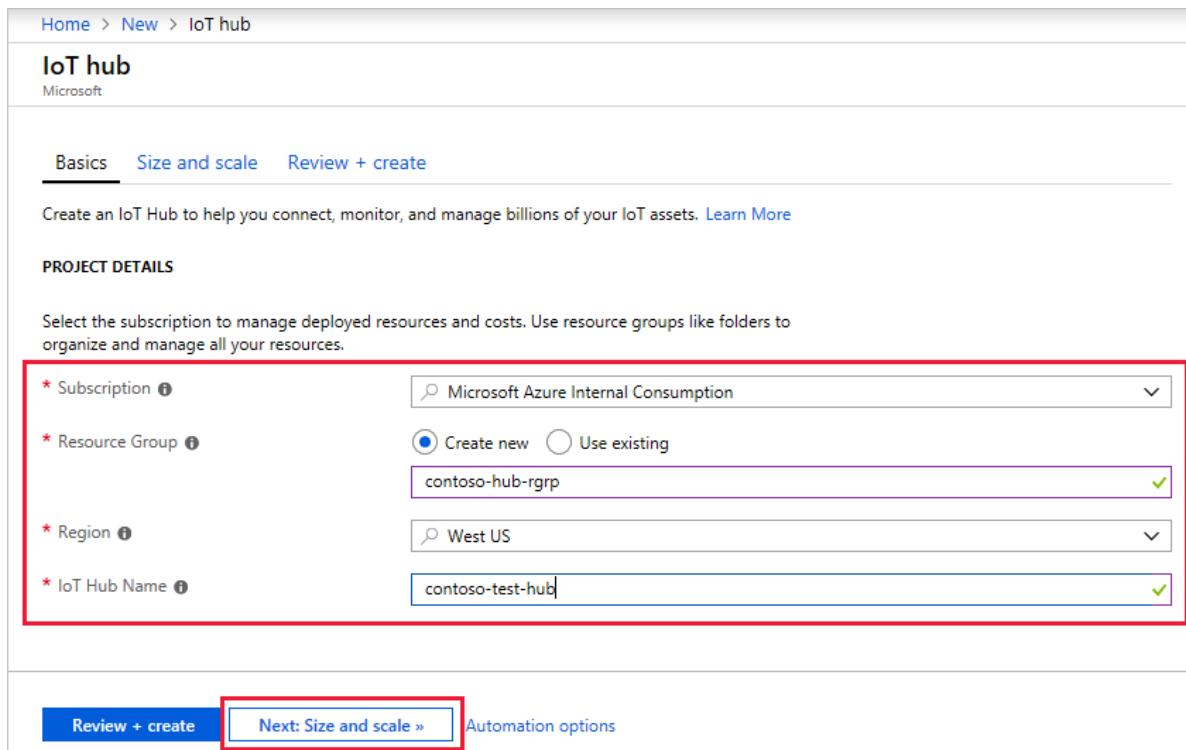
Once the build succeeds, the last few output lines will look similar to the following output:

```
$ cmake ..  
-- Building for: Visual Studio 15 2017  
-- Selecting Windows SDK version 10.0.16299.0 to target Windows 10.0.17134.  
-- The C compiler identification is MSVC 19.12.25835.0  
-- The CXX compiler identification is MSVC 19.12.25835.0  
  
...  
  
-- Configuring done  
-- Generating done  
-- Build files have been written to: E:/IoT Testing/azure-iot-sdk-c/cmake
```

Create an IoT hub

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose **+Create a resource**, then choose **Internet of Things**.
3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.



The screenshot shows the 'Basics' step of the Azure portal for creating an IoT hub. It includes fields for Subscription (set to Microsoft Azure Internal Consumption), Resource Group (set to contoso-hub-rgrp), Region (set to West US), and IoT Hub Name (set to contoso-test-hub). The 'Next: Size and scale >' button is highlighted with a red box.

Fill in the fields.

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

The screenshot shows the 'Size and scale' configuration page for an IoT hub. At the top, there are three tabs: 'Basics' (selected), 'Size and scale' (underlined), and 'Review + create'. A note below the tabs states: 'Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)'.

SCALE TIER AND UNITS

* Pricing and scale tier: S1: Standard tier. A link to 'Learn how to choose the right IoT Hub tier for your solution' is provided.

Number of S1 IoT Hub units: A slider set to 1. A note below it says: 'This determines your IoT Hub scale capability and can be changed as your need increases.'

A list of features with their status: Device-to-cloud-messages (Enabled), Message routing (Enabled), Cloud-to-device commands (Enabled), IoT Edge (Enabled), and Device management (Enabled).

Advanced Settings

Device-to-cloud partitions: A slider set to 4.

At the bottom, there are three buttons: 'Review + create' (highlighted in blue), '[« Previous: Basics](#)', and '[Automation options](#)'.

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: The number of messages allowed per unit per day depends on your hub's pricing tier. For example, if you want the IoT hub to support ingress of 700,000 messages, you choose two S1 tier units.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

Advanced / Device-to-cloud partitions: This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most IoT hubs only need four partitions.

5. Click **Review + create** to review your choices. You see something similar to this screen.

The screenshot shows the 'Review + create' step of the Azure portal for creating an IoT hub. The 'Review + create' button is highlighted with a red box. At the bottom left, the 'Create' button is also highlighted with a red box.

6. Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Register a device

A device must be registered with your IoT hub before it can connect. In this section, you will use the Azure Cloud Shell with the [IoT extension](#) to register a simulated device.

1. Run the following commands in Azure Cloud Shell to add the IoT Hub CLI extension and to create the device identity.

YourIoTHubName : Replace this placeholder below with the name you choose for your IoT hub.

MyCDevice : This is the name given for the registered device. Use MyCDevice as shown. If you choose a different name for your device, you will also need to use that name throughout this article, and update the device name in the sample applications before you run them.

```
az extension add --name azure-cli-iot-ext
az iot hub device-identity create --hub-name YourIoTHubName --device-id MyCDevice
```

2. Run the following commands in Azure Cloud Shell to get the *device connection string* for the device you just registered:

YourIoTHubName : Replace this placeholder below with the name you choose for your IoT hub.

```
az iot hub device-identity show-connection-string --hub-name YourIoTHubName --device-id MyCDevice --
output table
```

Make a note of the device connection string, which looks like:

```
HostName={YourIoTHubName}.azure-devices.net;DeviceId=MyNodeDevice;SharedAccessKey={YourSharedAccessKey}
```

You use this value later in the quickstart.

Send simulated telemetry

The simulated device application connects to a device-specific endpoint on your IoT hub and sends a string as simulated telemetry.

1. Using a text editor, open the `iothub_convenience_sample.c` source file and review the sample code for sending telemetry. The file is located in the following location:

```
\azure-iot-sdk-c\iothub_client\samples\iothub_convenience_sample\iothub_convenience_sample.c
```

2. Find the declaration of the `connectionString` constant:

```
/* Paste in your device connection string */
static const char* connectionString = "[device connection string]";
```

Replace the value of the `connectionString` constant with the device connection string you made a note of previously. Then save your changes to **iothub_convenience_sample.c**.

3. In a local terminal window, navigate to the `iothub_convenience_sample` project directory in the CMake directory that you created in the Azure IoT C SDK.

```
cd /azure-iot-sdk-c/cmake/iothub_client/samples/iothub_convenience_sample
```

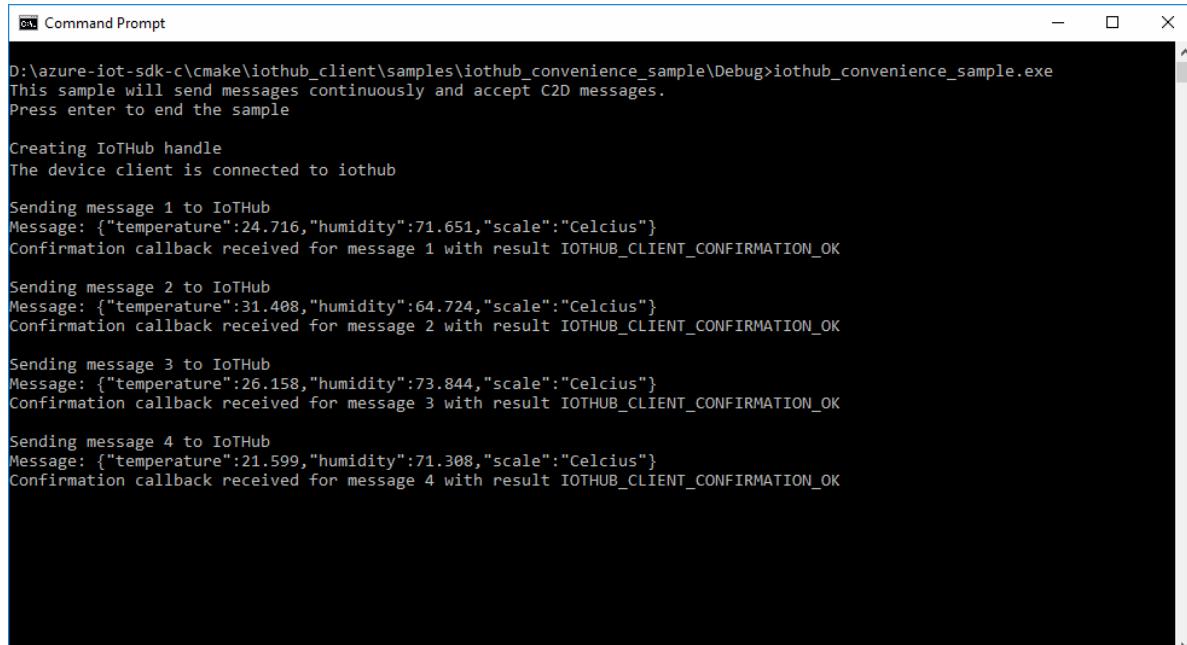
4. Run CMake in your local terminal window to build the sample with your updated `connectionString` value:

```
cmake --build . --target iothub_convenience_sample --config Debug
```

5. In a local terminal window, run the following command to run the simulated device application:

```
Debug\iothub_convenience_sample.exe
```

The following screenshot shows the output as the simulated device application sends telemetry to the IoT hub:



A screenshot of a Windows Command Prompt window titled "Command Prompt". The window shows the output of a CMake build and the execution of the "iothub_convenience_sample" application. The application logs messages indicating it's creating an IoT Hub handle, connecting to the device client, and sending four messages to the IoT Hub. Each message contains temperature, humidity, and scale information in Celsius. Confirmation callbacks are also shown for each message.

```
D:\azure-iot-sdk-c\cmake\iothub_client\samples\iothub_convenience_sample\Debug>iothub_convenience_sample.exe
This sample will send messages continuously and accept C2D messages.
Press enter to end the sample

Creating IoT Hub handle
The device client is connected to iothub

Sending message 1 to IoT Hub
Message: {"temperature":24.716,"humidity":71.651,"scale":"Celcius"}
Confirmation callback received for message 1 with result IOTHUB_CLIENT_CONFIRMATION_OK

Sending message 2 to IoT Hub
Message: {"temperature":31.408,"humidity":64.724,"scale":"Celcius"}
Confirmation callback received for message 2 with result IOTHUB_CLIENT_CONFIRMATION_OK

Sending message 3 to IoT Hub
Message: {"temperature":26.158,"humidity":73.844,"scale":"Celcius"}
Confirmation callback received for message 3 with result IOTHUB_CLIENT_CONFIRMATION_OK

Sending message 4 to IoT Hub
Message: {"temperature":21.599,"humidity":71.308,"scale":"Celcius"}
Confirmation callback received for message 4 with result IOTHUB_CLIENT_CONFIRMATION_OK
```

Read the telemetry from your hub

In this section, you will use the Azure Cloud Shell with the [IoT extension](#) to monitor the device messages that are sent by the simulated device.

1. Using the Azure Cloud Shell, run the following command to connect and read messages from your IoT hub:

YourIoTHubName : Replace this placeholder below with the name you choose for your IoT hub.

```
az iot hub monitor-events --hub-name YourIoTHubName --output table
```

```
@Azure:~$ az iot hub monitor-events --hub-name SendTeleHub --output table
Starting event monitor, use ctrl-c to stop...
event:
  origin: MyCDevice
  payload:
    humidity: 65.856
    scale: Celcius
    temperature: 26.142

event:
  origin: MyCDevice
  payload:
    humidity: 73.261
    scale: Celcius
    temperature: 23.623

event:
  origin: MyCDevice
  payload:
    humidity: 64.842
    scale: Celcius
    temperature: 33.556

event:
  origin: MyCDevice
  payload:
    humidity: 69.906
    scale: Celcius
    temperature: 32.279
```

Clean up resources

If you will be continuing to the next recommended article, you can keep the resources you've already created and reuse them.

Otherwise, you can delete the Azure resources created in this article to avoid charges.

IMPORTANT

Deleting a resource group is irreversible. The resource group and all the resources contained in it are permanently deleted. Make sure that you do not accidentally delete the wrong resource group or resources. If you created the IoT Hub inside an existing resource group that contains resources you want to keep, only delete the IoT Hub resource itself instead of deleting the resource group.

To delete a resource group by name:

1. Sign in to the [Azure portal](#) and click **Resource groups**.
2. In the **Filter by name...** textbox, type the name of the resource group containing your IoT Hub.
3. To the right of your resource group in the result list, click ... then **Delete resource group**.

The screenshot shows the Microsoft Azure portal interface. On the left sidebar, under 'FAVORITES', the 'Resource groups' option is selected and highlighted with a red box. The main content area is titled 'Resource groups' and shows a table with one item. The table has columns: NAME, SUBSCRIPTION, and LOCATION. A single row is selected, showing 'TestResources' in the NAME column, 'Prototype3' in the SUBSCRIPTION column, and 'All locations' in the LOCATION column. To the right of the row, there is a context menu with options: 'Delete resource group' (highlighted with a red box) and '...'. At the top of the page, the 'Subscriptions' section shows 'TestResources' is selected.

4. You will be asked to confirm the deletion of the resource group. Type the name of your resource group again to confirm, and then click **Delete**. After a few moments, the resource group and all of its contained resources are deleted.

Next steps

In this quickstart, you've setup an IoT hub, registered a device, sent simulated telemetry to the hub using a C application, and read the telemetry from the hub using the Azure Cloud Shell.

To learn more about developing with the Azure IoT Hub C SDK, continue to the following How-to guide:

[Develop using Azure IoT Hub C SDK](#)

Quickstart: Send telemetry from a device to an IoT hub and read it with a back-end application (Node.js)

2/28/2019 • 7 minutes to read

IoT Hub is an Azure service that enables you to ingest high volumes of telemetry from your IoT devices into the cloud for storage or processing. In this quickstart, you send telemetry from a simulated device application, through IoT Hub, to a back-end application for processing.

The quickstart uses two pre-written Node.js applications, one to send the telemetry and one to read the telemetry from the hub. Before you run these two applications, you create an IoT hub and register a device with the hub.

Open Azure Cloud Shell

Azure Cloud Shell is a free, interactive shell that you can use to run the steps in this article. Common Azure tools are preinstalled and configured in Cloud Shell for you to use with your account. Just select the **Copy** button to copy the code, paste it in Cloud Shell, and then press Enter to run it. There are a few ways to open Cloud Shell:

Select Try It in the upper-right corner of a code block.	
Open Cloud Shell in your browser.	
Select the Cloud Shell button on the menu in the upper-right corner of the Azure portal .	

If you don't have an Azure subscription, create a [free account](#) before you begin.

Prerequisites

The two sample applications you run in this quickstart are written in Node.js. You need Node.js v4.x.x or later on your development machine.

You can download Node.js for multiple platforms from nodejs.org.

You can verify the current version of Node.js on your development machine using the following command:

```
node --version
```

Download the sample Node.js project from <https://github.com/Azure-Samples/azure-iot-samples-node/archive/master.zip> and extract the ZIP archive.

Create an IoT hub

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose **+Create a resource**, then choose **Internet of Things**.
3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.

The screenshot shows the 'Basics' step of the IoT Hub creation wizard. It includes fields for Subscription, Resource Group, Region, and IoT Hub Name, each with validation checkmarks. A red box highlights the project details section. At the bottom, there are 'Review + create' and 'Next: Size and scale >' buttons, with 'Next: Size and scale' also highlighted by a red box.

Fill in the fields.

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Review + create

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS

* Pricing and scale tier [?](#) S1: Standard tier [?](#) [Learn how to choose the right IoT Hub tier for your solution](#)

Number of S1 IoT Hub units [?](#) 1 [1](#)

This determines your IoT Hub scale capability and can be changed as your need increases.

Device-to-cloud-messages ? Enabled
Message routing ? Enabled
Cloud-to-device commands ? Enabled
IoT Edge ? Enabled
Device management ? Enabled

[Advanced Settings](#)

Device-to-cloud partitions [?](#) 4 [4](#)

[Review + create](#) [« Previous: Basics](#) [Automation options](#)

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: The number of messages allowed per unit per day depends on your hub's pricing tier. For example, if you want the IoT hub to support ingress of 700,000 messages, you choose two S1 tier units.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

Advanced / Device-to-cloud partitions: This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most IoT hubs only need four partitions.

5. Click **Review + create** to review your choices. You see something similar to this screen.

The screenshot shows the 'Review + create' step of the Azure IoT hub creation wizard. It displays basic information like subscription, resource group, and region, as well as size and scale details including pricing tier, number of units, messages per day, and monthly cost. At the bottom, there are 'Create' and 'Automation options' buttons, with 'Create' being highlighted with a red box.

6. Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Register a device

A device must be registered with your IoT hub before it can connect. In this quickstart, you use the Azure Cloud Shell to register a simulated device.

1. Run the following commands in Azure Cloud Shell to add the IoT Hub CLI extension and to create the device identity.

YourIoTHubName: Replace this placeholder below with the name you choose for your IoT hub.

MyNodeDevice: The name of the device you're registering. Use **MyNodeDevice** as shown. If you choose a different name for your device, you need to use that name throughout this article, and update the device name in the sample applications before you run them.

```
az extension add --name azure-cli-iot-ext
az iot hub device-identity create --hub-name YourIoTHubName --device-id MyNodeDevice
```

2. Run the following commands in Azure Cloud Shell to get the *device connection string* for the device you just registered:

YourIoTHubName: Replace this placeholder below with the name you choose for your IoT hub.

```
az iot hub device-identity show-connection-string --hub-name YourIoTHubName --device-id
MyNodeDevice --output table
```

Make a note of the device connection string, which looks like:

```
HostName={YourIoTHubName}.azure-devices.net;DeviceId=MyNodeDevice;SharedAccessKey=
{YourSharedAccessKey}
```

You use this value later in the quickstart.

3. You also need a *service connection string* to enable the back-end application to connect to your IoT hub and retrieve the messages. The following command retrieves the service connection string for your IoT hub:

YourIoTHubName: Replace this placeholder below with the name you choose for your IoT hub.

```
az iot hub show-connection-string --hub-name YourIoTHubName --output table
```

Make a note of the service connection string, which looks like:

```
HostName={YourIoTHubName}.azure-devices.net;SharedAccessKeyName=iothubowner;SharedAccessKey={YourSharedAccessKey}
```

You use this value later in the quickstart. The service connection string is different from the device connection string.

Send simulated telemetry

The simulated device application connects to a device-specific endpoint on your IoT hub and sends simulated temperature and humidity telemetry.

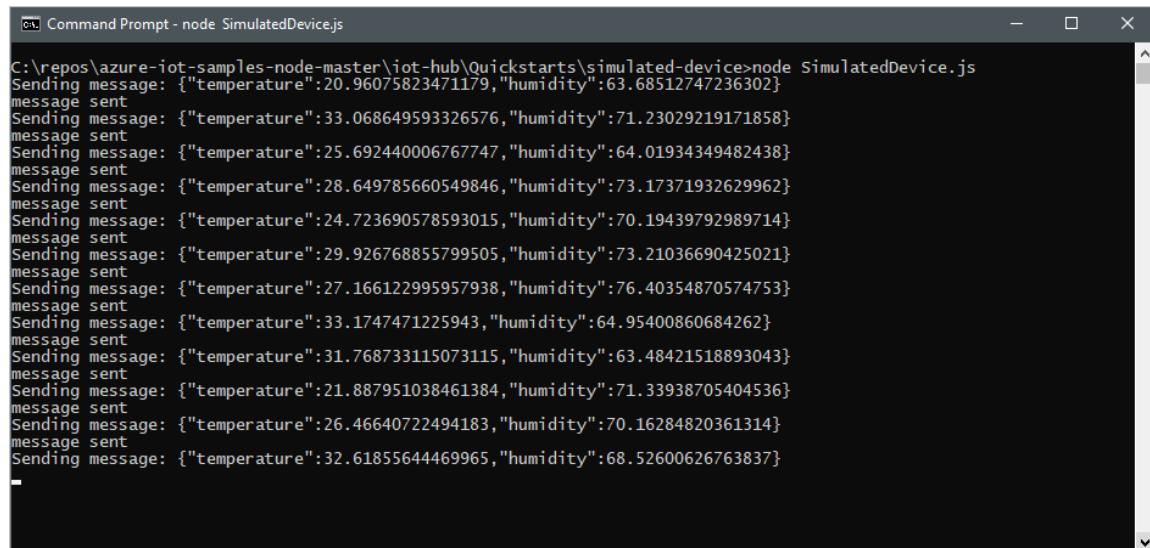
1. Open your local terminal window, navigate to the root folder of the sample Node.js project. Then navigate to the **iot-hub\Quickstarts\simulated-device** folder.
2. Open the **SimulatedDevice.js** file in a text editor of your choice.

Replace the value of the `connectionString` variable with the device connection string you made a note of previously. Then save your changes to **SimulatedDevice.js** file.

3. In the local terminal window, run the following commands to install the required libraries and run the simulated device application:

```
npm install
node SimulatedDevice.js
```

The following screenshot shows the output as the simulated device application sends telemetry to your IoT hub:



```
C:\repos\azure-iot-samples-node-master\iot-hub\Quickstarts\simulated-device>node SimulatedDevice.js
Sending message: {"temperature":20.96075823471179,"humidity":63.68512747236302}
message sent
Sending message: {"temperature":33.068649593326576,"humidity":71.23029219171858}
message sent
Sending message: {"temperature":25.692440006767747,"humidity":64.01934349482438}
message sent
Sending message: {"temperature":28.649785660549846,"humidity":73.17371932629962}
message sent
Sending message: {"temperature":24.723690578593015,"humidity":70.19439792989714}
message sent
Sending message: {"temperature":29.926768855799505,"humidity":73.21036690425021}
message sent
Sending message: {"temperature":27.166122995957938,"humidity":76.40354870574753}
message sent
Sending message: {"temperature":33.1747471225943,"humidity":64.95400860684262}
message sent
Sending message: {"temperature":31.768733115073115,"humidity":63.48421518893043}
message sent
Sending message: {"temperature":21.887951038461384,"humidity":71.33938705404536}
message sent
Sending message: {"temperature":26.46640722494183,"humidity":70.16284820361314}
message sent
Sending message: {"temperature":32.61855644469965,"humidity":68.52600626763837}
```

Read the telemetry from your hub

The back-end application connects to the service-side **Events** endpoint on your IoT Hub. The application

receives the device-to-cloud messages sent from your simulated device. An IoT Hub back-end application typically runs in the cloud to receive and process device-to-cloud messages.

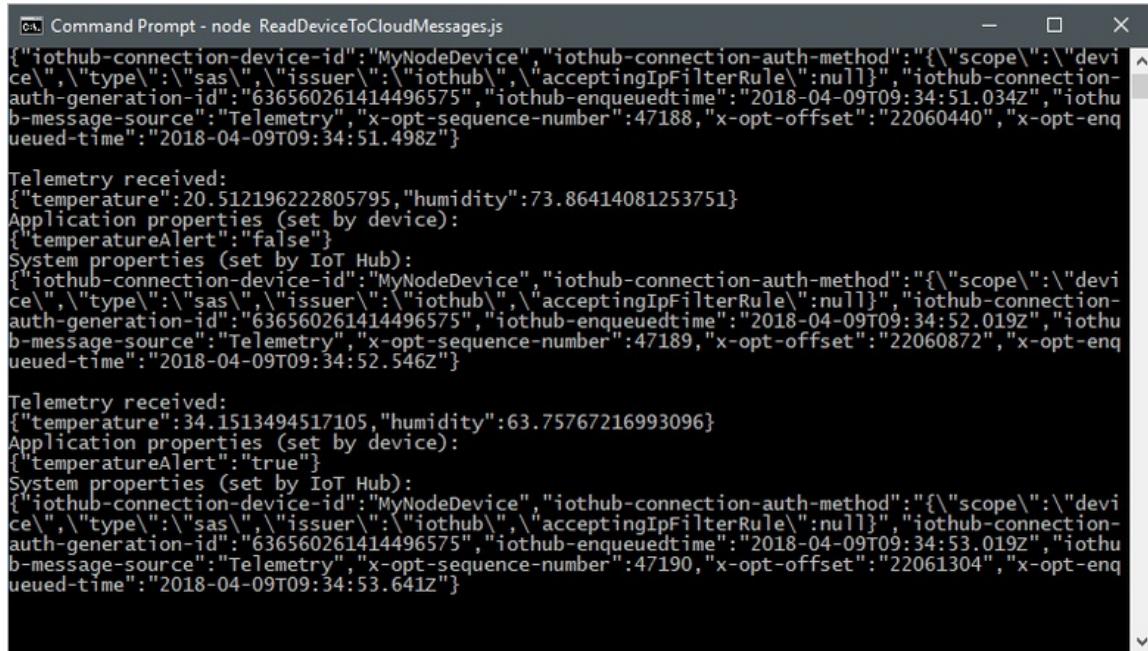
1. Open another local terminal window, navigate to the root folder of the sample Node.js project. Then navigate to the **iot-hub\Quickstarts\read-d2c-messages** folder.
2. Open the **ReadDeviceToCloudMessages.js** file in a text editor of your choice.

Replace the value of the `connectionString` variable with the service connection string you made a note of previously. Then save your changes to the **ReadDeviceToCloudMessages.js** file.

3. In the local terminal window, run the following commands to install the required libraries and run the back-end application:

```
npm install  
node ReadDeviceToCloudMessages.js
```

The following screenshot shows the output as the back-end application receives telemetry sent by the simulated device to the hub:



```
Command Prompt - node ReadDeviceToCloudMessages.js

{
  "iothub-connection-device-id": "MyNodeDevice",
  "iothub-connection-auth-method": "{\"scope\":\"device\\\", \"type\":\"sas\\\", \"issuer\": \"iothub\\\", \"acceptingIpFilterRule\":null}",
  "iothub-connection-auth-generation-id": "636560261414496575",
  "iothub-enqueueudtime": "2018-04-09T09:34:51.034Z",
  "iothub-message-source": "Telemetry",
  "x-opt-sequence-number": 47188,
  "x-opt-offset": "22060440",
  "x-opt-enqueued-time": "2018-04-09T09:34:51.498Z"
}

Telemetry received:
{
  "temperature": 20.512196222805795,
  "humidity": 73.86414081253751
}
Application properties (set by device):
{
  "temperatureAlert": "false"
}
System properties (set by IoT Hub):
{
  "iothub-connection-device-id": "MyNodeDevice",
  "iothub-connection-auth-method": "{\"scope\":\"device\\\", \"type\":\"sas\\\", \"issuer\": \"iothub\\\", \"acceptingIpFilterRule\":null}",
  "iothub-connection-auth-generation-id": "636560261414496575",
  "iothub-enqueueudtime": "2018-04-09T09:34:52.019Z",
  "iothub-message-source": "Telemetry",
  "x-opt-sequence-number": 47189,
  "x-opt-offset": "22060872",
  "x-opt-enqueued-time": "2018-04-09T09:34:52.546Z"
}

Telemetry received:
{
  "temperature": 34.1513494517105,
  "humidity": 63.75767216993096
}
Application properties (set by device):
{
  "temperatureAlert": "true"
}
System properties (set by IoT Hub):
{
  "iothub-connection-device-id": "MyNodeDevice",
  "iothub-connection-auth-method": "{\"scope\":\"device\\\", \"type\":\"sas\\\", \"issuer\": \"iothub\\\", \"acceptingIpFilterRule\":null}",
  "iothub-connection-auth-generation-id": "636560261414496575",
  "iothub-enqueueudtime": "2018-04-09T09:34:53.019Z",
  "iothub-message-source": "Telemetry",
  "x-opt-sequence-number": 47190,
  "x-opt-offset": "22061304",
  "x-opt-enqueued-time": "2018-04-09T09:34:53.641Z"
}
```

Clean up resources

If you will be continuing to the next recommended article, you can keep the resources you've already created and reuse them.

Otherwise, you can delete the Azure resources created in this article to avoid charges.

IMPORTANT

Deleting a resource group is irreversible. The resource group and all the resources contained in it are permanently deleted. Make sure that you do not accidentally delete the wrong resource group or resources. If you created the IoT Hub inside an existing resource group that contains resources you want to keep, only delete the IoT Hub resource itself instead of deleting the resource group.

To delete a resource group by name:

1. Sign in to the [Azure portal](#) and click **Resource groups**.

2. In the **Filter by name...** textbox, type the name of the resource group containing your IoT Hub.
3. To the right of your resource group in the result list, click ... then **Delete resource group**.

The screenshot shows the Microsoft Azure portal interface. On the left, the navigation bar includes 'Create a resource', 'All services', 'FAVORITES' (with 'Dashboard' and 'Resource groups' selected), 'All resources', and 'Recent'. The main content area is titled 'Resource groups' under 'Microsoft'. At the top, there are buttons for '+ Add', 'Edit columns', 'Refresh', and 'Assign Tags'. A search bar contains the text 'TestResources'. Below the search bar, it says 'Subscriptions: 1 of 7 selected - Don't see a subscription? Switch directories'. A table lists one item: '1 of 1 items selected'. The table has columns for 'NAME' (with a checkmark), 'SUBSCRIPTION' (Prototype3), and 'LOCATION'. The row for 'TestResources' shows 'Prototype3' in the subscription column. To the right of the table, there is a context menu with options: 'Delete resource group' (highlighted with a red box) and '...'. The entire row for 'TestResources' is also highlighted with a red dashed box.

4. You will be asked to confirm the deletion of the resource group. Type the name of your resource group again to confirm, and then click **Delete**. After a few moments, the resource group and all of its contained resources are deleted.

Next steps

In this quickstart, you've setup an IoT hub, registered a device, sent simulated telemetry to the hub using a Node.js application, and read the telemetry from the hub using a simple back-end application.

To learn how to control your simulated device from a back-end application, continue to the next quickstart.

[Quickstart: Control a device connected to an IoT hub](#)

Quickstart: Send telemetry from a device to an IoT hub and read it with a back-end application (C#)

2/28/2019 • 7 minutes to read

IoT Hub is an Azure service that enables you to ingest high volumes of telemetry from your IoT devices into the cloud for storage or processing. In this quickstart, you send telemetry from a simulated device application, through IoT Hub, to a back-end application for processing.

The quickstart uses two pre-written C# applications, one to send the telemetry and one to read the telemetry from the hub. Before you run these two applications, you create an IoT hub and register a device with the hub.

Open Azure Cloud Shell

Azure Cloud Shell is a free, interactive shell that you can use to run the steps in this article. Common Azure tools are preinstalled and configured in Cloud Shell for you to use with your account. Just select the **Copy** button to copy the code, paste it in Cloud Shell, and then press Enter to run it. There are a few ways to open Cloud Shell:

Select Try It in the upper-right corner of a code block.	
Open Cloud Shell in your browser.	
Select the Cloud Shell button on the menu in the upper-right corner of the Azure portal .	

If you don't have an Azure subscription, create a [free account](#) before you begin.

Prerequisites

The two sample applications you run in this quickstart are written using C#. You need the .NET Core SDK 2.1.0 or greater on your development machine.

You can download the .NET Core SDK for multiple platforms from [.NET](#).

You can verify the current version of C# on your development machine using the following command:

```
dotnet --version
```

Download the sample C# project from <https://github.com/Azure-Samples/azure-iot-samples-csharp/archive/master.zip> and extract the ZIP archive.

Create an IoT hub

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).

2. Choose +**Create a resource**, then choose **Internet of Things**.

3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.

The screenshot shows the 'Basics' step of the IoT hub creation wizard. At the top, there's a breadcrumb navigation: Home > New > IoT hub. Below it, the title 'IoT hub' is displayed with the Microsoft logo. A horizontal navigation bar at the top of the main form includes 'Basics' (which is underlined), 'Size and scale', and 'Review + create'. A descriptive text below the navigation bar says, 'Create an IoT Hub to help you connect, monitor, and manage billions of your IoT assets.' followed by a 'Learn More' link. The 'PROJECT DETAILS' section contains four fields, each marked with a red asterisk indicating they are required:

- * Subscription: A dropdown menu showing 'Microsoft Azure Internal Consumption' with a dropdown arrow icon.
- * Resource Group: A radio button group where 'Create new' is selected (indicated by a blue dot) and 'Use existing' is unselected (indicated by an empty circle). Below it is a dropdown menu containing 'contoso-hub-rgrp' with a green checkmark icon.
- * Region: A dropdown menu showing 'West US' with a dropdown arrow icon.
- * IoT Hub Name: A text input field containing 'contoso-test-hub' with a green checkmark icon.

At the bottom of the form, there are three buttons: 'Review + create' (blue), 'Next: Size and scale »' (highlighted with a red box), and 'Automation options'.

Fill in the fields.

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Review + create

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS

* Pricing and scale tier [?](#) S1: Standard tier [?](#) [Learn how to choose the right IoT Hub tier for your solution](#)

Number of S1 IoT Hub units [?](#) 1 [This determines your IoT Hub scale capability and can be changed as your need increases.](#)

Device-to-cloud-messages [?](#) Enabled
Message routing [?](#) Enabled
Cloud-to-device commands [?](#) Enabled
IoT Edge [?](#) Enabled
Device management [?](#) Enabled

Advanced Settings

Device-to-cloud partitions [?](#) 4

[Review + create](#) [« Previous: Basics](#) [Automation options](#)

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: The number of messages allowed per unit per day depends on your hub's pricing tier. For example, if you want the IoT hub to support ingress of 700,000 messages, you choose two S1 tier units.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

Advanced / Device-to-cloud partitions: This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most IoT hubs only need four partitions.

5. Click **Review + create** to review your choices. You see something similar to this screen.

BASICS

Subscription ⓘ	Microsoft Azure Internal Consumption
Resource Group ⓘ	contoso-hub-rgrp
Region ⓘ	West US
IoT Hub Name ⓘ	contoso-test-hub

SIZE AND SCALE

Pricing and scale tier ⓘ	\$1
Number of S1 IoT Hub units ⓘ	1
Messages per day ⓘ	400,000
Cost per month	25.00 USD

Create « Previous: Size and scale Automation options

6. Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Register a device

A device must be registered with your IoT hub before it can connect. In this quickstart, you use the Azure Cloud Shell to register a simulated device.

1. Run the following commands in Azure Cloud Shell to add the IoT Hub CLI extension and to create the device identity.

YourIoTHubName: Replace this placeholder below with the name you choose for your IoT hub.

MyDotnetDevice: The name of the device you're registering. Use **MyDotnetDevice** as shown. If you choose a different name for your device, you need to use that name throughout this article, and update the device name in the sample applications before you run them.

```
az extension add --name azure-cli-iot-ext
az iot hub device-identity create --hub-name YourIoTHubName --device-id MyDotnetDevice
```

2. Run the following commands in Azure Cloud Shell to get the *device connection string* for the device you just registered:

YourIoTHubName: Replace this placeholder below with the name you choose for your IoT hub.

```
az iot hub device-identity show-connection-string --hub-name YourIoTHubName --device-id
MyDotnetDevice --output table
```

Make a note of the device connection string, which looks like:

```
HostName={YourIoTHubName}.azure-devices.net;DeviceId=MyNodeDevice;SharedAccessKey=
{YourSharedAccessKey}
```

You use this value later in the quickstart.

3. You also need the *Event Hubs-compatible endpoint*, *Event Hubs-compatible path*, and *iothubowner primary key* from your IoT hub to enable the back-end application to connect to your IoT hub and retrieve the messages. The following commands retrieve these values for your IoT hub:

YourIoTHubName: Replace this placeholder below with the name you choose for your IoT hub.

```
az iot hub show --query properties.eventHubEndpoints.events.endpoint --name YourIoTHubName  
az iot hub show --query properties.eventHubEndpoints.events.path --name YourIoTHubName  
az iot hub policy show --name iothubowner --query primaryKey --hub-name YourIoTHubName
```

Make a note of these three values, which you use later in the quickstart.

Send simulated telemetry

The simulated device application connects to a device-specific endpoint on your IoT hub and sends simulated temperature and humidity telemetry.

1. In a local terminal window, navigate to the root folder of the sample C# project. Then navigate to the **iot-hub\Quickstarts\simulated-device** folder.
2. Open the **SimulatedDevice.cs** file in a text editor of your choice.

Replace the value of the `sConnectionString` variable with the device connection string you made a note of previously. Then save your changes to **SimulatedDevice.cs** file.

3. In the local terminal window, run the following commands to install the required packages for simulated device application:

```
dotnet restore
```

4. In the local terminal window, run the following command to build and run the simulated device application:

```
dotnet run
```

The following screenshot shows the output as the simulated device application sends telemetry to your IoT hub:

```
c:\repos\iot-hub-quickstarts-dotnet\simulated-device>dotnet run
IoT Hub Quickstarts #1 - Simulated device. Ctrl-C to exit.

04/04/2018 11:15:22 > Sending message: {"temperature":31.02260307922149,"humidity":66.922409537677851}
04/04/2018 11:15:23 > Sending message: {"temperature":25.782883018619792,"humidity":64.358296908605055}
04/04/2018 11:15:24 > Sending message: {"temperature":23.521925824937377,"humidity":63.064605203952922}
04/04/2018 11:15:25 > Sending message: {"temperature":29.148115233587156,"humidity":60.096141295552322}
04/04/2018 11:15:27 > Sending message: {"temperature":23.686750376916841,"humidity":61.101306500426169}
04/04/2018 11:15:28 > Sending message: {"temperature":31.85187338239135,"humidity":77.934687918952989}
04/04/2018 11:15:29 > Sending message: {"temperature":33.109382886024832,"humidity":62.153963494186272}
```

Read the telemetry from your hub

The back-end application connects to the service-side **Events** endpoint on your IoT Hub. The application receives the device-to-cloud messages sent from your simulated device. An IoT Hub back-end application typically runs in the cloud to receive and process device-to-cloud messages.

1. In another local terminal window, navigate to the root folder of the sample C# project. Then navigate to the **iot-hub\Quickstarts\read-d2c-messages** folder.
2. Open the **ReadDeviceToCloudMessages.cs** file in a text editor of your choice. Update the following variables and save your changes to the file.

VARIABLE	VALUE
s_eventHubsCompatibleEndpoint	Replace the value of the variable with the Event Hubs-compatible endpoint you made a note of previously.
s_eventHubsCompatiblePath	Replace the value of the variable with the Event Hubs-compatible path you made a note of previously.
s_iotHubSasKey	Replace the value of the variable with the iothubowner primary key you made a note of previously.

3. In the local terminal window, run the following commands to install the required libraries for the back-end application:

```
dotnet restore
```

4. In the local terminal window, run the following commands to build and run the back-end application:

```
dotnet run
```

The following screenshot shows the output as the back-end application receives telemetry sent by the simulated device to the hub:

```
Command Prompt - dotnet run
Application properties (set by device):
  temperatureAlert: false
System properties (set by IoT Hub):
  iothub-connection-device-id: MyDotnetDevice
  iothub-connection-auth-method: {"scope":"device","type":"sas","issuer":"iothub","acceptingIpFilterRule":null}
  iothub-connection-auth-generation-id: 636576682523837526
  iothub-enqueuedtime: 04/04/2018 10:23:53
  iothub-message-source: Telemetry
  x-opt-sequence-number: 46690
  x-opt-offset: 21828120
  x-opt-enqueued-time: 04/04/2018 10:23:53
Listening for messages on: 1
Message received on partition 1:
  {"temperature":21.947054528606614,"humidity":68.693635262871922}:
Application properties (set by device):
  temperatureAlert: false
System properties (set by IoT Hub):
  iothub-connection-device-id: MyDotnetDevice
  iothub-connection-auth-method: {"scope":"device","type":"sas","issuer":"iothub","acceptingIpFilterRule":null}
  iothub-connection-auth-generation-id: 636576682523837526
  iothub-enqueuedtime: 04/04/2018 10:23:54
  iothub-message-source: Telemetry
  x-opt-sequence-number: 46691
  x-opt-offset: 21828560
  x-opt-enqueued-time: 04/04/2018 10:23:54
Listening for messages on: 1
```

Clean up resources

If you will be continuing to the next recommended article, you can keep the resources you've already created and reuse them.

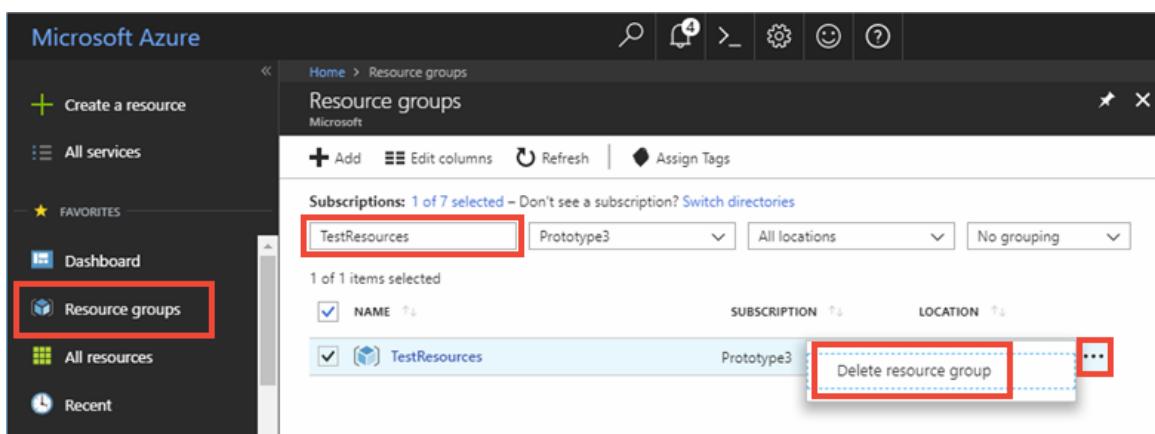
Otherwise, you can delete the Azure resources created in this article to avoid charges.

IMPORTANT

Deleting a resource group is irreversible. The resource group and all the resources contained in it are permanently deleted. Make sure that you do not accidentally delete the wrong resource group or resources. If you created the IoT Hub inside an existing resource group that contains resources you want to keep, only delete the IoT Hub resource itself instead of deleting the resource group.

To delete a resource group by name:

1. Sign in to the [Azure portal](#) and click **Resource groups**.
2. In the **Filter by name...** textbox, type the name of the resource group containing your IoT Hub.
3. To the right of your resource group in the result list, click ... then **Delete resource group**.



4. You will be asked to confirm the deletion of the resource group. Type the name of your resource group again to confirm, and then click **Delete**. After a few moments, the resource group and all of its contained resources are deleted.

Next steps

In this quickstart, you've setup an IoT hub, registered a device, sent simulated telemetry to the hub using a C# application, and read the telemetry from the hub using a simple back-end application.

To learn how to control your simulated device from a back-end application, continue to the next quickstart.

[Quickstart: Control a device connected to an IoT hub](#)

Quickstart: Send telemetry from a device to an IoT hub and read it with a back-end application (Java)

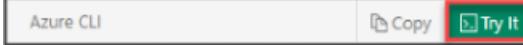
2/28/2019 • 7 minutes to read

IoT Hub is an Azure service that enables you to ingest high volumes of telemetry from your IoT devices into the cloud for storage or processing. In this quickstart, you send telemetry from a simulated device application, through IoT Hub, to a back-end application for processing.

The quickstart uses two pre-written Java applications, one to send the telemetry and one to read the telemetry from the hub. Before you run these two applications, you create an IoT hub and register a device with the hub.

Open Azure Cloud Shell

Azure Cloud Shell is a free, interactive shell that you can use to run the steps in this article. Common Azure tools are preinstalled and configured in Cloud Shell for you to use with your account. Just select the **Copy** button to copy the code, paste it in Cloud Shell, and then press Enter to run it. There are a few ways to open Cloud Shell:

Select Try It in the upper-right corner of a code block.	
Open Cloud Shell in your browser.	
Select the Cloud Shell button on the menu in the upper-right corner of the Azure portal .	

If you don't have an Azure subscription, create a [free account](#) before you begin.

Prerequisites

The two sample applications you run in this quickstart are written using Java. You need Java SE 8 or later on your development machine.

You can download Java for multiple platforms from [Oracle](#).

You can verify the current version of Java on your development machine using the following command:

```
java -version
```

To build the samples, you need to install Maven 3. You can download Maven for multiple platforms from [Apache Maven](#).

You can verify the current version of Maven on your development machine using the following command:

```
mvn --version
```

Download the sample Java project from <https://github.com/Azure-Samples/azure-iot-samples>

[java/archive/master.zip](#) and extract the ZIP archive.

Create an IoT hub

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose **+Create a resource**, then choose **Internet of Things**.
3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.

The screenshot shows the 'Basics' step of the IoT Hub creation wizard. The 'PROJECT DETAILS' section is highlighted with a red box. It contains fields for Subscription (set to Microsoft Azure Internal Consumption), Resource Group (set to Create new, with 'contoso-hub-rgrp' selected), Region (set to West US), and IoT Hub Name (set to contoso-test-hub). Below the form is a navigation bar with three buttons: 'Review + create' (disabled), 'Next: Size and scale >', and 'Automation options'.

Fill in the fields.

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Review + create

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS

* Pricing and scale tier [?](#) S1: Standard tier [▼](#) [Learn how to choose the right IoT Hub tier for your solution](#)

Number of S1 IoT Hub units [?](#) 1 This determines your IoT Hub scale capability and can be changed as your need increases.

Device-to-cloud-messages [?](#) Enabled
Message routing [?](#) Enabled
Cloud-to-device commands [?](#) Enabled
IoT Edge [?](#) Enabled
Device management [?](#) Enabled

Advanced Settings

Device-to-cloud partitions [?](#) 4

[Review + create](#) [« Previous: Basics](#) [Automation options](#)

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: The number of messages allowed per unit per day depends on your hub's pricing tier. For example, if you want the IoT hub to support ingress of 700,000 messages, you choose two S1 tier units.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

Advanced / Device-to-cloud partitions: This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most IoT hubs only need four partitions.

5. Click **Review + create** to review your choices. You see something similar to this screen.

The screenshot shows the 'Review + create' step of the Azure IoT hub creation wizard. It displays basic information like subscription, resource group, region, and IoT Hub name, as well as size and scale details including pricing tier, number of units, messages per day, and cost per month. At the bottom, there are 'Create' and 'Automation options' buttons, with 'Create' being highlighted by a red box.

6. Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Register a device

A device must be registered with your IoT hub before it can connect. In this quickstart, you use the Azure Cloud Shell to register a simulated device.

1. Run the following commands in Azure Cloud Shell to add the IoT Hub CLI extension and to create the device identity.

YourIoTHubName: Replace this placeholder below with the name you choose for your IoT hub.

MyJavaDevice: The name of the device you're registering. Use **MyJavaDevice** as shown. If you choose a different name for your device, you need to use that name throughout this article, and update the device name in the sample applications before you run them.

```
az extension add --name azure-cli-iot-ext
az iot hub device-identity create --hub-name YourIoTHubName --device-id MyJavaDevice
```

2. Run the following commands in Azure Cloud Shell to get the *device connection string* for the device you just registered: **YourIoTHubName: Replace this placeholder below with the name you choose for your IoT hub.

```
az iot hub device-identity show-connection-string --hub-name YourIoTHubName --device-id MyJavaDevice
--output table
```

Make a note of the device connection string, which looks like:

```
HostName={YourIoTHubName}.azure-devices.net;DeviceId=MyNodeDevice;SharedAccessKey=
{YourSharedAccessKey}
```

You use this value later in the quickstart.

3. You also need the *Event Hubs-compatible endpoint*, *Event Hubs-compatible path*, and *iothubowner*

primary key from your IoT hub to enable the back-end application to connect to your IoT hub and retrieve the messages. The following commands retrieve these values for your IoT hub:

**YourIoTHubName: Replace this placeholder below with the name you choose for your IoT hub.

```
az iot hub show --query properties.eventHubEndpoints.events.endpoint --name YourIoTHubName  
az iot hub show --query properties.eventHubEndpoints.events.path --name YourIoTHubName  
az iot hub policy show --name iothubowner --query primaryKey --hub-name YourIoTHubName
```

Make a note of these three values, which you use later in the quickstart.

Send simulated telemetry

The simulated device application connects to a device-specific endpoint on your IoT hub and sends simulated temperature and humidity telemetry.

1. In a local terminal window, navigate to the root folder of the sample Java project. Then navigate to the **iot-hub\Quickstarts\simulated-device** folder.
2. Open the **src/main/java/com/microsoft/docs/iothub/samples/SimulatedDevice.java** file in a text editor of your choice.

Replace the value of the `connString` variable with the device connection string you made a note of previously. Then save your changes to **SimulatedDevice.java** file.

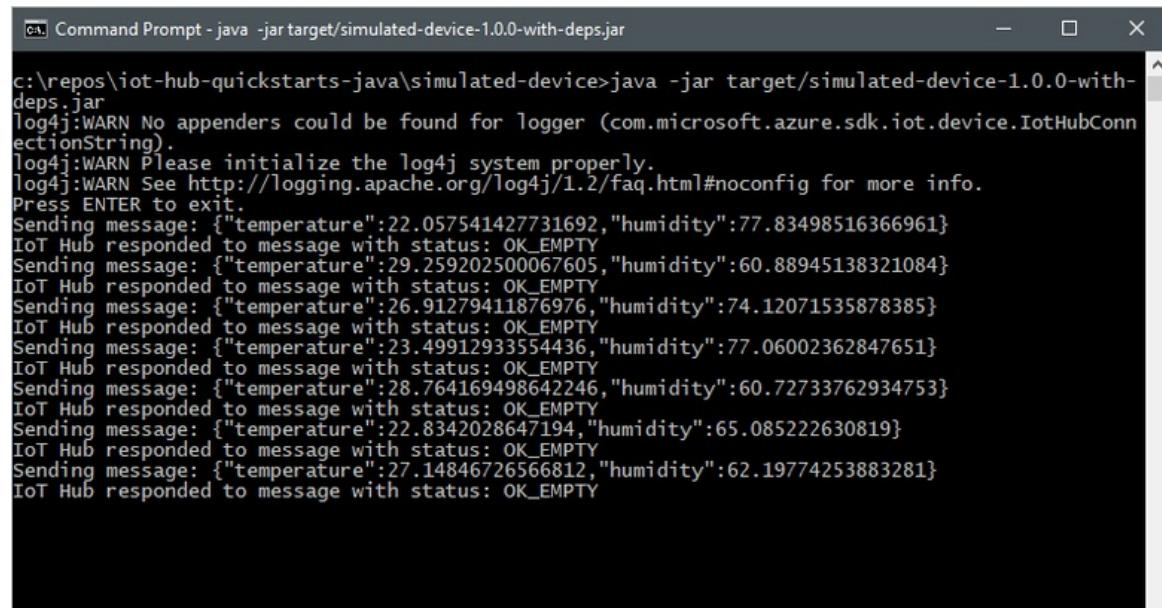
3. In the local terminal window, run the following commands to install the required libraries and build the simulated device application:

```
mvn clean package
```

4. In the local terminal window, run the following commands to run the simulated device application:

```
java -jar target/simulated-device-1.0.0-with-deps.jar
```

The following screenshot shows the output as the simulated device application sends telemetry to your IoT hub:



The screenshot shows a Windows Command Prompt window titled "Command Prompt - java -jar target/simulated-device-1.0.0-with-deps.jar". The window displays the execution of a Java application. The application logs several messages, including log4j warnings about appenders and configuration, and then sends multiple temperature and humidity messages to an IoT Hub, receiving responses for each message. The output is as follows:

```
c:\repos\iot-hub-quickstarts-java\simulated-device>java -jar target/simulated-device-1.0.0-with-deps.jar  
Log4j:WARN No appenders could be found for logger (com.microsoft.azure.sdk.iot.device.IotHubConnectionString).  
Log4j:WARN Please initialize the log4j system properly.  
Log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.  
Press ENTER to exit.  
Sending message: {"temperature":22.057541427731692,"humidity":77.83498516366961}  
IoT Hub responded to message with status: OK_EMPTY  
Sending message: {"temperature":29.259202500067605,"humidity":60.88945138321084}  
IoT Hub responded to message with status: OK_EMPTY  
Sending message: {"temperature":26.91279411876976,"humidity":74.12071535878385}  
IoT Hub responded to message with status: OK_EMPTY  
Sending message: {"temperature":23.49912933554436,"humidity":77.06002362847651}  
IoT Hub responded to message with status: OK_EMPTY  
Sending message: {"temperature":28.764169498642246,"humidity":60.72733762934753}  
IoT Hub responded to message with status: OK_EMPTY  
Sending message: {"temperature":22.8342028647194,"humidity":65.085222630819}  
IoT Hub responded to message with status: OK_EMPTY  
Sending message: {"temperature":27.14846726566812,"humidity":62.19774253883281}  
IoT Hub responded to message with status: OK_EMPTY
```

Read the telemetry from your hub

The back-end application connects to the service-side **Events** endpoint on your IoT Hub. The application receives the device-to-cloud messages sent from your simulated device. An IoT Hub back-end application typically runs in the cloud to receive and process device-to-cloud messages.

1. In another local terminal window, navigate to the root folder of the sample Java project. Then navigate to the **iot-hub\Quickstarts\read-d2c-messages** folder.
2. Open the **src/main/java/com/microsoft/docs/iothub/samples/ReadDeviceToCloudMessages.java** file in a text editor of your choice. Update the following variables and save your changes to the file.

VARIABLE	VALUE
<code>eventHubsCompatibleEndpoint</code>	Replace the value of the variable with the Event Hubs-compatible endpoint you made a note of previously.
<code>eventHubsCompatiblePath</code>	Replace the value of the variable with the Event Hubs-compatible path you made a note of previously.
<code>iotHubSasKey</code>	Replace the value of the variable with the iothubowner primary key you made a note of previously.

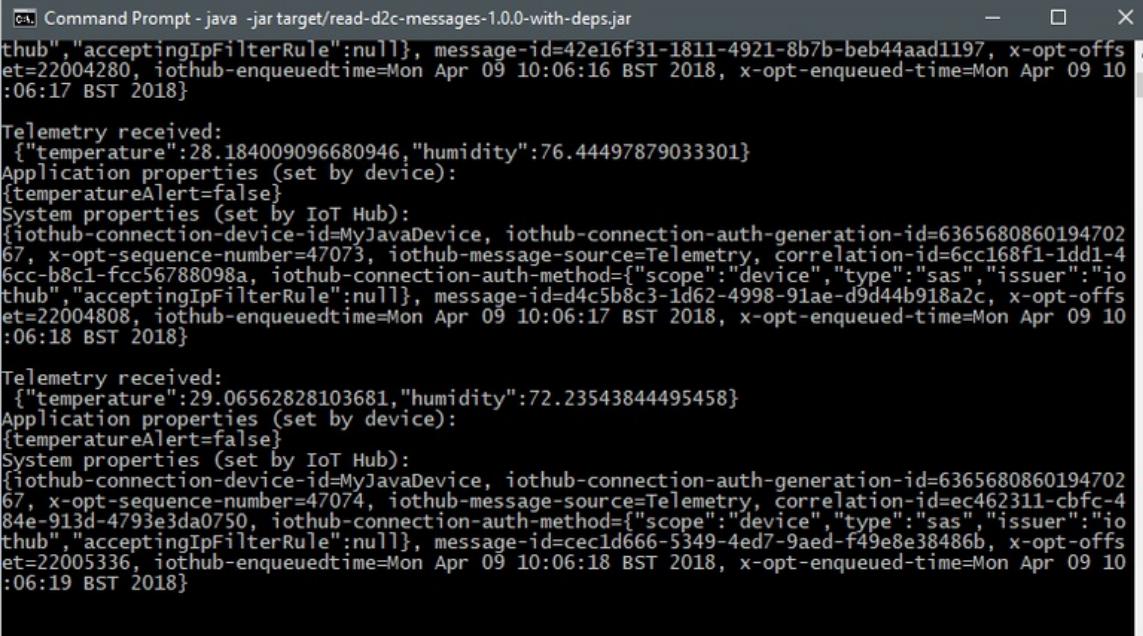
3. In the local terminal window, run the following commands to install the required libraries and build the back-end application:

```
mvn clean package
```

4. In the local terminal window, run the following commands to run the back-end application:

```
java -jar target/read-d2c-messages-1.0.0-with-deps.jar
```

The following screenshot shows the output as the back-end application receives telemetry sent by the simulated device to the hub:



```
Command Prompt - java -jar target/read-d2c-messages-1.0.0-with-deps.jar

```

Clean up resources

If you will be continuing to the next recommended article, you can keep the resources you've already created and reuse them.

Otherwise, you can delete the Azure resources created in this article to avoid charges.

IMPORTANT

Deleting a resource group is irreversible. The resource group and all the resources contained in it are permanently deleted. Make sure that you do not accidentally delete the wrong resource group or resources. If you created the IoT Hub inside an existing resource group that contains resources you want to keep, only delete the IoT Hub resource itself instead of deleting the resource group.

To delete a resource group by name:

1. Sign in to the [Azure portal](#) and click **Resource groups**.
2. In the **Filter by name...** textbox, type the name of the resource group containing your IoT Hub.
3. To the right of your resource group in the result list, click ... then **Delete resource group**.

The screenshot shows the Microsoft Azure portal interface. On the left sidebar, the 'Resource groups' option is highlighted with a red box. In the main content area, the 'Resource groups' blade is open, displaying a list of resource groups. A search bar at the top has 'TestResources' typed into it, also highlighted with a red box. Below the search bar, there are filter options: 'Prototype3', 'All locations', and 'No grouping'. The list shows one item: 'TestResources' under 'Prototype3'. To the right of this item is a 'Delete resource group' button, which is also highlighted with a red box. The overall interface is dark-themed.

4. You will be asked to confirm the deletion of the resource group. Type the name of your resource group again to confirm, and then click **Delete**. After a few moments, the resource group and all of its contained resources are deleted.

Next steps

In this quickstart, you've setup an IoT hub, registered a device, sent simulated telemetry to the hub using a Java application, and read the telemetry from the hub using a simple back-end application.

To learn how to control your simulated device from a back-end application, continue to the next quickstart.

[Quickstart: Control a device connected to an IoT hub](#)

Quickstart: Send telemetry from a device to an IoT hub and read it with a back-end application (Python)

3/1/2019 • 7 minutes to read

IoT Hub is an Azure service that enables you to ingest high volumes of telemetry from your IoT devices into the cloud for storage or processing. In this quickstart, you send telemetry from a simulated device application, through IoT Hub, to a back-end application for processing.

The quickstart uses a pre-written Python application to send the telemetry and a CLI utility to read the telemetry from the hub. Before you run these two applications, you create an IoT hub and register a device with the hub.

Open Azure Cloud Shell

Azure Cloud Shell is a free, interactive shell that you can use to run the steps in this article. Common Azure tools are preinstalled and configured in Cloud Shell for you to use with your account. Just select the **Copy** button to copy the code, paste it in Cloud Shell, and then press Enter to run it. There are a few ways to open Cloud Shell:

Select Try It in the upper-right corner of a code block.	
Open Cloud Shell in your browser.	
Select the Cloud Shell button on the menu in the upper-right corner of the Azure portal .	

If you don't have an Azure subscription, create a [free account](#) before you begin.

Prerequisites

The two sample applications you run in this quickstart are written using Python. Currently, the Microsoft Azure IoT SDKs for Python support only specific versions of Python for each platform. To learn more, see the [Python SDK Readme](#).

This quickstart assumes you are using a Windows development machine. For Windows systems, only [Python 3.6.x](#) is supported. The Python installer you choose should be based on the architecture of the system that you are working with. If your system CPU architecture is 32 bit, then download the x86 installer; for the 64bit architecture, download the x86-64 installer. Additionally, make sure the [Microsoft Visual C++ Redistributable for Visual Studio 2017](#) is installed for your architecture (x86 or x64).

You can download Python for other platforms from [Python.org](#).

You can verify the current version of Python on your development machine using one of the following commands:

```
python --version
```

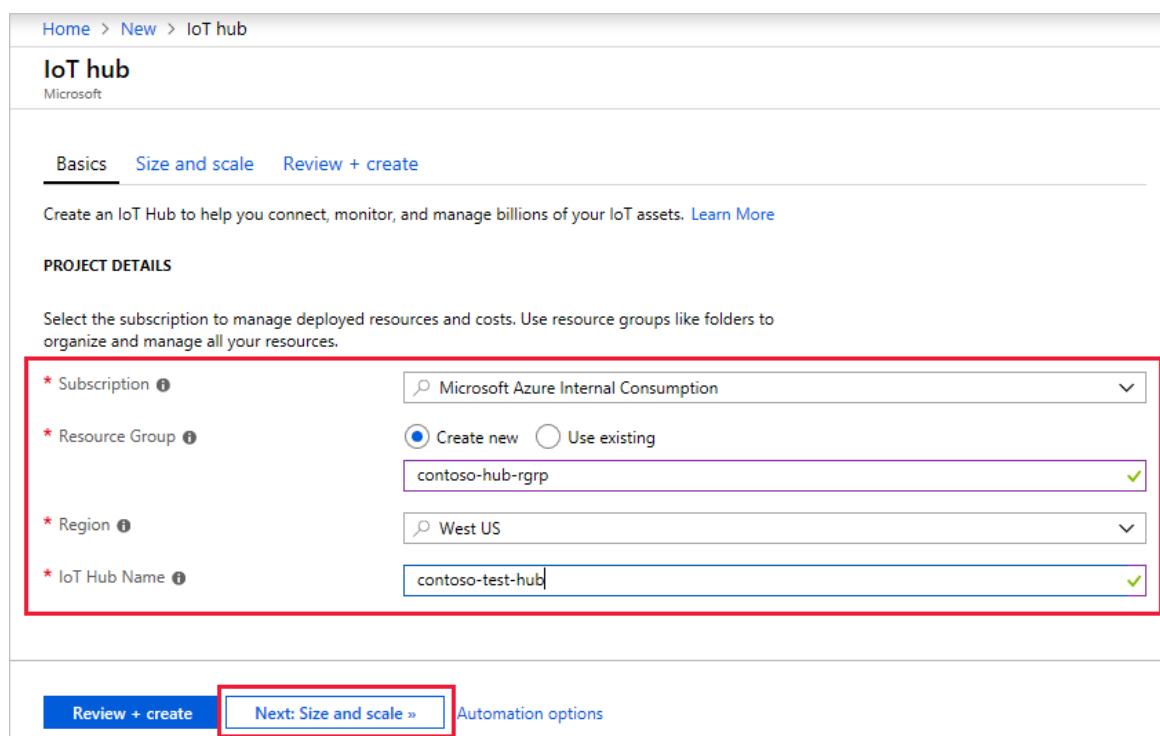
```
python3 --version
```

Download the sample Python project from <https://github.com/Azure-Samples/azure-iot-samples-python/archive/master.zip> and extract the ZIP archive.

Create an IoT hub

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose **+Create a resource**, then choose **Internet of Things**.
3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.



Fill in the fields.

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

- Click **Next: Size and scale** to continue creating your IoT hub.

The screenshot shows the 'Size and scale' tab of the Azure IoT Hub creation wizard. It includes sections for selecting the pricing tier (S1: Standard tier), choosing the number of units (1 unit selected), and enabling various features like Device-to-cloud-messages, Message routing, Cloud-to-device commands, IoT Edge, and Device management. An 'Advanced Settings' section is also visible at the bottom.

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS

* Pricing and scale tier [?](#) S1: Standard tier [?](#) Learn how to choose the right IoT Hub tier for your solution

Number of S1 IoT Hub units [?](#) 1 This determines your IoT Hub scale capability and can be changed as your need increases.

Device-to-cloud-messages [?](#) Enabled
Message routing [?](#) Enabled
Cloud-to-device commands [?](#) Enabled
IoT Edge [?](#) Enabled
Device management [?](#) Enabled

Advanced Settings

Device-to-cloud partitions [?](#) 4

Review + create [« Previous: Basics](#) Automation options

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: The number of messages allowed per unit per day depends on your hub's pricing tier. For example, if you want the IoT hub to support ingress of 700,000 messages, you choose two S1 tier units.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

Advanced / Device-to-cloud partitions: This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most IoT hubs only need four partitions.

- Click **Review + create** to review your choices. You see something similar to this screen.

The screenshot shows the 'Review + create' step of the Azure IoT hub creation wizard. It displays basic information like subscription, resource group, region, and IoT Hub name, as well as size and scale details including pricing tier, number of units, messages per day, and cost per month. At the bottom, there are 'Create' and 'Automation options' buttons.

6. Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Register a device

A device must be registered with your IoT hub before it can connect. In this quickstart, you use the Azure Cloud Shell to register a simulated device.

1. Run the following commands in Azure Cloud Shell to add the IoT Hub CLI extension and to create the device identity.

YourIoTHubName : Replace this placeholder below with the name you choose for your IoT hub.

MyPythonDevice : This is the name given for the registered device. Use MyPythonDevice as shown. If you choose a different name for your device, you will also need to use that name throughout this article, and update the device name in the sample applications before you run them.

```
az extension add --name azure-cli-iot-ext
az iot hub device-identity create --hub-name YourIoTHubName --device-id MyPythonDevice
```

2. Run the following commands in Azure Cloud Shell to get the *device connection string* for the device you just registered:

YourIoTHubName : Replace this placeholder below with the name you choose for your IoT hub.

```
az iot hub device-identity show-connection-string --hub-name YourIoTHubName --device-id
MyPythonDevice --output table
```

Make a note of the device connection string, which looks like:

```
HostName={YourIoTHubName}.azure-devices.net;DeviceId=MyNodeDevice;SharedAccessKey=
{YourSharedAccessKey}
```

You use this value later in the quickstart.

Send simulated telemetry

The simulated device application connects to a device-specific endpoint on your IoT hub and sends simulated temperature and humidity telemetry.

1. In a local terminal window, navigate to the root folder of the sample Python project. Then navigate to the **iot-hub\Quickstarts\simulated-device** folder.
2. Open the **SimulatedDevice.py** file in a text editor of your choice.

Replace the value of the `CONNECTION_STRING` variable with the device connection string you made a note of previously. Then save your changes to **SimulatedDevice.py** file.

3. In the local terminal window, run the following commands to install the required libraries for the simulated device application:

```
pip install azure-iothub-device-client
```

4. In the local terminal window, run the following commands to run the simulated device application:

```
python SimulatedDevice.py
```

The following screenshot shows the output as the simulated device application sends telemetry to your IoT hub:

```
c:\repos\iot-hub-quickstarts-python\simulated-device>python SimulatedDevice.py
IoT Hub Quickstart #1 - Simulated device
Press Ctrl-C to exit
IoT Hub device sending periodic messages, press Ctrl-C to exit
Sending message: {"temperature": 25.57, "humidity": 71.57}
Sending message: {"temperature": 21.78, "humidity": 63.41}
IoT Hub responded to message with status: OK
IoT Hub responded to message with status: OK
Sending message: {"temperature": 28.86, "humidity": 71.56}
IoT Hub responded to message with status: OK
Sending message: {"temperature": 30.92, "humidity": 74.79}
IoT Hub responded to message with status: OK
Sending message: {"temperature": 20.50, "humidity": 73.00}
IoT Hub responded to message with status: OK
Sending message: {"temperature": 24.16, "humidity": 64.02}
IoT Hub responded to message with status: OK
Sending message: {"temperature": 31.89, "humidity": 73.26}
IoT Hub responded to message with status: OK
Sending message: {"temperature": 34.25, "humidity": 62.60}
IoT Hub responded to message with status: OK
Sending message: {"temperature": 26.87, "humidity": 72.21}
IoT Hub responded to message with status: OK
Sending message: {"temperature": 23.57, "humidity": 66.02}
IoT Hub responded to message with status: OK
Sending message: {"temperature": 20.59, "humidity": 70.30}
IoT Hub responded to message with status: OK
```

Read the telemetry from your hub

The IoT Hub CLI extension can connect to the service-side **Events** endpoint on your IoT Hub. The extension receives the device-to-cloud messages sent from your simulated device. An IoT Hub back-end application typically runs in the cloud to receive and process device-to-cloud messages.

Run the following commands in Azure Cloud Shell, replacing `YourIoTHubName` with the name of your IoT hub:

```
az iot hub monitor-events --hub-name YourIoTHubName --device-id MyPythonDevice
```

The following screenshot shows the output as the extension receives telemetry sent by the simulated device to the hub:

Azure Cloud Shell window showing event monitoring output. The text in the terminal pane reads:

```
Starting event monitor, filtering on device: MyPythonDevice, use ctrl-c to stop...
{
  "event": {
    "origin": "MyPythonDevice",
    "payload": "{\"temperature\": 27.46,\"humidity\": 70.67}"
  }
}
{
  "event": {
    "origin": "MyPythonDevice",
    "payload": "{\"temperature\": 30.86,\"humidity\": 66.13}"
  }
}
{
  "event": {
    "origin": "MyPythonDevice",
    "payload": "{\"temperature\": 29.33,\"humidity\": 62.86}"
  }
}
{
  "event": {
    "origin": "MyPythonDevice",
    "payload": "{\"temperature\": 20.44,\"humidity\": 75.25}"
  }
}
{
  "event": {
    "origin": "MyPythonDevice",
    "payload": "{\"temperature\": 29.98,\"humidity\": 78.20}"
  }
}
```

Clean up resources

If you will be continuing to the next recommended article, you can keep the resources you've already created and reuse them.

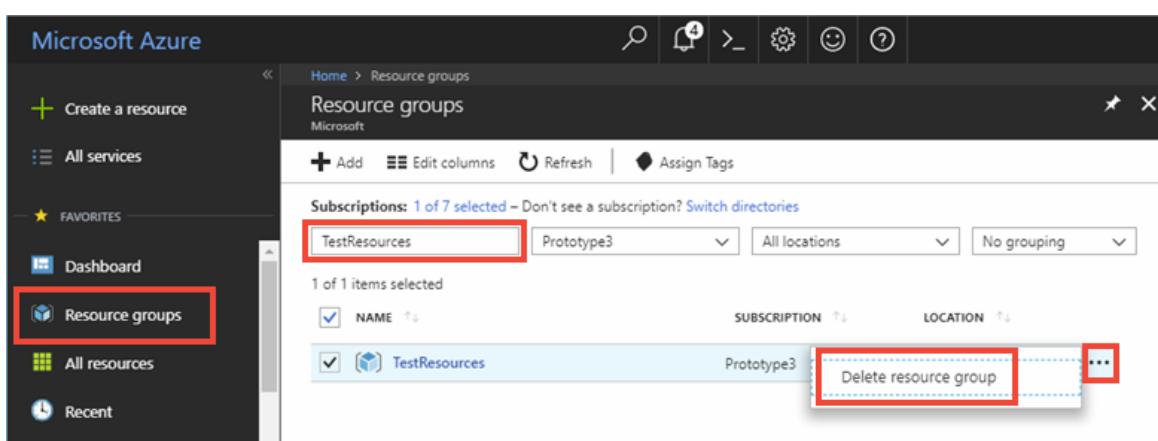
Otherwise, you can delete the Azure resources created in this article to avoid charges.

IMPORTANT

Deleting a resource group is irreversible. The resource group and all the resources contained in it are permanently deleted. Make sure that you do not accidentally delete the wrong resource group or resources. If you created the IoT Hub inside an existing resource group that contains resources you want to keep, only delete the IoT Hub resource itself instead of deleting the resource group.

To delete a resource group by name:

1. Sign in to the [Azure portal](#) and click **Resource groups**.
2. In the **Filter by name...** textbox, type the name of the resource group containing your IoT Hub.
3. To the right of your resource group in the result list, click ... then **Delete resource group**.



4. You will be asked to confirm the deletion of the resource group. Type the name of your resource group again to confirm, and then click **Delete**. After a few moments, the resource group and all of its contained resources are deleted.

Next steps

In this quickstart, you've setup an IoT hub, registered a device, sent simulated telemetry to the hub using a Python application, and read the telemetry from the hub using a simple back-end application.

To learn how to control your simulated device from a back-end application, continue to the next quickstart.

[Quickstart: Control a device connected to an IoT hub](#)

Quickstart: Send IoT telemetry from an Android device

3/10/2019 • 7 minutes to read

IoT Hub is an Azure service that enables you to ingest high volumes of telemetry from your IoT devices into the cloud for storage or processing. In this quickstart, you send telemetry to an IoT Hub from an Android application running on a physical or simulated device.

The quickstart uses a pre-written Android application to send the telemetry. The telemetry will be read from the IoT Hub using the Azure Cloud Shell. Before you run the application, you create an IoT hub and register a device with the hub.

Open Azure Cloud Shell

Azure Cloud Shell is a free, interactive shell that you can use to run the steps in this article. Common Azure tools are preinstalled and configured in Cloud Shell for you to use with your account. Just select the **Copy** button to copy the code, paste it in Cloud Shell, and then press Enter to run it. There are a few ways to open Cloud Shell:

Select Try It in the upper-right corner of a code block.	
Open Cloud Shell in your browser.	
Select the Cloud Shell button on the menu in the upper-right corner of the Azure portal .	

If you don't have an Azure subscription, create a [free account](#) before you begin.

Prerequisites

- Android studio from <https://developer.android.com/studio/>. For more information regarding Android Studio installation, see [android-installation](#).
- Android SDK 27 is used by the sample in this article.
- The [sample Android application](#) you run in this quickstart is part of the `azure-iot-samples-java` repository on GitHub. Download or clone the [azure-iot-samples-java](#) repository.

Create an IoT hub

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose **+Create a resource**, then choose **Internet of Things**.
3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Review + create

Create an IoT Hub to help you connect, monitor, and manage billions of your IoT assets. [Learn More](#)

PROJECT DETAILS

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

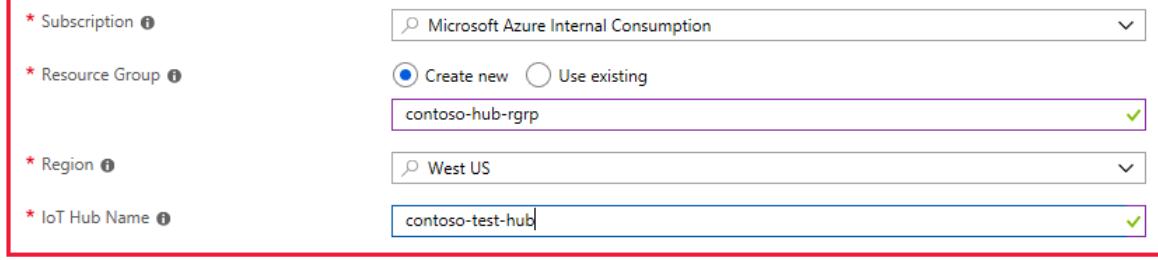
* Subscription [?](#) Microsoft Azure Internal Consumption

* Resource Group [?](#) Create new Use existing contoso-hub-rgrp

* Region [?](#) West US

* IoT Hub Name [?](#) contoso-test-hub

[Review + create](#) [Next: Size and scale »](#) Automation options



Fill in the fields.

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Review + create

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS

* Pricing and scale tier [?](#) S1: Standard tier [?](#) Learn how to choose the right IoT Hub tier for your solution

Number of S1 IoT Hub units [?](#) 1 This determines your IoT Hub scale capability and can be changed as your need increases.

Device-to-cloud-messages [?](#) Enabled

Message routing [?](#) Enabled

Cloud-to-device commands [?](#) Enabled

IoT Edge [?](#) Enabled

Device management [?](#) Enabled

Advanced Settings

Device-to-cloud partitions [?](#) 4

Review + create [« Previous: Basics](#) Automation options

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: The number of messages allowed per unit per day depends on your hub's pricing tier. For example, if you want the IoT hub to support ingress of 700,000 messages, you choose two S1 tier units.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

Advanced / Device-to-cloud partitions: This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most IoT hubs only need four partitions.

5. Click **Review + create** to review your choices. You see something similar to this screen.

The screenshot shows the 'Review + create' step of creating an IoT hub. It displays basic information like subscription, resource group, region, and IoT Hub Name. Under 'SIZE AND SCALE', it shows the pricing tier (S1), number of units (1), messages per day (400,000), and monthly cost (25.00 USD). At the bottom, there are buttons for 'Create' (highlighted with a red box) and 'Automation options'.

6. Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Register a device

A device must be registered with your IoT hub before it can connect. In this quickstart, you use the Azure Cloud Shell to register a simulated device.

1. Run the following commands in Azure Cloud Shell to add the IoT Hub CLI extension and to create the device identity.

YourIoTHubName: Replace this placeholder below with the name you choose for your IoT hub.

MyAndroidDevice: MyAndroidDevice is the name given for the registered device. Use MyAndroidDevice as shown. If you choose a different name for your device, you will also need to use that name throughout this article, and update the device name in the sample applications before you run them.

```
az extension add --name azure-cli-iot-ext
az iot hub device-identity create --hub-name YourIoTHubName --device-id MyAndroidDevice
```

2. Run the following commands in Azure Cloud Shell to get the *device connection string* for the device you just registered:

YourIoTHubName: Replace this placeholder below with the name you choose for your IoT hub.

```
az iot hub device-identity show-connection-string --hub-name YourIoTHubName --device-id
MyAndroidDevice --output table
```

Make a note of the device connection string, which looks like:

```
HostName={YourIoTHubName}.azure-devices.net;DeviceId=MyAndroidDevice;SharedAccessKey=
{YourSharedAccessKey}
```

You use this value later in this quickstart to send telemetry.

Send telemetry

1. Open the GitHub sample Android project in Android Studio. The project is located in the following directory of your cloned or downloaded copy of [azure-iot-sample-java](#) repository.

```
\azure-iot-samples-java\iot-hub\Samples\device\AndroidSample
```

2. In Android Studio, open *gradle.properties* for the sample project and replace the **Device_Connection_String** placeholder with your device connection string you noted earlier.

```
DeviceConnectionString=HostName={YourIoTHubName}.azure-devices.net;DeviceId=MyNodeDevice;SharedAccessKey={YourSharedAccessKey}
```

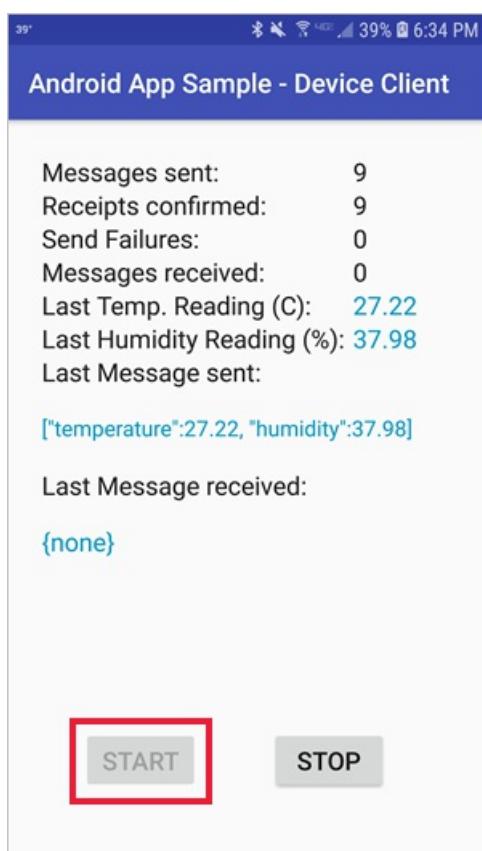
3. In Android Studio, click **File > Sync Project with Gradle Files**. Verify the build completes.

NOTE

If the project sync fails, it may be for one of the following reasons:

- The versions of the Android Gradle plugin and Gradle referenced in the project are out of date for your version of Android Studio. Follow [these instructions](#) to reference and install the correct versions of the plugin and Gradle for your installation.
- The license agreement for the Android SDK has not been signed. Follow the instructions in the Build output to sign the license agreement and download the SDK.

4. Once the build has completed, click **Run > Run 'app'**. Configure the app to run on a physical Android device or an Android emulator. For more information on running an Android app on a physical device or emulator, see [Run your app](#).
5. Once the app loads, click the **Start** button to start sending telemetry to your IoT Hub:



Read the telemetry from your hub

In this section, you will use the Azure Cloud Shell with the [IoT extension](#) to monitor the device messages that are sent by the Android device.

1. Using the Azure Cloud Shell, run the following command to connect and read messages from your IoT hub:

YourIoTHubName: Replace this placeholder below with the name you choose for your IoT hub.

```
az iot hub monitor-events --hub-name YourIoTHubName --output table
```

The following screenshot shows the output as the IoT hub receives telemetry sent by the Android device:

```
UserName@Azure:~$ az iot hub monitor-events --hub-name JavaTesting --output table
Starting event monitor, use ctrl-c to stop...
event:
  origin: MyAndroidDevice
  payload: '{"temperature":20.16, "humidity":35.21}'

event:
  origin: MyAndroidDevice
  payload: '{"temperature":23.92, "humidity":40.28}'

event:
  origin: MyAndroidDevice
  payload: '{"temperature":21.90, "humidity":35.45}'

event:
  origin: MyAndroidDevice
  payload: '{"temperature":22.87, "humidity":39.04}'

event:
  origin: MyAndroidDevice
  payload: '{"temperature":25.31, "humidity":43.22}'
```

Clean up resources

If you will be continuing to the next recommended article, you can keep the resources you've already created and reuse them.

Otherwise, you can delete the Azure resources created in this article to avoid charges.

IMPORTANT

Deleting a resource group is irreversible. The resource group and all the resources contained in it are permanently deleted. Make sure that you do not accidentally delete the wrong resource group or resources. If you created the IoT Hub inside an existing resource group that contains resources you want to keep, only delete the IoT Hub resource itself instead of deleting the resource group.

To delete a resource group by name:

1. Sign in to the [Azure portal](#) and click **Resource groups**.
2. In the **Filter by name...** textbox, type the name of the resource group containing your IoT Hub.
3. To the right of your resource group in the result list, click ... then **Delete resource group**.

The screenshot shows the Microsoft Azure portal interface. On the left, there's a sidebar with options like 'Create a resource', 'All services', 'FAVORITES' (which includes 'Dashboard', 'Resource groups' [highlighted with a red box], 'All resources', and 'Recent'), and a search bar. The main content area is titled 'Resource groups' and shows a table with one row. The table has columns for 'NAME' (with 'TestResources' checked), 'SUBSCRIPTION' (Prototype3), and 'LOCATION'. To the right of the 'NAME' column, there's a 'Delete resource group' button (highlighted with a red box) and a three-dot ellipsis button.

4. You will be asked to confirm the deletion of the resource group. Type the name of your resource group again to confirm, and then click **Delete**. After a few moments, the resource group and all of its contained resources are deleted.

Next steps

In this quickstart, you've setup an IoT hub, registered a device, sent simulated telemetry to the hub using an Android application, and read the telemetry from the hub using the Azure Cloud Shell.

To learn how to control your simulated device from a back-end application, continue to the next quickstart.

[Quickstart: Control a device connected to an IoT hub](#)

Quickstart: Send telemetry from a device to an IoT hub (iOS)

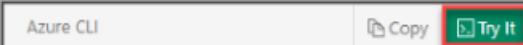
2/28/2019 • 7 minutes to read

IoT Hub is an Azure service that enables you to ingest high volumes of telemetry from your IoT devices into the cloud for storage or processing. In this article, you send telemetry from a simulated device application to IoT Hub. Then you can view the data from a back-end application.

This article uses a pre-written Swift application to send the telemetry and a CLI utility to read the telemetry from IoT Hub.

Open Azure Cloud Shell

Azure Cloud Shell is a free, interactive shell that you can use to run the steps in this article. Common Azure tools are preinstalled and configured in Cloud Shell for you to use with your account. Just select the **Copy** button to copy the code, paste it in Cloud Shell, and then press Enter to run it. There are a few ways to open Cloud Shell:

Select Try It in the upper-right corner of a code block.	
Open Cloud Shell in your browser.	
Select the Cloud Shell button on the menu in the upper-right corner of the Azure portal .	

If you don't have an Azure subscription, create a [free account](#) before you begin.

Prerequisites

- Download the code sample from [Azure samples](#)
- The latest version of [XCode](#), running the latest version of the iOS SDK. This quickstart was tested with XCode 9.3 and iOS 11.3.
- The latest version of [CocoaPods](#).

Create an IoT hub

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose +**Create a resource**, then choose **Internet of Things**.
3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Review + create

Create an IoT Hub to help you connect, monitor, and manage billions of your IoT assets. [Learn More](#)

PROJECT DETAILS

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

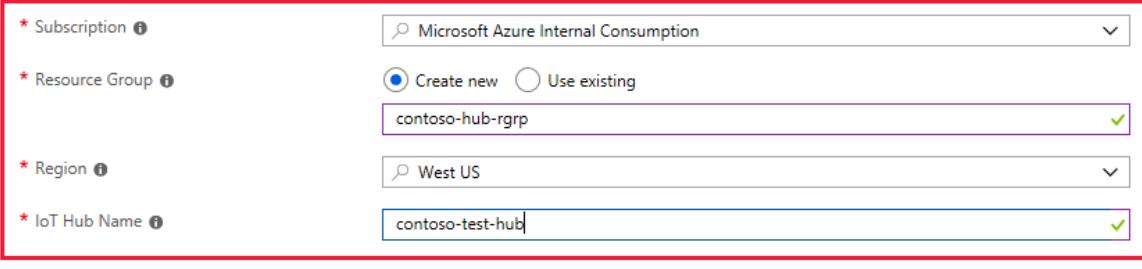
* Subscription [?](#) Microsoft Azure Internal Consumption

* Resource Group [?](#) Create new Use existing contoso-hub-rgrp

* Region [?](#) West US

* IoT Hub Name [?](#) contoso-test-hub

Review + create **Next: Size and scale »** Automation options



Fill in the fields.

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Review + create

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS

* Pricing and scale tier [?](#) S1: Standard tier [?](#) [Learn how to choose the right IoT Hub tier for your solution](#)

Number of S1 IoT Hub units [?](#) 1 [1](#) This determines your IoT Hub scale capability and can be changed as your need increases.

Device-to-cloud-messages [?](#) Enabled

Message routing [?](#) Enabled

Cloud-to-device commands [?](#) Enabled

IoT Edge [?](#) Enabled

Device management [?](#) Enabled

Advanced Settings

Device-to-cloud partitions [?](#) 4 [4](#)

[Review + create](#) [« Previous: Basics](#) [Automation options](#)

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: The number of messages allowed per unit per day depends on your hub's pricing tier. For example, if you want the IoT hub to support ingress of 700,000 messages, you choose two S1 tier units.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

Advanced / Device-to-cloud partitions: This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most IoT hubs only need four partitions.

5. Click **Review + create** to review your choices. You see something similar to this screen.

The screenshot shows the 'Review + create' step of the Azure IoT hub creation wizard. It displays basic information like subscription, resource group, region, and IoT hub name, followed by size and scale details including pricing tier, number of units, messages per day, and monthly cost. At the bottom, there are 'Create' and 'Automation options' buttons.

Subscription	Microsoft Azure Internal Consumption
Resource Group	contoso-hub-rgrp
Region	West US
IoT Hub Name	contoso-test-hub
SIZE AND SCALE	
Pricing and scale tier	S1
Number of S1 IoT Hub units	1
Messages per day	400,000
Cost per month	25.00 USD

Create « Previous: Size and scale Automation options

6. Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Register a device

A device must be registered with your IoT hub before it can connect. In this quickstart, you use the Azure Cloud Shell to register a simulated device.

1. Run the following commands in Azure Cloud Shell to add the IoT Hub CLI extension and to create the device identity.

YourIoTHubName : Replace this placeholder below with the name you choose for your IoT hub.

myiOSdevice : This is the name given for the registered device. Use myiOSdevice as shown. If you choose a different name for your device, you will also need to use that name throughout this article, and update the device name in the sample applications before you run them.

```
az extension add --name azure-cli-iot-ext
az iot hub device-identity create --hub-name YourIoTHubName --device-id myiOSdevice
```

2. Run the following command to get the *device connection string* for the device you just registered:

```
az iot hub device-identity show-connection-string --hub-name YourIoTHubName --device-id myiOSdevice -o table
```

Make a note of the device connection string, which looks like:

```
HostName={YourIoTHubName}.azure-devices.net;DeviceId=MyNodeDevice;SharedAccessKey={YourSharedAccessKey}
```

You use this value later in the quickstart.

Send simulated telemetry

The sample application runs on an iOS device, which connects to a device-specific endpoint on your IoT hub and sends simulated temperature and humidity telemetry.

Install CocoaPods

CocoaPods manage dependencies for iOS projects that use third-party libraries.

In a local terminal window, navigate to the Azure-IoT-Samples-iOS folder that you downloaded in the prerequisites. Then, navigate to the sample project:

```
cd quickstart/sample-device
```

Make sure that XCode is closed, then run the following command to install the CocoaPods that are declared in the **podfile** file:

```
pod install
```

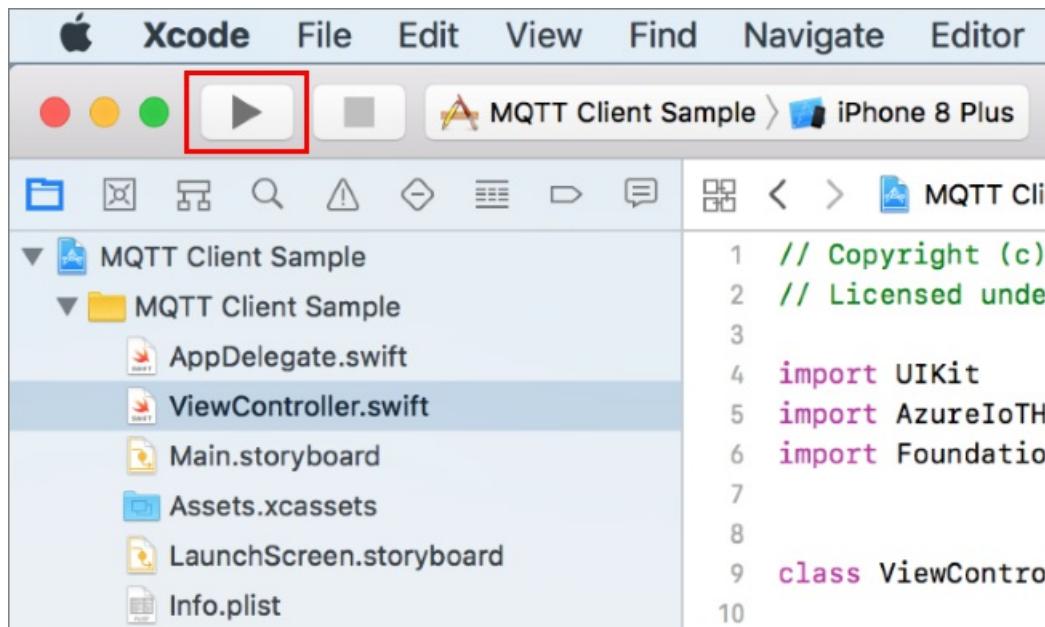
Along with installing the pods required for your project, the installation command also created an XCode workspace file that is already configured to use the pods for dependencies.

Run the sample application

1. Open the sample workspace in XCode.

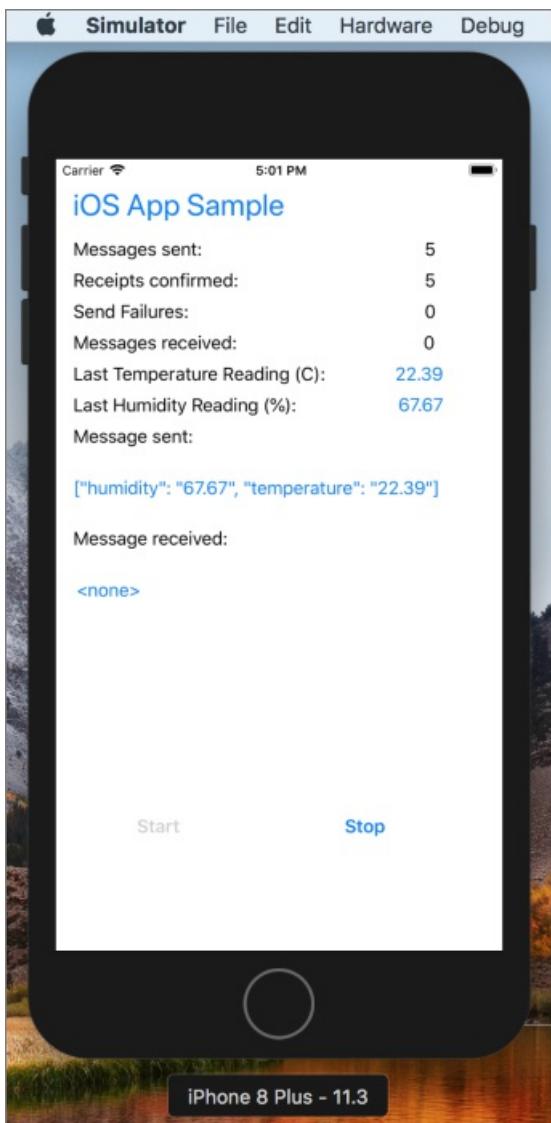
```
open "MQTT Client Sample.xcworkspace"
```

2. Expand the **MQTT Client Sample** project and then expand the folder of the same name.
3. Open **ViewController.swift** for editing in XCode.
4. Search for the **connectionString** variable and update the value with the device connection string that you made a note of previously.
5. Save your changes.
6. Run the project in the device emulator with the **Build and run** button or the key combo **command + r**.



7. When the emulator opens, select **Start** in the sample app.

The following screenshot shows some example output as the application sends simulated telemetry to your IoT hub:



Read the telemetry from your hub

The sample app that you ran on the XCode emulator shows data about messages sent from the device. You can also view the data through your IoT hub as it is received. The IoT Hub CLI extension can connect to the service-side **Events** endpoint on your IoT Hub. The extension receives the device-to-cloud messages sent from your simulated device. An IoT Hub back-end application typically runs in the cloud to receive and process device-to-cloud messages.

Run the following commands in Azure Cloud Shell, replacing `YourIoTHubName` with the name of your IoT hub:

```
az iot hub monitor-events --device-id myiOSdevice --hub-name YourIoTHubName
```

The following screenshot shows the output as the extension receives telemetry sent by the simulated device to the hub:

The following screenshot shows the type of telemetry that you see in your local terminal window:

Azure Cloud Shell window showing IoT device data. The output pane displays several events from a device named 'myiOSdevice'. Each event includes an origin, payload, and timestamp. The payloads show temperature and humidity values.

```
payload: '{"temperature": 24.94,"humidity": 76.39, "device-id": myiOSdevice}'  
event:  
origin: myiOSdevice  
payload: '{"temperature": 31.04,"humidity": 70.78, "device-id": myiOSdevice}'  
  
event:  
origin: myiOSdevice  
payload: '{"temperature": 34.53,"humidity": 61.95, "device-id": myiOSdevice}'  
  
event:  
origin: myiOSdevice  
payload: '{"temperature": 20.63,"humidity": 60.84, "device-id": myiOSdevice}'  
  
event:  
origin: myiOSdevice  
payload: '{"temperature": 27.08,"humidity": 79.13, "device-id": myiOSdevice}'  
  
event:  
origin: myiOSdevice  
payload: '{"temperature": 25.04,"humidity": 75.59, "device-id": myiOSdevice}'
```

Clean up resources

If you will be continuing to the next recommended article, you can keep the resources you've already created and reuse them.

Otherwise, you can delete the Azure resources created in this article to avoid charges.

IMPORTANT

Deleting a resource group is irreversible. The resource group and all the resources contained in it are permanently deleted. Make sure that you do not accidentally delete the wrong resource group or resources. If you created the IoT Hub inside an existing resource group that contains resources you want to keep, only delete the IoT Hub resource itself instead of deleting the resource group.

To delete a resource group by name:

1. Sign in to the [Azure portal](#) and click **Resource groups**.
2. In the **Filter by name...** textbox, type the name of the resource group containing your IoT Hub.
3. To the right of your resource group in the result list, click ... then **Delete resource group**.

The screenshot shows the Microsoft Azure portal's Resource groups blade. The left sidebar has a red box around the 'Resource groups' item. The main area shows a table with one item selected. The 'NAME' column shows 'TestResources', the 'SUBSCRIPTION' column shows 'Prototype3', and the 'LOCATION' column shows 'All locations'. A red box highlights the 'Delete resource group' button in the 'Actions' column for the selected row.

NAME	SUBSCRIPTION	LOCATION	Actions
TestResources	Prototype3	All locations	<input checked="" type="button"/> Delete resource group ...

4. You will be asked to confirm the deletion of the resource group. Type the name of your resource group

again to confirm, and then click **Delete**. After a few moments, the resource group and all of its contained resources are deleted.

Next steps

In this article, you set up an IoT hub, registered a device, sent simulated telemetry to the hub from an iOS device, and read the telemetry from the hub.

To learn how to control your simulated device from a back-end application, continue to the next quickstart.

[Quickstart: Control a device connected to an IoT hub](#)

Quickstart: Control a device connected to an IoT hub (Node.js)

2/28/2019 • 8 minutes to read

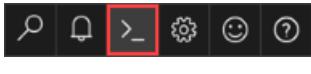
IoT Hub is an Azure service that enables you to ingest high volumes of telemetry from your IoT devices into the cloud and manage your devices from the cloud. In this quickstart, you use a *direct method* to control a simulated device connected to your IoT hub. You can use direct methods to remotely change the behavior of a device connected to your IoT hub.

The quickstart uses two pre-written Node.js applications:

- A simulated device application that responds to direct methods called from a back-end application. To receive the direct method calls, this application connects to a device-specific endpoint on your IoT hub.
- A back-end application that calls the direct methods on the simulated device. To call a direct method on a device, this application connects to service-side endpoint on your IoT hub.

Open Azure Cloud Shell

Azure Cloud Shell is a free, interactive shell that you can use to run the steps in this article. Common Azure tools are preinstalled and configured in Cloud Shell for you to use with your account. Just select the **Copy** button to copy the code, paste it in Cloud Shell, and then press Enter to run it. There are a few ways to open Cloud Shell:

Select Try It in the upper-right corner of a code block.	
Open Cloud Shell in your browser.	
Select the Cloud Shell button on the menu in the upper-right corner of the Azure portal .	

If you don't have an Azure subscription, create a [free account](#) before you begin.

Prerequisites

The two sample applications you run in this quickstart are written using Node.js. You need Node.js v4.x.x or later on your development machine.

You can download Node.js for multiple platforms from nodejs.org.

You can verify the current version of Node.js on your development machine using the following command:

```
node --version
```

If you haven't already done so, download the sample Node.js project from <https://github.com/Azure-Samples/azure-iot-samples-node/archive/master.zip> and extract the ZIP archive.

Create an IoT hub

If you completed the previous [Quickstart: Send telemetry from a device to an IoT hub](#), you can skip this step.

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose **+Create a resource**, then choose **Internet of Things**.
3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.

The screenshot shows the 'Basics' step of the IoT hub creation wizard. The 'PROJECT DETAILS' section is highlighted with a red box. Inside this box, the following fields are filled:

- Subscription:** Microsoft Azure Internal Consumption
- Resource Group:** contoso-hub-rgrp (radio button selected: Create new)
- Region:** West US
- IoT Hub Name:** contoso-test-hub

Below the form, there are three buttons: 'Review + create' (disabled), 'Next: Size and scale >', and 'Automation options'.

Fill in the fields.

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics **Size and scale** Review + create

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS

* Pricing and scale tier ⓘ **S1: Standard tier** [Learn how to choose the right IoT Hub tier for your solution](#)

Number of S1 IoT Hub units ⓘ **1** This determines your IoT Hub scale capability and can be changed as your need increases.

Device-to-cloud-messages ⓘ Enabled

Message routing ⓘ Enabled

Cloud-to-device commands ⓘ Enabled

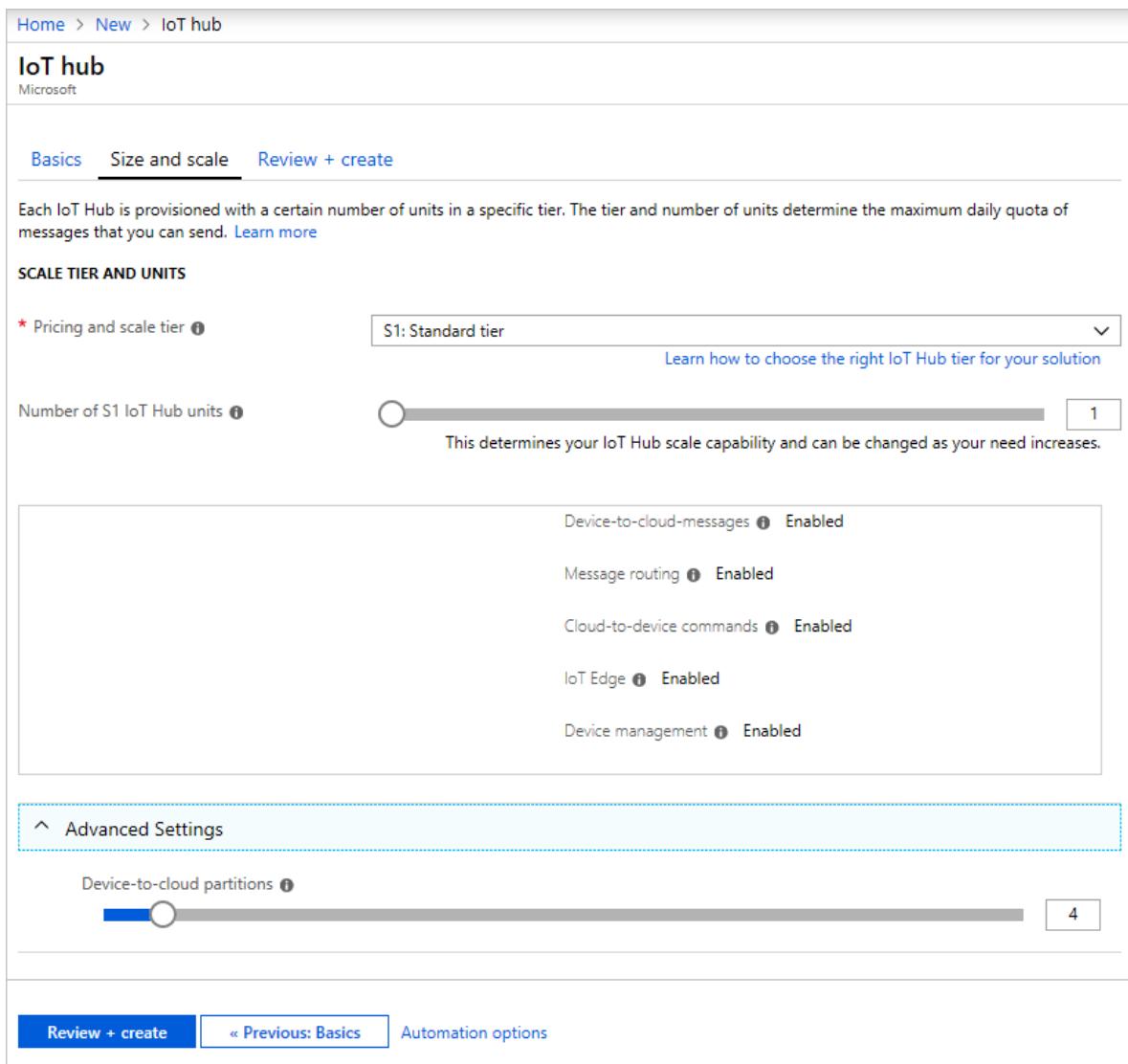
IoT Edge ⓘ Enabled

Device management ⓘ Enabled

Advanced Settings

Device-to-cloud partitions ⓘ **4**

Review + create [« Previous: Basics](#) Automation options



On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: The number of messages allowed per unit per day depends on your hub's pricing tier. For example, if you want the IoT hub to support ingress of 700,000 messages, you choose two S1 tier units.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

Advanced / Device-to-cloud partitions: This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most IoT hubs only need four partitions.

5. Click **Review + create** to review your choices. You see something similar to this screen.

The screenshot shows the 'Review + create' step of the IoT hub creation wizard. It displays basic information like subscription, resource group, region, and IoT Hub Name, followed by size and scale details including pricing tier, number of units, messages per day, and cost per month. At the bottom, there are 'Create', 'Previous: Size and scale', and 'Automation options' buttons.

6. Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Register a device

If you completed the previous [Quickstart: Send telemetry from a device to an IoT hub](#), you can skip this step.

A device must be registered with your IoT hub before it can connect. In this quickstart, you use the Azure Cloud Shell to register a simulated device.

1. Run the following commands in Azure Cloud Shell to add the IoT Hub CLI extension and to create the device identity.

YourIoTHubName: Replace this placeholder below with the name you chose for your IoT hub.

MyNodeDevice: The name of the device you're registering. Use **MyNodeDevice** as shown. If you choose a different name for your device, you need to use that name throughout this article, and update the device name in the sample applications before you run them.

```
az extension add --name azure-cli-iot-ext
az iot hub device-identity create \
--hub-name YourIoTHubName --device-id MyNodeDevice
```

2. Run the following commands in Azure Cloud Shell to get the *device connection string* for the device you just registered:

YourIoTHubName: Replace this placeholder below with the name you chose for your IoT hub.

```
az iot hub device-identity show-connection-string \
--hub-name YourIoTHubName \
--device-id MyNodeDevice \
--output table
```

Make a note of the device connection string, which looks like:

```
HostName={YourIoTHubName}.azure-devices.net;DeviceId=MyNodeDevice;SharedAccessKey={YourSharedAccessKey}
```

You use this value later in the quickstart.

3. You also need a *service connection string* to enable the back-end application to connect to your IoT hub and retrieve the messages. The following command retrieves the service connection string for your IoT hub:

YourIoTHubName: Replace this placeholder below with the name you chose for your IoT hub.

```
az iot hub show-connection-string \
--hub-name YourIoTHubName --output table
```

Make a note of the service connection string, which looks like:

```
HostName={YourIoTHubName}.azure-devices.net;SharedAccessKeyName=iothubowner;SharedAccessKey=
{YourSharedAccessKey}
```

You use this value later in the quickstart. The service connection string is different from the device connection string.

Listen for direct method calls

The simulated device application connects to a device-specific endpoint on your IoT hub, sends simulated telemetry, and listens for direct method calls from your hub. In this quickstart, the direct method call from the hub tells the device to change the interval at which it sends telemetry. The simulated device sends an acknowledgement back to your hub after it executes the direct method.

1. In a local terminal window, navigate to the root folder of the sample Node.js project. Then navigate to the **iot-hub\Quickstarts\simulated-device-2** folder.
2. Open the **SimulatedDevice.js** file in a text editor of your choice.

Replace the value of the `connectionString` variable with the device connection string you made a note of previously. Then save your changes to **SimulatedDevice.js** file.

3. In the local terminal window, run the following commands to install the required libraries and run the simulated device application:

```
npm install
node SimulatedDevice.js
```

The following screenshot shows the output as the simulated device application sends telemetry to your IoT hub:

```
C:\repos\azure-iot-samples-node-master\iot-hub\Quickstarts\simulated-device-2>node SimulatedDevice.js
Sending message: {"temperature":32.65112701016555,"humidity":73.17958581118941}
message sent
Sending message: {"temperature":22.27637758730817,"humidity":77.5875936028784}
message sent
Sending message: {"temperature":24.2611329645007,"humidity":65.76733083703007}
message sent
Sending message: {"temperature":25.31156486930323,"humidity":70.63998837831352}
message sent
Sending message: {"temperature":21.087605444857296,"humidity":61.15552541083167}
message sent
Sending message: {"temperature":30.479856770935484,"humidity":73.37969387371349}
message sent
Sending message: {"temperature":34.39632957050537,"humidity":65.0301505896214}
message sent
Sending message: {"temperature":20.84696605080118,"humidity":64.41257845076707}
message sent
Sending message: {"temperature":21.811672888503267,"humidity":60.882015530820304}
message sent
Sending message: {"temperature":29.075477188088207,"humidity":79.7075045958173}
message sent
Sending message: {"temperature":22.89560591532754,"humidity":79.67572175305389}
message sent
Sending message: {"temperature":27.627274905770303,"humidity":65.40800635846655}
message sent
Sending message: {"temperature":24.47101816737808,"humidity":66.22733914342348}
message sent
```

Call the direct method

The back-end application connects to a service-side endpoint on your IoT Hub. The application makes direct method calls to a device through your IoT hub and listens for acknowledgements. An IoT Hub back-end application typically runs in the cloud.

1. In another local terminal window, navigate to the root folder of the sample Node.js project. Then navigate to the **iot-hub\Quickstarts\back-end-application** folder.
2. Open the **BackEndApplication.js** file in a text editor of your choice.
Replace the value of the `connectionString` variable with the service connection string you made a note of previously. Then save your changes to the **BackEndApplication.js** file.
3. In the local terminal window, run the following commands to install the required libraries and run the back-end application:

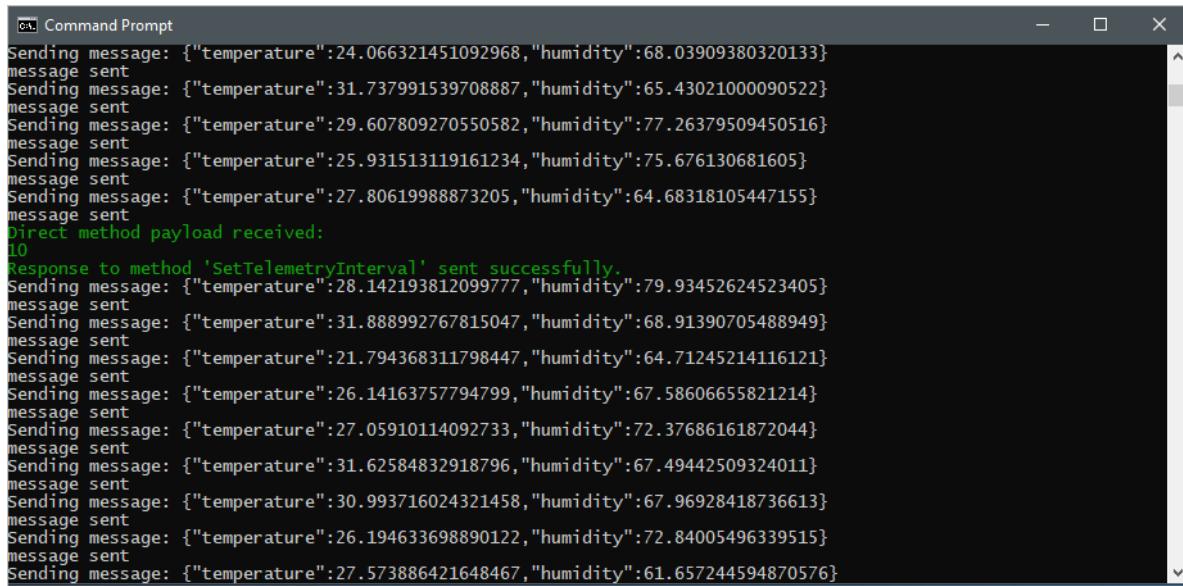
```
npm install
node BackEndApplication.js
```

The following screenshot shows the output as the application makes a direct method call to the device and receives an acknowledgement:

```
C:\repos\azure-iot-samples-node-master\iot-hub\Quickstarts\back-end-application>node BackEndApplication.js
Response from SetTelemetryInterval on MyNodeDevice:
{
  "status": 200,
  "payload": "Telemetry interval set: 10"
}
C:\repos\azure-iot-samples-node-master\iot-hub\Quickstarts\back-end-application>
C:\repos\azure-iot-samples-node-master\iot-hub\Quickstarts\back-end-application>
```

After you run the back-end application, you see a message in the console window running the simulated

device, and the rate at which it sends messages changes:



```
Command Prompt
Sending message: {"temperature":24.066321451092968,"humidity":68.03909380320133}
message sent
Sending message: {"temperature":31.737991539708887,"humidity":65.43021000090522}
message sent
Sending message: {"temperature":29.607809270550582,"humidity":77.26379509450516}
message sent
Sending message: {"temperature":25.931513119161234,"humidity":75.676130681605}
message sent
Sending message: {"temperature":27.80619988873205,"humidity":64.68318105447155}
message sent
Direct method payload received:
10
Response to method 'SetTelemetryInterval' sent successfully.
Sending message: {"temperature":28.142193812099777,"humidity":79.93452624523405}
message sent
Sending message: {"temperature":31.888992767815047,"humidity":68.91390705488949}
message sent
Sending message: {"temperature":21.794368311798447,"humidity":64.71245214116121}
message sent
Sending message: {"temperature":26.14163757794799,"humidity":67.58606655821214}
message sent
Sending message: {"temperature":27.05910114092733,"humidity":72.37686161872044}
message sent
Sending message: {"temperature":31.62584832918796,"humidity":67.49442509324011}
message sent
Sending message: {"temperature":30.993716024321458,"humidity":67.96928418736613}
message sent
Sending message: {"temperature":26.194633698890122,"humidity":72.84005496339515}
message sent
Sending message: {"temperature":27.573886421648467,"humidity":61.657244594870576}
```

Clean up resources

If you will be continuing to the next recommended article, you can keep the resources you've already created and reuse them.

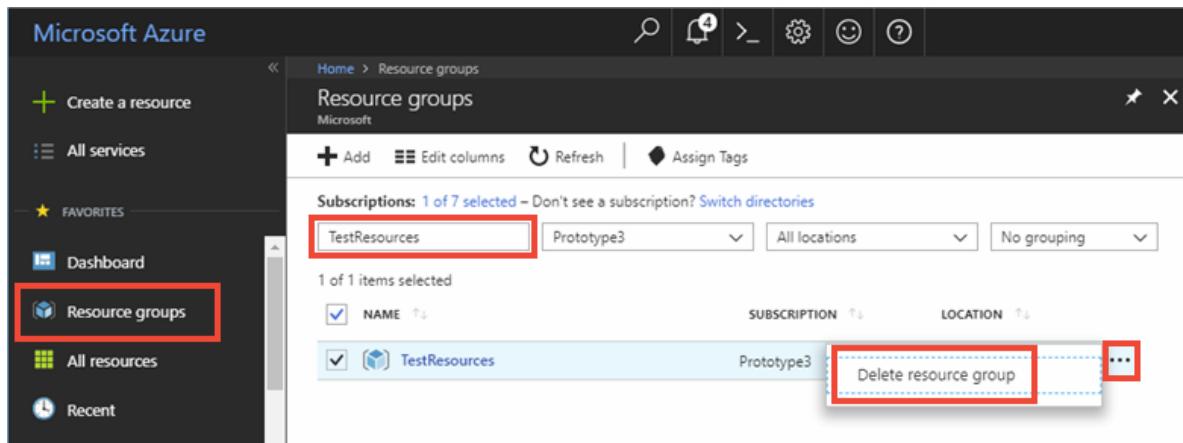
Otherwise, you can delete the Azure resources created in this article to avoid charges.

IMPORTANT

Deleting a resource group is irreversible. The resource group and all the resources contained in it are permanently deleted. Make sure that you do not accidentally delete the wrong resource group or resources. If you created the IoT Hub inside an existing resource group that contains resources you want to keep, only delete the IoT Hub resource itself instead of deleting the resource group.

To delete a resource group by name:

1. Sign in to the [Azure portal](#) and click **Resource groups**.
2. In the **Filter by name...** textbox, type the name of the resource group containing your IoT Hub.
3. To the right of your resource group in the result list, click ... then **Delete resource group**.



The screenshot shows the Microsoft Azure portal's Resource groups page. On the left, there's a sidebar with 'Create a resource', 'All services', 'FAVORITES' (which includes 'Dashboard' and 'Resource groups'), 'All resources', and 'Recent'. The 'Resource groups' item is highlighted with a red box. The main area shows a table of resource groups. A row for 'TestResources' is selected, indicated by a blue highlight. The 'NAME' column shows 'TestResources', the 'SUBSCRIPTION' column shows 'Prototype3', and the 'LOCATION' column shows 'All locations'. To the right of the table, there are three buttons: 'Delete resource group' (highlighted with a red box), '...', and '...'. Above the table, there's a search bar with 'TestResources' typed into it, also highlighted with a red box. The top navigation bar includes icons for search, refresh, and other settings.

4. You will be asked to confirm the deletion of the resource group. Type the name of your resource group again to confirm, and then click **Delete**. After a few moments, the resource group and all of its contained resources are deleted.

Next steps

In this quickstart, you called a direct method on a device from a back-end application, and responded to the direct method call in a simulated device application.

To learn how to route device-to-cloud messages to different destinations in the cloud, continue to the next tutorial.

[Tutorial: Route telemetry to different endpoints for processing](#)

Quickstart: Control a device connected to an IoT hub (.NET)

2/28/2019 • 8 minutes to read

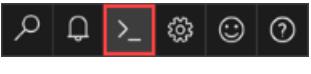
IoT Hub is an Azure service that enables you to ingest high volumes of telemetry from your IoT devices into the cloud and manage your devices from the cloud. In this quickstart, you use a *direct method* to control a simulated device connected to your IoT hub. You can use direct methods to remotely change the behavior of a device connected to your IoT hub.

The quickstart uses two pre-written .NET applications:

- A simulated device application that responds to direct methods called from a back-end application. To receive the direct method calls, this application connects to a device-specific endpoint on your IoT hub.
- A back-end application that calls the direct methods on the simulated device. To call a direct method on a device, this application connects to service-side endpoint on your IoT hub.

Open Azure Cloud Shell

Azure Cloud Shell is a free, interactive shell that you can use to run the steps in this article. Common Azure tools are preinstalled and configured in Cloud Shell for you to use with your account. Just select the **Copy** button to copy the code, paste it in Cloud Shell, and then press Enter to run it. There are a few ways to open Cloud Shell:

Select Try It in the upper-right corner of a code block.	
Open Cloud Shell in your browser.	
Select the Cloud Shell button on the menu in the upper-right corner of the Azure portal .	

If you don't have an Azure subscription, create a [free account](#) before you begin.

Prerequisites

The two sample applications you run in this quickstart are written using C#. You need the .NET Core SDK 2.1.0 or greater on your development machine.

You can download the .NET Core SDK for multiple platforms from [.NET](#).

You can verify the current version of C# on your development machine using the following command:

```
dotnet --version
```

If you haven't already done so, download the sample C# project from <https://github.com/Azure-Samples/azure-iot-samples-csharp/archive/master.zip> and extract the ZIP archive.

Create an IoT hub

If you completed the previous [Quickstart: Send telemetry from a device to an IoT hub](#), you can skip this step.

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose **+Create a resource**, then choose **Internet of Things**.
3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.

The screenshot shows the 'Basics' step of the IoT hub creation wizard. It includes a 'PROJECT DETAILS' section with fields for Subscription (set to Microsoft Azure Internal Consumption), Resource Group (contoso-hub-rgrp), Region (West US), and IoT Hub Name (contoso-test-hub). The 'Next: Size and scale >' button is highlighted with a red box.

Fill in the fields.

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Review + create

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS

* Pricing and scale tier [?](#) S1: Standard tier [▼](#) [Learn how to choose the right IoT Hub tier for your solution](#)

Number of S1 IoT Hub units [?](#) [1](#) This determines your IoT Hub scale capability and can be changed as your need increases.

Device-to-cloud-messages [?](#) Enabled

Message routing [?](#) Enabled

Cloud-to-device commands [?](#) Enabled

IoT Edge [?](#) Enabled

Device management [?](#) Enabled

Advanced Settings

Device-to-cloud partitions [?](#) [4](#)

Review + create [« Previous: Basics](#) [Automation options](#)

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: The number of messages allowed per unit per day depends on your hub's pricing tier. For example, if you want the IoT hub to support ingress of 700,000 messages, you choose two S1 tier units.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

Advanced / Device-to-cloud partitions: This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most IoT hubs only need four partitions.

5. Click **Review + create** to review your choices. You see something similar to this screen.

The screenshot shows the 'Review + create' step of the Azure portal for creating an IoT hub. The 'Review + create' button is highlighted with a red box. The page is divided into two main sections: 'BASICS' and 'SIZE AND SCALE'. In the 'BASICS' section, the following details are listed:

Subscription	Microsoft Azure Internal Consumption
Resource Group	contoso-hub-rgrp
Region	West US
IoT Hub Name	contoso-test-hub

In the 'SIZE AND SCALE' section, the following details are listed:

Pricing and scale tier	S1
Number of S1 IoT Hub units	1
Messages per day	400,000
Cost per month	25.00 USD

At the bottom, there are three buttons: 'Create' (highlighted with a red box), '[« Previous: Size and scale](#)', and '[Automation options](#)'.

6. Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Register a device

If you completed the previous [Quickstart: Send telemetry from a device to an IoT hub](#), you can skip this step.

A device must be registered with your IoT hub before it can connect. In this quickstart, you use the Azure Cloud Shell to register a simulated device.

1. Run the following commands in Azure Cloud Shell to add the IoT Hub CLI extension and to create the device identity.

YourIoTHubName: Replace this placeholder below with the name you chose for your IoT hub.

MyDotnetDevice: The name of the device you're registering. Use **MyDotnetDevice** as shown. If you choose a different name for your device, you need to use that name throughout this article, and update the device name in the sample applications before you run them.

```
az extension add --name azure-cli-iot-ext
az iot hub device-identity create \
--hub-name YourIoTHubName --device-id MyDotnetDevice
```

2. Run the following commands in Azure Cloud Shell to get the *device connection string* for the device you just registered:

YourIoTHubName: Replace this placeholder below with the name you chose for your IoT hub.

```
az iot hub device-identity show-connection-string \
--hub-name YourIoTHubName \
--device-id MyDotnetDevice
--output table
```

Make a note of the device connection string, which looks like:

```
HostName={YourIoTHubName}.azure-devices.net;DeviceId=MyNodeDevice;SharedAccessKey={YourSharedAccessKey}
```

You use this value later in the quickstart.

Retrieve the service connection string

You also need your IoT hub *service connection string* to enable the back-end application to connect to the hub and retrieve the messages. The following command retrieves the service connection string for your IoT hub:

```
az iot hub show-connection-string --hub-name YourIoTHubName --output table
```

Make a note of the service connection string, which looks like:

```
HostName={YourIoTHubName}.azure-devices.net;SharedAccessKeyName=iothubowner;SharedAccessKey={YourSharedAccessKey}
```

You use this value later in the quickstart. The service connection string is different from the device connection string.

Listen for direct method calls

The simulated device application connects to a device-specific endpoint on your IoT hub, sends simulated telemetry, and listens for direct method calls from your hub. In this quickstart, the direct method call from the hub tells the device to change the interval at which it sends telemetry. The simulated device sends an acknowledgement back to your hub after it executes the direct method.

1. In a local terminal window, navigate to the root folder of the sample C# project. Then navigate to the **iot-hub\Quickstarts\simulated-device-2** folder.
2. Open the **SimulatedDevice.cs** file in a text editor of your choice.

Replace the value of the `sConnectionString` variable with the device connection string you made a note of previously. Then save your changes to **SimulatedDevice.cs** file.

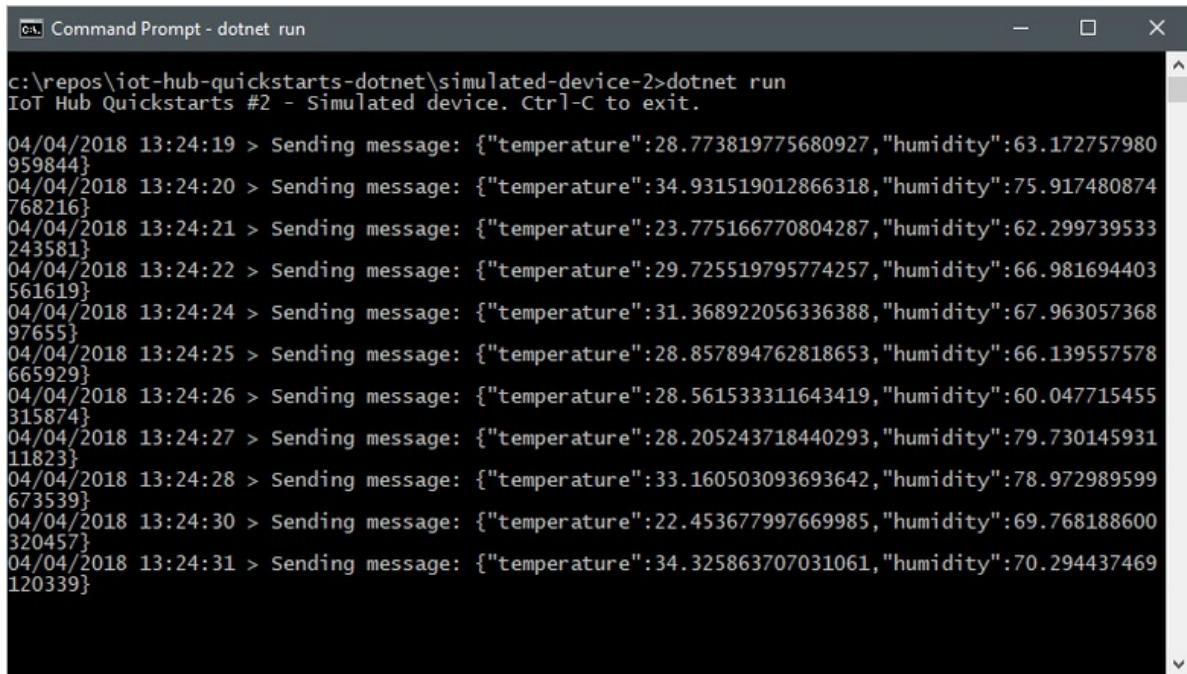
3. In the local terminal window, run the following commands to install the required packages for simulated device application:

```
dotnet restore
```

4. In the local terminal window, run the following command to build and run the simulated device application:

```
dotnet run
```

The following screenshot shows the output as the simulated device application sends telemetry to your IoT hub:



```
c:\repos\iot-hub-quickstarts-dotnet\simulated-device-2>dotnet run
IoT Hub Quickstarts #2 - Simulated device. Ctrl-C to exit.

04/04/2018 13:24:19 > Sending message: {"temperature":28.773819775680927,"humidity":63.172757980959844}
04/04/2018 13:24:20 > Sending message: {"temperature":34.931519012866318,"humidity":75.917480874768216}
04/04/2018 13:24:21 > Sending message: {"temperature":23.775166770804287,"humidity":62.299739533243581}
04/04/2018 13:24:22 > Sending message: {"temperature":29.725519795774257,"humidity":66.981694403561619}
04/04/2018 13:24:24 > Sending message: {"temperature":31.368922056336388,"humidity":67.96305736897655}
04/04/2018 13:24:25 > Sending message: {"temperature":28.857894762818653,"humidity":66.139557578665929}
04/04/2018 13:24:26 > Sending message: {"temperature":28.561533311643419,"humidity":60.047715455315874}
04/04/2018 13:24:27 > Sending message: {"temperature":28.205243718440293,"humidity":79.73014593111823}
04/04/2018 13:24:28 > Sending message: {"temperature":33.160503093693642,"humidity":78.972989599673539}
04/04/2018 13:24:30 > Sending message: {"temperature":22.453677997669985,"humidity":69.768188600320457}
04/04/2018 13:24:31 > Sending message: {"temperature":34.325863707031061,"humidity":70.294437469120339}
```

Call the direct method

The back-end application connects to a service-side endpoint on your IoT Hub. The application makes direct method calls to a device through your IoT hub and listens for acknowledgements. An IoT Hub back-end application typically runs in the cloud.

1. In another local terminal window, navigate to the root folder of the sample C# project. Then navigate to the **iot-hub\Quickstarts\back-end-application** folder.
2. Open the **BackEndApplication.cs** file in a text editor of your choice.

Replace the value of the `sConnectionString` variable with the service connection string you made a note of previously. Then save your changes to the **BackEndApplication.cs** file.

3. In the local terminal window, run the following commands to install the required libraries for the back-end application:

```
dotnet restore
```

4. In the local terminal window, run the following commands to build and run the back-end application:

```
dotnet run
```

The following screenshot shows the output as the application makes a direct method call to the device and receives an acknowledgement:

After you run the back-end application, you see a message in the console window running the simulated device, and the rate at which it sends messages changes:

Clean up resources

If you will be continuing to the next recommended article, you can keep the resources you've already created and reuse them.

Otherwise, you can delete the Azure resources created in this article to avoid charges.

IMPORTANT

Deleting a resource group is irreversible. The resource group and all the resources contained in it are permanently deleted. Make sure that you do not accidentally delete the wrong resource group or resources. If you created the IoT Hub inside an existing resource group that contains resources you want to keep, only delete the IoT Hub resource itself instead of deleting the resource group.

To delete a resource group by name:

1. Sign in to the [Azure portal](#) and click **Resource groups**.
2. In the **Filter by name...** textbox, type the name of the resource group containing your IoT Hub.
3. To the right of your resource group in the result list, click ... then **Delete resource group**.

The screenshot shows the Microsoft Azure portal interface. On the left, the sidebar has 'Resource groups' selected, indicated by a red box. The main area is titled 'Resource groups' and shows a table of resources. A subscription named 'TestResources' is selected, highlighted with a red box. In the table, there is one item named 'TestResources' under the 'NAME' column, also highlighted with a red box. To the right of this item is a context menu with options: 'Delete resource group' (highlighted with a red box) and '...'. The top navigation bar includes icons for search, refresh, and settings.

4. You will be asked to confirm the deletion of the resource group. Type the name of your resource group again to confirm, and then click **Delete**. After a few moments, the resource group and all of its contained resources are deleted.

Next steps

In this quickstart, you called a direct method on a device from a back-end application, and responded to the direct method call in a simulated device application.

To learn how to route device-to-cloud messages to different destinations in the cloud, continue to the next tutorial.

[Tutorial: Route telemetry to different endpoints for processing](#)

Quickstart: Control a device connected to an IoT hub (Java)

2/28/2019 • 8 minutes to read

IoT Hub is an Azure service that enables you to ingest high volumes of telemetry from your IoT devices into the cloud and manage your devices from the cloud. In this quickstart, you use a *direct method* to control a simulated device connected to your IoT hub. You can use direct methods to remotely change the behavior of a device connected to your IoT hub.

The quickstart uses two pre-written Java applications:

- A simulated device application that responds to direct methods called from a back-end application. To receive the direct method calls, this application connects to a device-specific endpoint on your IoT hub.
- A back-end application that calls the direct methods on the simulated device. To call a direct method on a device, this application connects to service-side endpoint on your IoT hub.

Open Azure Cloud Shell

Azure Cloud Shell is a free, interactive shell that you can use to run the steps in this article. Common Azure tools are preinstalled and configured in Cloud Shell for you to use with your account. Just select the **Copy** button to copy the code, paste it in Cloud Shell, and then press Enter to run it. There are a few ways to open Cloud Shell:

Select Try It in the upper-right corner of a code block.	
Open Cloud Shell in your browser.	
Select the Cloud Shell button on the menu in the upper-right corner of the Azure portal .	

If you don't have an Azure subscription, create a [free account](#) before you begin.

Prerequisites

The two sample applications you run in this quickstart are written using Java. You need Java SE 8 or later on your development machine.

You can download Java for multiple platforms from [Oracle](#).

You can verify the current version of Java on your development machine using the following command:

```
java -version
```

To build the samples, you need to install Maven 3. You can download Maven for multiple platforms from [Apache Maven](#).

You can verify the current version of Maven on your development machine using the following command:

```
mvn --version
```

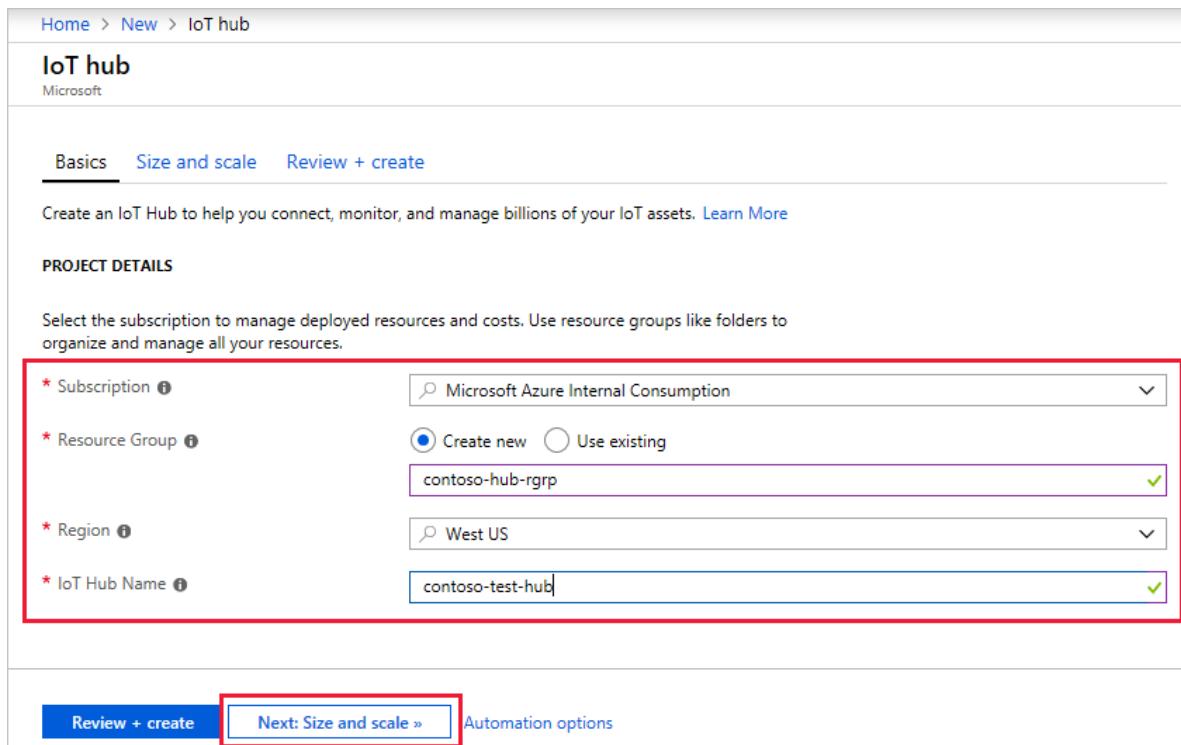
If you haven't already done so, download the sample Java project from <https://github.com/Azure-Samples/azure-iot-samples-java/archive/master.zip> and extract the ZIP archive.

Create an IoT hub

If you completed the previous [Quickstart: Send telemetry from a device to an IoT hub](#), you can skip this step.

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose **+Create a resource**, then choose **Internet of Things**.
3. Click **Iot Hub** from the list on the right. You see the first screen for creating an IoT hub.



Fill in the fields.

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

The screenshot shows the 'Size and scale' configuration page for an IoT hub. At the top, there are tabs for 'Basics', 'Size and scale' (which is selected), and 'Review + create'. A note below the tabs states: 'Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send.' A link 'Learn more' is provided. The 'SCALE TIER AND UNITS' section includes a dropdown for 'Pricing and scale tier' set to 'S1: Standard tier' with a note 'Learn how to choose the right IoT Hub tier for your solution'. Below it is a slider for 'Number of S1 IoT Hub units' set to 1, with a note: 'This determines your IoT Hub scale capability and can be changed as your need increases.' A large box lists enabled features: 'Device-to-cloud-messages' (Enabled), 'Message routing' (Enabled), 'Cloud-to-device commands' (Enabled), 'IoT Edge' (Enabled), and 'Device management' (Enabled). An 'Advanced Settings' section is collapsed, showing a slider for 'Device-to-cloud partitions' set to 4. At the bottom, there are buttons for 'Review + create', '< Previous: Basics', and 'Automation options'.

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: The number of messages allowed per unit per day depends on your hub's pricing tier. For example, if you want the IoT hub to support ingress of 700,000 messages, you choose two S1 tier units.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

Advanced / Device-to-cloud partitions: This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most IoT hubs only need four partitions.

5. Click **Review + create** to review your choices. You see something similar to this screen.

The screenshot shows the 'Review + create' step of the IoT hub creation wizard. It displays basic information like subscription, resource group, region, and IoT Hub Name, followed by size and scale details including pricing tier, number of units, messages per day, and cost. At the bottom, the 'Create' button is highlighted with a red box.

6. Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Register a device

If you completed the previous [Quickstart: Send telemetry from a device to an IoT hub](#), you can skip this step.

A device must be registered with your IoT hub before it can connect. In this quickstart, you use the Azure Cloud Shell to register a simulated device.

1. Run the following commands in Azure Cloud Shell to add the IoT Hub CLI extension and to create the device identity.

YourIoTHubName: Replace this placeholder below with the name you chose for your IoT hub.

MyJavaDevice: The name of the device you're registering. Use **MyJavaDevice** as shown. If you choose a different name for your device, you need to use that name throughout this article, and update the device name in the sample applications before you run them.

```
az extension add --name azure-cli-iot-ext
az iot hub device-identity create \
--hub-name YourIoTHubName --device-id MyJavaDevice
```

2. Run the following commands in Azure Cloud Shell to get the *device connection string* for the device you just registered:

YourIoTHubName: Replace this placeholder below with the name you choose for your IoT hub.

```
az iot hub device-identity show-connection-string \
--hub-name YourIoTHubName \
--device-id MyJavaDevice \
--output table
```

Make a note of the device connection string, which looks like:

```
HostName={YourIoTHubName}.azure-devices.net;DeviceId=MyNodeDevice;SharedAccessKey={YourSharedAccessKey}
```

You use this value later in the quickstart.

Retrieve the service connection string

You also need a *service connection string* to enable the back-end application to connect to your IoT hub and retrieve the messages. The following command retrieves the service connection string for your IoT hub:

YourIoTHubName: Replace this placeholder below with the name you chose for your IoT hub.

```
az iot hub show-connection-string --hub-name YourIoTHubName --output table
```

Make a note of the service connection string, which looks like:

```
HostName={YourIoTHubName}.azure-devices.net;SharedAccessKeyName=iothubowner;SharedAccessKey={YourSharedAccessKey}
```

You use this value later in the quickstart. The service connection string is different from the device connection string.

Listen for direct method calls

The simulated device application connects to a device-specific endpoint on your IoT hub, sends simulated telemetry, and listens for direct method calls from your hub. In this quickstart, the direct method call from the hub tells the device to change the interval at which it sends telemetry. The simulated device sends an acknowledgement back to your hub after it executes the direct method.

1. In a local terminal window, navigate to the root folder of the sample Java project. Then navigate to the **iot-hub\Quickstarts\simulated-device-2** folder.
2. Open the **src/main/java/com/microsoft/docs/iothub/samples/SimulatedDevice.java** file in a text editor of your choice.

Replace the value of the `connString` variable with the device connection string you made a note of previously. Then save your changes to **SimulatedDevice.java** file.

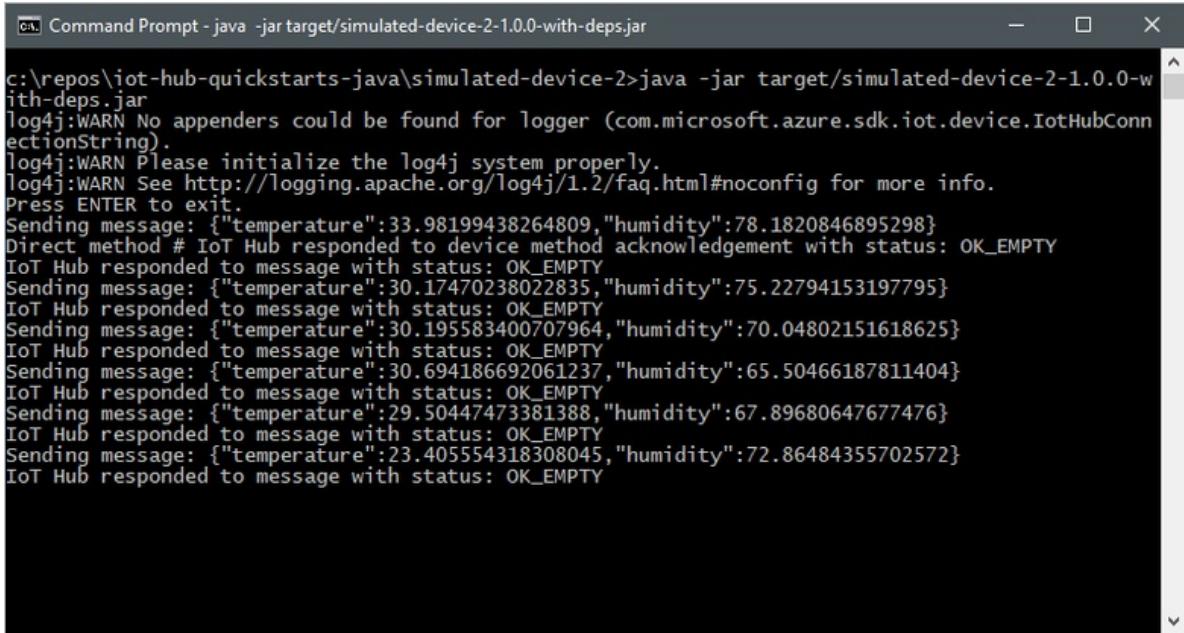
3. In the local terminal window, run the following commands to install the required libraries and build the simulated device application:

```
mvn clean package
```

4. In the local terminal window, run the following commands to run the simulated device application:

```
java -jar target/simulated-device-2-1.0.0-with-deps.jar
```

The following screenshot shows the output as the simulated device application sends telemetry to your IoT hub:



```
c:\repos\iot-hub-quickstarts-java\simulated-device-2>java -jar target/simulated-device-2-1.0.0-with-deps.jar
c:\repos\iot-hub-quickstarts-java\simulated-device-2>java -jar target/simulated-device-2-1.0.0-with-deps.jar
log4j:WARN No appenders could be found for logger (com.microsoft.azure.sdk.iot.device.IoTHubConnectionString).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
Press ENTER to exit.
Sending message: {"temperature":33.98199438264809,"humidity":78.1820846895298}
Direct method # IoT Hub responded to device method acknowledgement with status: OK_EMPTY
IoT Hub responded to message with status: OK_EMPTY
Sending message: {"temperature":30.17470238022835,"humidity":75.22794153197795}
IoT Hub responded to message with status: OK_EMPTY
Sending message: {"temperature":30.195583400707964,"humidity":70.04802151618625}
IoT Hub responded to message with status: OK_EMPTY
Sending message: {"temperature":30.694186692061237,"humidity":65.50466187811404}
IoT Hub responded to message with status: OK_EMPTY
Sending message: {"temperature":29.50447473381388,"humidity":67.89680647677476}
IoT Hub responded to message with status: OK_EMPTY
Sending message: {"temperature":23.405554318308045,"humidity":72.86484355702572}
IoT Hub responded to message with status: OK_EMPTY
```

Call the direct method

The back-end application connects to a service-side endpoint on your IoT Hub. The application makes direct method calls to a device through your IoT hub and listens for acknowledgements. An IoT Hub back-end application typically runs in the cloud.

1. In another local terminal window, navigate to the root folder of the sample Java project. Then navigate to the **iot-hub\Quickstarts\back-end-application** folder.
2. Open the **src/main/java/com/microsoft/docs/iothub/samples/BackEndApplication.java** file in a text editor of your choice.

Replace the value of the `iotHubConnectionString` variable with the service connection string you made a note of previously. Then save your changes to the **BackEndApplication.java** file.

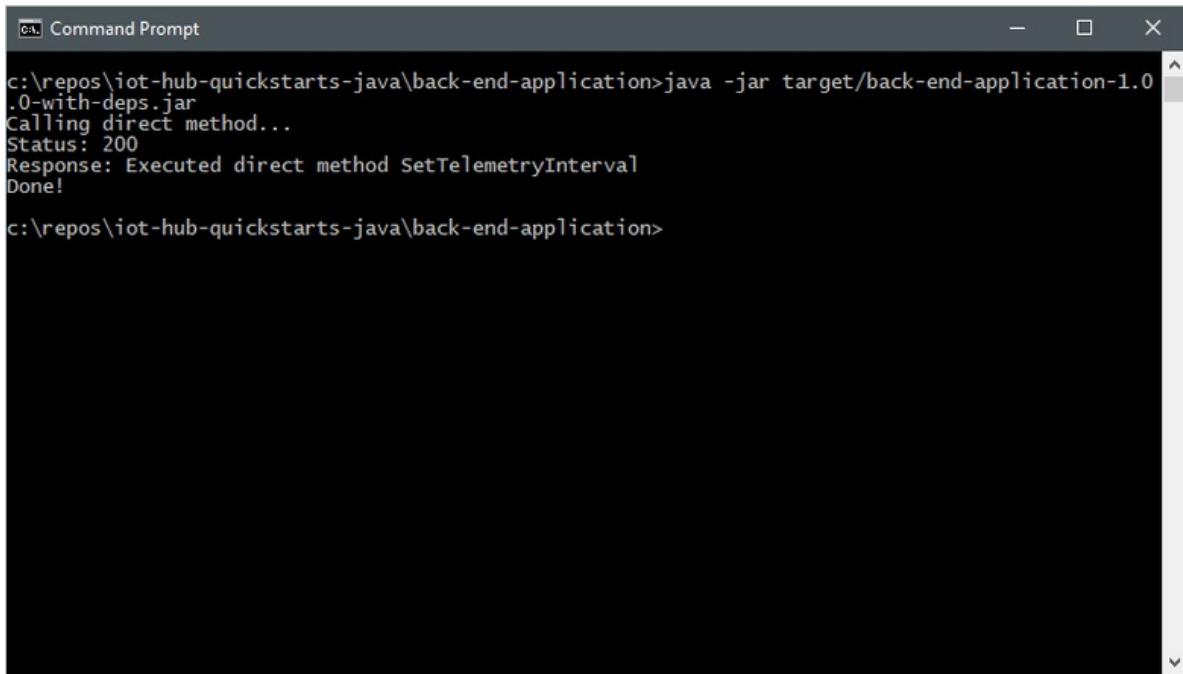
3. In the local terminal window, run the following commands to install the required libraries and build the back-end application:

```
mvn clean package
```

4. In the local terminal window, run the following commands to run the back-end application:

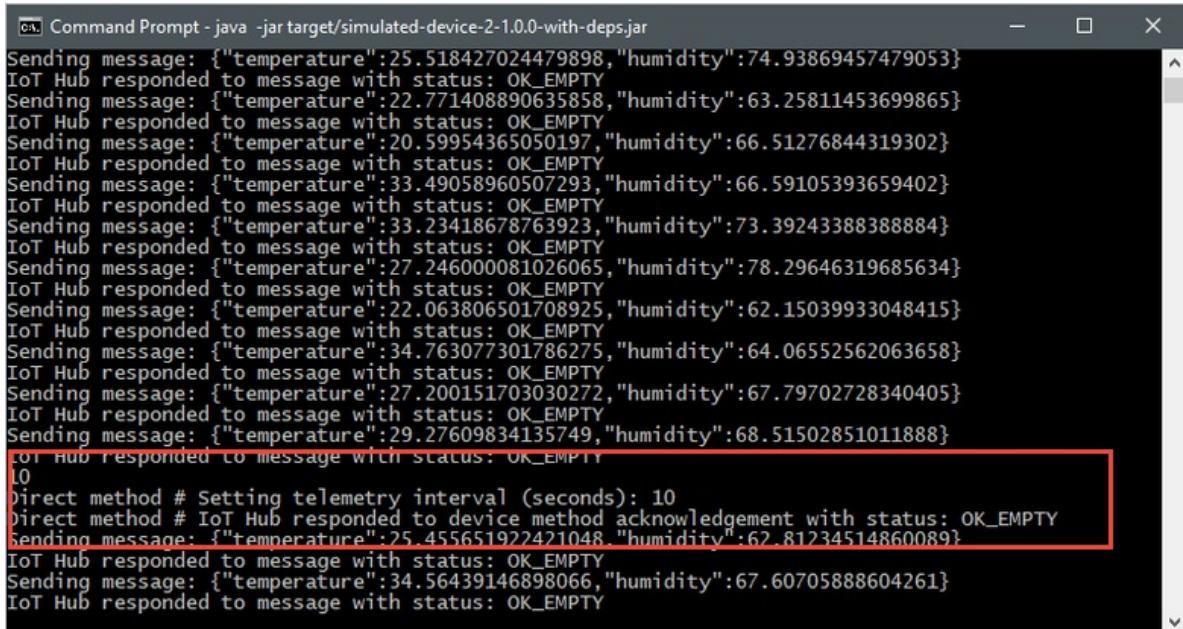
```
java -jar target/back-end-application-1.0.0-with-deps.jar
```

The following screenshot shows the output as the application makes a direct method call to the device and receives an acknowledgement:



```
c:\repos\iot-hub-quickstarts-java\back-end-application>java -jar target/back-end-application-1.0.0-with-deps.jar
Calling direct method...
Status: 200
Response: Executed direct method SetTelemetryInterval
Done!
c:\repos\iot-hub-quickstarts-java\back-end-application>
```

After you run the back-end application, you see a message in the console window running the simulated device, and the rate at which it sends messages changes:



```
c:\ Command Prompt - java -jar target/simulated-device-2-1.0.0-with-deps.jar
Sending message: {"temperature":25.518427024479898,"humidity":74.93869457479053}
IoT Hub responded to message with status: OK_EMPTY
Sending message: {"temperature":22.771408890635858,"humidity":63.25811453699865}
IoT Hub responded to message with status: OK_EMPTY
Sending message: {"temperature":20.59954365050197,"humidity":66.51276844319302}
IoT Hub responded to message with status: OK_EMPTY
Sending message: {"temperature":33.49058960507293,"humidity":66.59105393659402}
IoT Hub responded to message with status: OK_EMPTY
Sending message: {"temperature":33.23418678763923,"humidity":73.39243388388884}
IoT Hub responded to message with status: OK_EMPTY
Sending message: {"temperature":27.246000081026065,"humidity":78.29646319685634}
IoT Hub responded to message with status: OK_EMPTY
Sending message: {"temperature":22.063806501708925,"humidity":62.15039933048415}
IoT Hub responded to message with status: OK_EMPTY
Sending message: {"temperature":34.763077301786275,"humidity":64.06552562063658}
IoT Hub responded to message with status: OK_EMPTY
Sending message: {"temperature":27.200151703030272,"humidity":67.79702728340405}
IoT Hub responded to message with status: OK_EMPTY
Sending message: {"temperature":29.27609834135749,"humidity":68.51502851011888}
IoT Hub responded to message with status: OK_EMPTY
IoT Hub responded to message with status: OK_EMPTY
Direct method # Setting telemetry interval (seconds): 10
Direct method # IoT Hub responded to device method acknowledgement with status: OK_EMPTY
Sending message: {"temperature":25.455651922421048,"humidity":62.81234514860089}
IoT Hub responded to message with status: OK_EMPTY
Sending message: {"temperature":34.56439146898066,"humidity":67.60705888604261}
IoT Hub responded to message with status: OK_EMPTY
```

Clean up resources

If you will be continuing to the next recommended article, you can keep the resources you've already created and reuse them.

Otherwise, you can delete the Azure resources created in this article to avoid charges.

IMPORTANT

Deleting a resource group is irreversible. The resource group and all the resources contained in it are permanently deleted. Make sure that you do not accidentally delete the wrong resource group or resources. If you created the IoT Hub inside an existing resource group that contains resources you want to keep, only delete the IoT Hub resource itself instead of deleting the resource group.

To delete a resource group by name:

1. Sign in to the [Azure portal](#) and click **Resource groups**.
2. In the **Filter by name...** textbox, type the name of the resource group containing your IoT Hub.
3. To the right of your resource group in the result list, click ... then **Delete resource group**.

The screenshot shows the Microsoft Azure portal interface. On the left, the navigation bar includes 'Create a resource', 'All services', 'FAVORITES' (with 'Dashboard', 'Resource groups' highlighted with a red box, 'All resources', and 'Recent'), and a search bar. The main content area is titled 'Resource groups' under 'Microsoft'. It shows a table with one item selected: 'TestResources' (Subscription: Prototype3, Location: All locations). A red box highlights the 'TestResources' entry in the table. To the right of the table, there is a context menu with options: 'Delete resource group' (highlighted with a red box) and '...'. At the top of the main area, there are buttons for '+ Add', 'Edit columns', 'Refresh', and 'Assign Tags', along with filter dropdowns for 'Subscriptions', 'Locations', and 'Grouping'.

4. You will be asked to confirm the deletion of the resource group. Type the name of your resource group again to confirm, and then click **Delete**. After a few moments, the resource group and all of its contained resources are deleted.

Next steps

In this quickstart, you called a direct method on a device from a back-end application, and responded to the direct method call in a simulated device application.

To learn how to route device-to-cloud messages to different destinations in the cloud, continue to the next tutorial.

[Tutorial: Route telemetry to different endpoints for processing](#)

Quickstart: Control a device connected to an IoT hub (Python)

2/28/2019 • 8 minutes to read

IoT Hub is an Azure service that enables you to ingest high volumes of telemetry from your IoT devices into the cloud and manage your devices from the cloud. In this quickstart, you use a *direct method* to control a simulated device connected to your IoT hub. You can use direct methods to remotely change the behavior of a device connected to your IoT hub.

The quickstart uses two pre-written Python applications:

- A simulated device application that responds to direct methods called from a back-end application. To receive the direct method calls, this application connects to a device-specific endpoint on your IoT hub.
- A back-end application that calls the direct methods on the simulated device. To call a direct method on a device, this application connects to service-side endpoint on your IoT hub.

Open Azure Cloud Shell

Azure Cloud Shell is a free, interactive shell that you can use to run the steps in this article. Common Azure tools are preinstalled and configured in Cloud Shell for you to use with your account. Just select the **Copy** button to copy the code, paste it in Cloud Shell, and then press Enter to run it. There are a few ways to open Cloud Shell:

Select **Try It** in the upper-right corner of a code block.



Open Cloud Shell in your browser.



Select the **Cloud Shell** button on the menu in the upper-right corner of the [Azure portal](#).



If you don't have an Azure subscription, create a [free account](#) before you begin.

Prerequisites

The two sample applications you run in this quickstart are written using Python. You need either Python 2.7.x or 3.5.x on your development machine.

You can download Python for multiple platforms from [Python.org](#).

You can verify the current version of Python on your development machine using one of the following commands:

```
python --version
```

```
python3 --version
```

If you haven't already done so, download the sample Python project from <https://github.com/Azure-Samples/azure-iot-samples-python/archive/master.zip> and extract the ZIP archive.

Create an IoT hub

If you completed the previous [Quickstart: Send telemetry from a device to an IoT hub](#), you can skip this step.

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose **+Create a resource**, then choose **Internet of Things**.
3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.

The screenshot shows the 'Basics' step of the IoT hub creation wizard. It includes a 'PROJECT DETAILS' section with dropdowns for Subscription (Microsoft Azure Internal Consumption), Resource Group (contoso-hub-rgrp), Region (West US), and IoT Hub Name (contoso-test-hub). The 'Next: Size and scale >' button is highlighted with a red box.

Fill in the fields.

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics **Size and scale** Review + create

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS

* Pricing and scale tier [?](#) S1: Standard tier [?](#) [Learn how to choose the right IoT Hub tier for your solution](#)

Number of S1 IoT Hub units [?](#) [1](#) This determines your IoT Hub scale capability and can be changed as your need increases.

Device-to-cloud-messages [?](#) Enabled

Message routing [?](#) Enabled

Cloud-to-device commands [?](#) Enabled

IoT Edge [?](#) Enabled

Device management [?](#) Enabled

Advanced Settings

Device-to-cloud partitions [?](#) [4](#)

[Review + create](#) [« Previous: Basics](#) [Automation options](#)

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: The number of messages allowed per unit per day depends on your hub's pricing tier. For example, if you want the IoT hub to support ingress of 700,000 messages, you choose two S1 tier units.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

Advanced / Device-to-cloud partitions: This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most IoT hubs only need four partitions.

5. Click **Review + create** to review your choices. You see something similar to this screen.

Subscription Microsoft Azure Internal Consumption
Resource Group contoso-hub-rgrp
Region West US
IoT Hub Name contoso-test-hub

Pricing and scale tier \$1
Number of S1 IoT Hub units 1
Messages per day 400,000
Cost per month 25.00 USD

Create << Previous: Size and scale Automation options

6. Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Register a device

If you completed the previous [Quickstart: Send telemetry from a device to an IoT hub](#), you can skip this step.

A device must be registered with your IoT hub before it can connect. In this quickstart, you use the Azure Cloud Shell to register a simulated device.

1. Run the following commands in Azure Cloud Shell to add the IoT Hub CLI extension and to create the device identity.

YourIoTHubName : Replace this placeholder below with the name you chose for your IoT hub.

MyPythonDevice : This is the name given for the registered device. Use MyPythonDevice as shown. If you choose a different name for your device, you will also need to use that name throughout this article, and update the device name in the sample applications before you run them.

```
az extension add --name azure-cli-iot-ext
az iot hub device-identity create --hub-name YourIoTHubName --device-id MyPythonDevice
```

2. Run the following commands in Azure Cloud Shell to get the *device connection string* for the device you just registered:

YourIoTHubName : Replace this placeholder below with the name you chose for your IoT hub.

```
az iot hub device-identity show-connection-string --hub-name YourIoTHubName --device-id MyPythonDevice
--output table
```

Make a note of the device connection string, which looks like:

```
HostName={YourIoTHubName}.azure-devices.net;DeviceId=MyNodeDevice;SharedAccessKey={YourSharedAccessKey}
```

You use this value later in the quickstart.

3. You also need a *service connection string* to enable the back-end application to connect to your IoT hub and retrieve the messages. The following command retrieves the service connection string for your IoT hub:

YourIoTHubName : Replace this placeholder below with the name you choose for your IoT hub.

```
az iot hub show-connection-string \
--hub-name YourIoTHubName \
--output table
```

Make a note of the service connection string, which looks like:

```
HostName={YourIoTHubName}.azure-devices.net;SharedAccessKeyName=iothubowner;SharedAccessKey=
{YourSharedAccessKey}
```

You use this value later in the quickstart. The service connection string is different from the device connection string.

Listen for direct method calls

The simulated device application connects to a device-specific endpoint on your IoT hub, sends simulated telemetry, and listens for direct method calls from your hub. In this quickstart, the direct method call from the hub tells the device to change the interval at which it sends telemetry. The simulated device sends an acknowledgement back to your hub after it executes the direct method.

1. In a local terminal window, navigate to the root folder of the sample Python project. Then navigate to the **iot-hub\Quickstarts\simulated-device-2** folder.
2. Open the **SimulatedDevice.py** file in a text editor of your choice.

Replace the value of the `CONNECTION_STRING` variable with the device connection string you made a note of previously. Then save your changes to **SimulatedDevice.py** file.

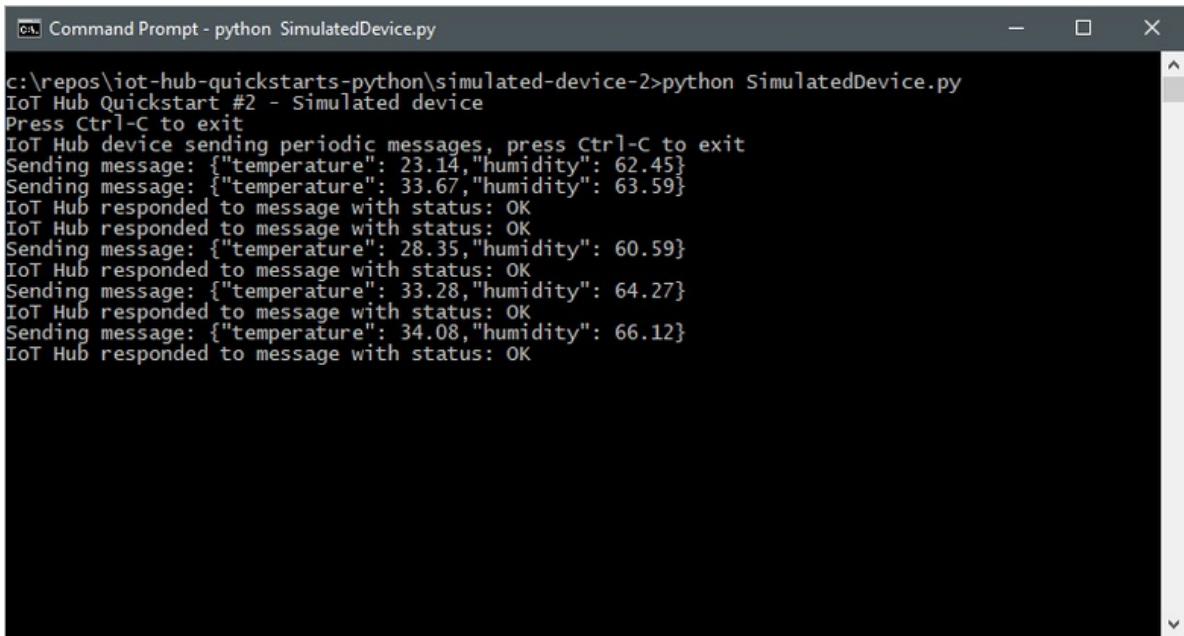
3. In the local terminal window, run the following commands to install the required libraries for the simulated device application:

```
pip install azure-iothub-device-client
```

4. In the local terminal window, run the following commands to run the simulated device application:

```
python SimulatedDevice.py
```

The following screenshot shows the output as the simulated device application sends telemetry to your IoT hub:



```
c:\repos\iot-hub-quickstarts-python\simulated-device-2>python SimulatedDevice.py
IoT Hub Quickstart #2 - Simulated device
Press Ctrl-C to exit
IoT Hub device sending periodic messages, press Ctrl-C to exit
Sending message: {"temperature": 23.14,"humidity": 62.45}
Sending message: {"temperature": 33.67,"humidity": 63.59}
IoT Hub responded to message with status: OK
IoT Hub responded to message with status: OK
Sending message: {"temperature": 28.35,"humidity": 60.59}
IoT Hub responded to message with status: OK
Sending message: {"temperature": 33.28,"humidity": 64.27}
IoT Hub responded to message with status: OK
Sending message: {"temperature": 34.08,"humidity": 66.12}
IoT Hub responded to message with status: OK
```

Call the direct method

The back-end application connects to a service-side endpoint on your IoT Hub. The application makes direct method calls to a device through your IoT hub and listens for acknowledgements. An IoT Hub back-end application typically runs in the cloud.

1. In another local terminal window, navigate to the root folder of the sample Python project. Then navigate to the **iot-hub\Quickstarts\back-end-application** folder.
2. Open the **BackEndApplication.py** file in a text editor of your choice.

Replace the value of the `CONNECTION_STRING` variable with the service connection string you made a note of previously. Then save your changes to the **BackEndApplication.py** file.

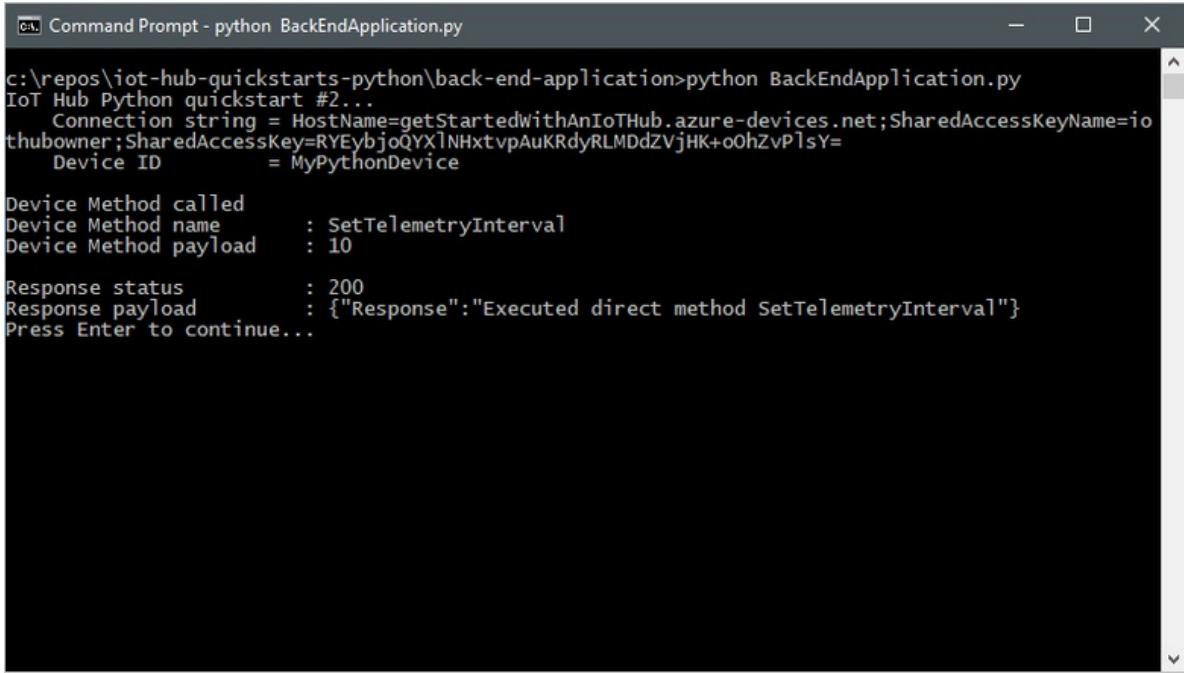
3. In the local terminal window, run the following commands to install the required libraries for the simulated device application:

```
pip install azure-iothub-service-client future
```

4. In the local terminal window, run the following commands to run the back-end application:

```
python BackEndApplication.py
```

The following screenshot shows the output as the application makes a direct method call to the device and receives an acknowledgement:

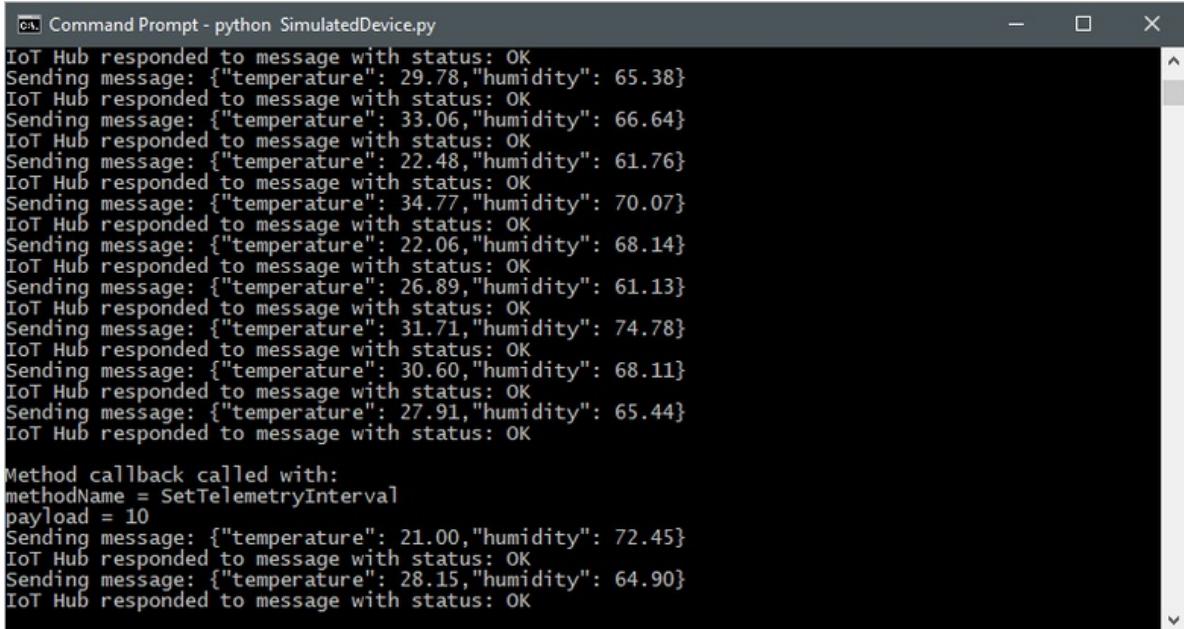


```
c:\repos\iot-hub-quickstarts-python\back-end-application>python BackEndApplication.py
IoT Hub Python quickstart #2...
Connection string = HostName=getStartedWithAnIoTHub.azure-devices.net;SharedAccessKeyName=iothubowner;SharedAccessKey=RYEybjQYX1NHxtvpAuKRdyRLMDdZVjHK+oOhZvPlsY=
Device ID          = MyPythonDevice

Device Method called
Device Method name   : SetTelemetryInterval
Device Method payload : 10

Response status      : 200
Response payload     : {"Response":"Executed direct method SetTelemetryInterval"}
Press Enter to continue...
```

After you run the back-end application, you see a message in the console window running the simulated device, and the rate at which it sends messages changes:



```
IoT Hub responded to message with status: OK
Sending message: {"temperature": 29.78, "humidity": 65.38}
IoT Hub responded to message with status: OK
Sending message: {"temperature": 33.06, "humidity": 66.64}
IoT Hub responded to message with status: OK
Sending message: {"temperature": 22.48, "humidity": 61.76}
IoT Hub responded to message with status: OK
Sending message: {"temperature": 34.77, "humidity": 70.07}
IoT Hub responded to message with status: OK
Sending message: {"temperature": 22.06, "humidity": 68.14}
IoT Hub responded to message with status: OK
Sending message: {"temperature": 26.89, "humidity": 61.13}
IoT Hub responded to message with status: OK
Sending message: {"temperature": 31.71, "humidity": 74.78}
IoT Hub responded to message with status: OK
Sending message: {"temperature": 30.60, "humidity": 68.11}
IoT Hub responded to message with status: OK
Sending message: {"temperature": 27.91, "humidity": 65.44}
IoT Hub responded to message with status: OK

Method callback called with:
methodName = SetTelemetryInterval
payload = 10
Sending message: {"temperature": 21.00, "humidity": 72.45}
IoT Hub responded to message with status: OK
Sending message: {"temperature": 28.15, "humidity": 64.90}
IoT Hub responded to message with status: OK
```

Clean up resources

If you will be continuing to the next recommended article, you can keep the resources you've already created and reuse them.

Otherwise, you can delete the Azure resources created in this article to avoid charges.

IMPORTANT

Deleting a resource group is irreversible. The resource group and all the resources contained in it are permanently deleted. Make sure that you do not accidentally delete the wrong resource group or resources. If you created the IoT Hub inside an existing resource group that contains resources you want to keep, only delete the IoT Hub resource itself instead of deleting the resource group.

To delete a resource group by name:

1. Sign in to the [Azure portal](#) and click **Resource groups**.
2. In the **Filter by name...** textbox, type the name of the resource group containing your IoT Hub.
3. To the right of your resource group in the result list, click ... then **Delete resource group**.

The screenshot shows the Microsoft Azure portal interface. On the left, the navigation bar includes 'Create a resource', 'All services', 'FAVORITES' (with 'Dashboard', 'Resource groups' highlighted with a red box, 'All resources', and 'Recent'), and a search bar. The main content area is titled 'Resource groups' under 'Subscriptions: 1 of 7 selected'. A table lists one item: 'TestResources' (Subscription: Prototype3, Location: All locations). To the right of the table, there is a context menu with options: 'Delete resource group' (highlighted with a red box) and '...'. The top right of the page has standard Azure navigation icons.

4. You will be asked to confirm the deletion of the resource group. Type the name of your resource group again to confirm, and then click **Delete**. After a few moments, the resource group and all of its contained resources are deleted.

Next steps

In this quickstart, you've called a direct method on a device from a back-end application, and responded to the direct method call in a simulated device application.

To learn how to route device-to-cloud messages to different destinations in the cloud, continue to the next tutorial.

[Tutorial: Route telemetry to different endpoints for processing](#)

Quickstart: Control a device connected to an IoT hub (Android)

3/10/2019 • 10 minutes to read

IoT Hub is an Azure service that enables you to ingest high volumes of telemetry from your IoT devices into the cloud and manage your devices from the cloud. In this quickstart, you use a *direct method* to control a simulated device connected to your IoT hub. You can use direct methods to remotely change the behavior of a device connected to your IoT hub.

The quickstart uses two pre-written Java applications:

- A simulated device application that responds to direct methods called from a back-end service application. To receive the direct method calls, this application connects to a device-specific endpoint on your IoT hub.
- A service application that calls the direct method on the Android device. To call a direct method on a device, this application connects to service-side endpoint on your IoT hub.

Open Azure Cloud Shell

Azure Cloud Shell is a free, interactive shell that you can use to run the steps in this article. Common Azure tools are preinstalled and configured in Cloud Shell for you to use with your account. Just select the **Copy** button to copy the code, paste it in Cloud Shell, and then press Enter to run it. There are a few ways to open Cloud Shell:

Select Try It in the upper-right corner of a code block.	
Open Cloud Shell in your browser.	
Select the Cloud Shell button on the menu in the upper-right corner of the Azure portal .	

If you don't have an Azure subscription, create a [free account](#) before you begin.

Prerequisites

- Android studio from <https://developer.android.com/studio/>. For more information regarding Android Studio installation, see [android-installation](#).
- Android SDK 27 is used by the sample in this article.
- Two sample applications are required by this quickstart: The [Device SDK sample Android application](#) and the [Service SDK sample Android application](#). Both of these samples are part of the `azure-iot-samples-java` repository on GitHub. Download or clone the [azure-iot-samples-java](#) repository.

Create an IoT hub

If you completed the previous [Quickstart: Send telemetry from a device to an IoT hub](#), you can skip this step and use the IoT hub you have already created.

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose +**Create a resource**, then choose **Internet of Things**.
3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.

The screenshot shows the 'Basics' step of the IoT Hub creation wizard. The 'PROJECT DETAILS' section is highlighted with a red box. Inside, the 'Subscription' dropdown shows 'Microsoft Azure Internal Consumption'. The 'Resource Group' section has 'Create new' selected and 'contoso-hub-rgrp' entered. The 'Region' dropdown shows 'West US'. The 'IoT Hub Name' field contains 'contoso-test-hub'. Below the form, the 'Review + create' and 'Next: Size and scale »' buttons are visible, with 'Next: Size and scale »' also highlighted by a red box.

Fill in the fields.

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics **Size and scale** Review + create

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS

* Pricing and scale tier [?](#) S1: Standard tier [?](#) [Learn how to choose the right IoT Hub tier for your solution](#)

Number of S1 IoT Hub units [?](#) [1](#) This determines your IoT Hub scale capability and can be changed as your need increases.

Device-to-cloud-messages [?](#) Enabled

Message routing [?](#) Enabled

Cloud-to-device commands [?](#) Enabled

IoT Edge [?](#) Enabled

Device management [?](#) Enabled

Advanced Settings

Device-to-cloud partitions [?](#) [4](#)

[Review + create](#) [« Previous: Basics](#) [Automation options](#)

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: The number of messages allowed per unit per day depends on your hub's pricing tier. For example, if you want the IoT hub to support ingress of 700,000 messages, you choose two S1 tier units.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

Advanced / Device-to-cloud partitions: This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most IoT hubs only need four partitions.

5. Click **Review + create** to review your choices. You see something similar to this screen.

The screenshot shows the 'Review + create' step of the IoT hub creation wizard. It displays basic information like subscription, resource group, region, and IoT Hub Name, as well as size and scale details including pricing tier, number of units, messages per day, and cost per month. At the bottom, there are buttons for 'Create' (highlighted), 'Previous: Size and scale', and 'Automation options'.

6. Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Register a device

If you completed the previous [Quickstart: Send telemetry from a device to an IoT hub](#), you can skip this step and use the same device registered in the previous quickstart.

A device must be registered with your IoT hub before it can connect. In this quickstart, you use the Azure Cloud Shell to register a simulated device.

1. Run the following commands in Azure Cloud Shell to add the IoT Hub CLI extension and to create the device identity.

YourIoTHubName: Replace this placeholder below with the name you chose for your IoT hub.

MyAndroidDevice: This value is the name given for the registered device. Use MyAndroidDevice as shown. If you choose a different name for your device, you may also need to use that name throughout this article, and update the device name in the sample applications before you run them.

```
az extension add --name azure-cli-iot-ext
az iot hub device-identity create \
--hub-name YourIoTHubName --device-id MyAndroidDevice
```

2. Run the following commands in Azure Cloud Shell to get the *device connection string* for the device you just registered:

YourIoTHubName: Replace this placeholder below with the name you choose for your IoT hub.

```
az iot hub device-identity show-connection-string \
--hub-name YourIoTHubName \
--device-id MyAndroidDevice \
--output table
```

Make a note of the device connection string, which looks like:

```
HostName={YourIoTHubName}.azure-devices.net;DeviceId=MyAndroidDevice;SharedAccessKey={YourSharedAccessKey}
```

You use this value later in the quickstart.

Retrieve the service connection string

You also need a *service connection string* to enable the back-end service applications to connect to your IoT hub in order to execute methods and retrieve messages. The following command retrieves the service connection string for your IoT hub:

YourIoTHubName: Replace this placeholder below with the name you chose for your IoT hub.

```
az iot hub show-connection-string --hub-name YourIoTHubName --output table
```

Make a note of the service connection string, which looks like:

```
HostName={YourIoTHubName}.azure-devices.net;SharedAccessKeyName=iothubowner;SharedAccessKey={YourSharedAccessKey}
```

You use this value later in the quickstart. The service connection string is different from the device connection string.

Listen for direct method calls

The device SDK sample application can be run on a physical Android device or an Android emulator. The sample connects to a device-specific endpoint on your IoT hub, sends simulated telemetry, and listens for direct method calls from your hub. In this quickstart, the direct method call from the hub tells the device to change the interval at which it sends telemetry. The simulated device sends an acknowledgement back to your hub after it executes the direct method.

1. Open the GitHub sample Android project in Android Studio. The project is located in the following directory of your cloned or downloaded copy of [azure-iot-sample-java](#) repository.

```
\azure-iot-samples-java\iot-hub\Samples\device\AndroidSample
```

2. In Android Studio, open *gradle.properties* for the sample project and replace the **Device_Connection_String** placeholder with your device connection string you noted earlier.

```
DeviceConnectionString=HostName={YourIoTHubName}.azure-devices.net;DeviceId=MyAndroidDevice;SharedAccessKey={YourSharedAccessKey}
```

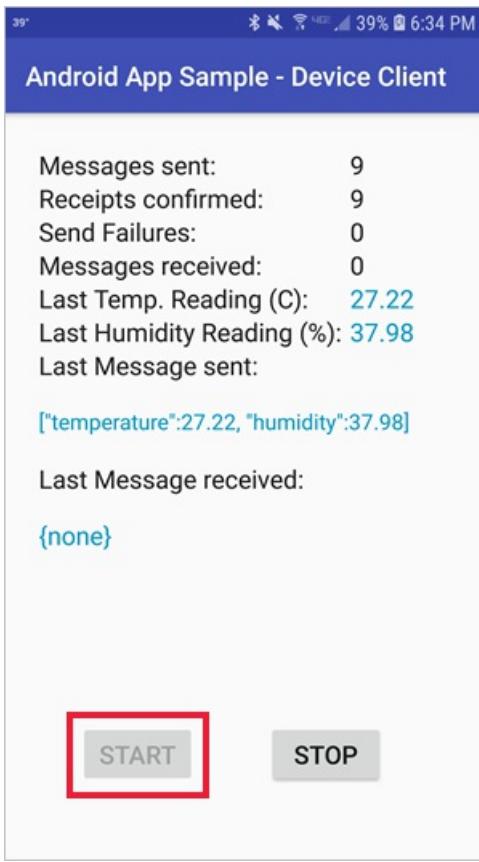
3. In Android Studio, click **File > Sync Project with Gradle Files**. Verify the build completes.

NOTE

If the project sync fails, it may be for one of the following reasons:

- The versions of the Android Gradle plugin and Gradle referenced in the project are out of date for your version of Android Studio. Follow [these instructions](#) to reference and install the correct versions of the plugin and Gradle for your installation.
- The license agreement for the Android SDK has not been signed. Follow the instructions in the Build output to sign the license agreement and download the SDK.

- Once the build has completed, click **Run > Run 'app'**. Configure the app to run on a physical Android device or an Android emulator. For more information on running an Android app on a physical device or emulator, see [Run your app](#).
- Once the app loads, click the **Start** button to start sending telemetry to your IoT Hub:



This app needs to be left running on a physical device or emulator while you execute the service SDK sample to update the telemetry interval during run-time.

Read the telemetry from your hub

In this section, you will use the Azure Cloud Shell with the [IoT extension](#) to monitor the device messages that are sent by the Android device.

- Using the Azure Cloud Shell, run the following command to connect and read messages from your IoT hub:

YourIoTHubName: Replace this placeholder below with the name you choose for your IoT hub.

```
az iot hub monitor-events --hub-name YourIoTHubName --output table
```

The following screenshot shows the output as the IoT hub receives telemetry sent by the Android device:

```
UserName@Azure:-$ az iot hub monitor-events --hub-name JavaTesting --output table
Starting event monitor, use ctrl-c to stop...
event:
  origin: MyAndroidDevice
  payload: '{"temperature":20.16, "humidity":35.21}'

event:
  origin: MyAndroidDevice
  payload: '{"temperature":23.92, "humidity":40.28}'

event:
  origin: MyAndroidDevice
  payload: '{"temperature":21.90, "humidity":35.45}'

event:
  origin: MyAndroidDevice
  payload: '{"temperature":22.87, "humidity":39.04}'

event:
  origin: MyAndroidDevice
  payload: '{"temperature":25.31, "humidity":43.22}'
```

By default the telemetry app is sending telemetry from the Android device every 5 seconds. In the next section, you will use a direct method call to update the telemetry interval for the Android IoT device.

Call the direct method

The service application connects to a service-side endpoint on your IoT Hub. The application makes direct method calls to a device through your IoT hub and listens for acknowledgements.

Run this app on a separate physical Android device or Android emulator.

An IoT Hub back-end service application typically runs in the cloud where it is easier to mitigate the risks associated with the sensitive connection string that controls all devices on an IoT Hub. In this example, we are running it as an Android app for demonstration purposes only. The other language versions of this quickstart provide other examples that align more closely with a back-end service application.

1. Open the GitHub service sample Android project in Android Studio. The project is located in the following directory of your cloned or downloaded copy of [azure-iot-sample-java](#) repository.

```
\azure-iot-samples-java\iot-hub\Samples\service\AndroidSample
```

2. In Android Studio, open *gradle.properties* for the sample project and update the value for **ConnectionString** and **DeviceId** properties with your service connection string you noted earlier and the Android device ID you registered.

```
ConnectionString=HostName={YourIoTHubName}.azure-
devices.net;SharedAccessKeyName=iothubowner;SharedAccessKey={YourSharedAccessKey}
DeviceId=MyAndroidDevice
```

3. In Android Studio, click **File > Sync Project with Gradle Files**. Verify the build completes.

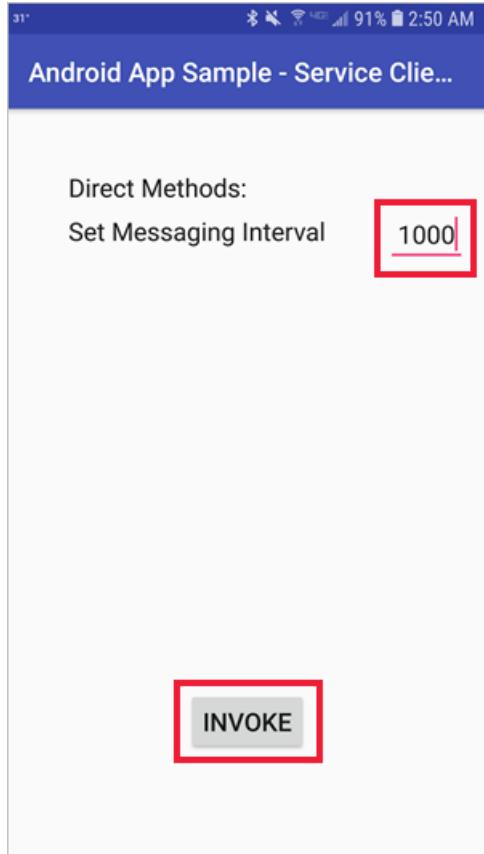
NOTE

If the project sync fails, it may be for one of the following reasons:

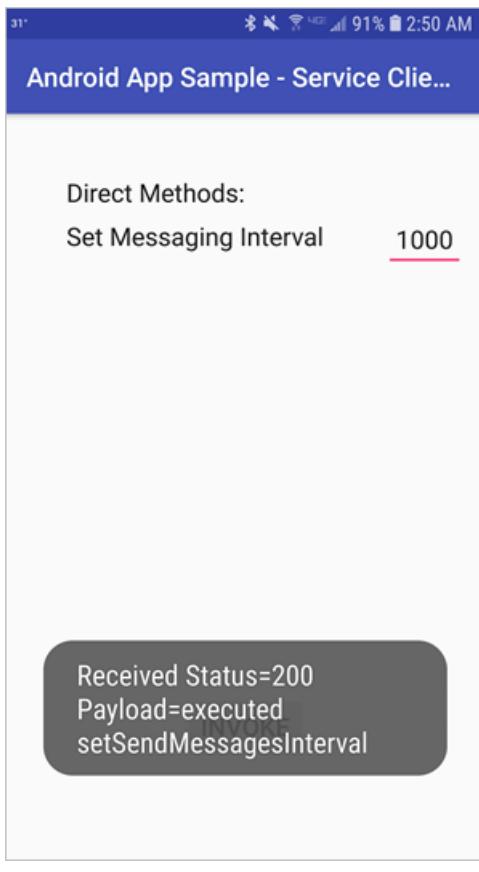
- The versions of the Android Gradle plugin and Gradle referenced in the project are out of date for your version of Android Studio. Follow [these instructions](#) to reference and install the correct versions of the plugin and Gradle for your installation.
- The license agreement for the Android SDK has not been signed. Follow the instructions in the Build output to sign the license agreement and download the SDK.

- Once the build has completed, click **Run > Run 'app'**. Configure the app to run on a separate physical Android device or an Android emulator. For more information on running an Android app on a physical device or emulator, see [Run your app](#).
- Once the app loads, update the **Set Messaging Interval** value to **1000** and click **Invoke**.

The telemetry messaging interval is in milliseconds. The default telemetry interval of the device sample is set for 5 seconds. This change will update the Android IoT device so that telemetry is sent every second.



- The app will receive an acknowledgement indicating whether the method executed successfully or not.



Clean up resources

If you will be continuing to the next recommended article, you can keep the resources you've already created and reuse them.

Otherwise, you can delete the Azure resources created in this article to avoid charges.

IMPORTANT

Deleting a resource group is irreversible. The resource group and all the resources contained in it are permanently deleted. Make sure that you do not accidentally delete the wrong resource group or resources. If you created the IoT Hub inside an existing resource group that contains resources you want to keep, only delete the IoT Hub resource itself instead of deleting the resource group.

To delete a resource group by name:

1. Sign in to the [Azure portal](#) and click **Resource groups**.
2. In the **Filter by name...** textbox, type the name of the resource group containing your IoT Hub.
3. To the right of your resource group in the result list, click ... then **Delete resource group**.

The screenshot shows the Microsoft Azure portal interface. On the left, the navigation bar includes 'Create a resource', 'All services', 'FAVORITES' (with 'Dashboard' selected), 'Resource groups' (which is highlighted with a red box), 'All resources', and 'Recent'. The main content area is titled 'Resource groups' and shows a table with one item. The table has columns: NAME, SUBSCRIPTION, and LOCATION. A single row is selected, showing 'TestResources' under NAME, 'Prototype3' under SUBSCRIPTION, and 'East US' under LOCATION. To the right of the row, there is a context menu with options: 'Delete resource group' (highlighted with a red box) and '...'. At the top of the page, there are search, filter, and settings icons.

4. You will be asked to confirm the deletion of the resource group. Type the name of your resource group again to confirm, and then click **Delete**. After a few moments, the resource group and all of its contained resources are deleted.

Next steps

In this quickstart, you called a direct method on a device from a back-end application, and responded to the direct method call in a simulated device application.

To learn how to route device-to-cloud messages to different destinations in the cloud, continue to the next tutorial.

[Tutorial: Route telemetry to different endpoints for processing](#)

Quickstart: Communicate to device applications in C# via IoT Hub device streams (preview)

3/5/2019 • 6 minutes to read

IoT Hub device streams allow service and device applications to communicate in a secure and firewall-friendly manner. This quickstart involves two C# programs that leverage device streams to send data back and forth (echo).

Open Azure Cloud Shell

Azure Cloud Shell is a free, interactive shell that you can use to run the steps in this article. Common Azure tools are preinstalled and configured in Cloud Shell for you to use with your account. Just select the **Copy** button to copy the code, paste it in Cloud Shell, and then press Enter to run it. There are a few ways to open Cloud Shell:

Select Try It in the upper-right corner of a code block.	
Open Cloud Shell in your browser.	
Select the Cloud Shell button on the menu in the upper-right corner of the Azure portal .	

If you don't have an Azure subscription, create a [free account](#) before you begin.

Prerequisites

The two sample applications you run in this quickstart are written using C#. You need the .NET Core SDK 2.1.0 or greater on your development machine.

You can download the .NET Core SDK for multiple platforms from [.NET](#).

You can verify the current version of C# on your development machine using the following command:

```
dotnet --version
```

Download the sample C# project from <https://github.com/Azure-Samples/azure-iot-samples-csharp/archive/master.zip> and extract the ZIP archive. You will need it on both device and service side.

Create an IoT hub

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose +**Create a resource**, then choose **Internet of Things**.
3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.

IoT hub

Microsoft

Basics Size and scale Review + create

Create an IoT Hub to help you connect, monitor, and manage billions of your IoT assets. [Learn More](#)

PROJECT DETAILS

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

* Subscription <small>i</small>	<input type="text"/>	<input type="button" value="▼"/>
* Resource Group <small>i</small>	(New) IoTHubDeviceStreamsRG Create new	<input type="button" value="▼"/>
* Region <small>i</small>	Central US EUAP	<input type="button" value="▼"/>
* IoT Hub Name <small>i</small>	IoTHubDeviceStreams	<input checked="" type="checkbox"/>

Fill in the fields:

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Ensure you select a supported region (e.g., Central US or Central US EUAP).

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

IoT hub

Microsoft

[Basics](#) [Size and scale](#) [Review + create](#)

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS* Pricing and scale tier [?](#)

F1: Free tier

[Learn how to choose the right IoT Hub tier for your solution](#)Number of F1 IoT Hub units [?](#)

This determines your IoT Hub scale capability and can be changed as your need increases.

Pricing and scale tier ? F1	Device-to-cloud-messages ? Enabled
Messages per day ? 8,000	Message routing ? Enabled
Cost per month	Cloud-to-device commands ? Enabled
	IoT Edge ? Enabled
	Device management ? Enabled

[Advanced Settings](#)

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: Ensure you select one of the standard (S1, S2, S3) or the Free (F1) tier. This choice can also be guided by the size of your fleet and the non-streaming workloads you expect in your hub (e.g., telemetry messages). For example, the free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: This choice depends on non-streaming workload you expect in your hub - you can select 1 for now.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

1. Click **Review + create** to review your choices. You see something similar to this screen.

IoT hub

Microsoft

Basics Size and scale Review + create

BASICS

Subscription ?	IoTHubDeviceStreamsRG
Resource Group ?	Central US EUAP
Region ?	IoTHubDeviceStreams

SIZE AND SCALE

Pricing and scale tier ?	F1
Number of F1 IoT Hub units ?	1
Messages per day ?	8,000
Cost per month	

- Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Register a device

A device must be registered with your IoT hub before it can connect. In this quickstart, you use the Azure Cloud Shell to register a simulated device.

- Run the following commands in Azure Cloud Shell to add the IoT Hub CLI extension and to create the device identity.

YourIoTHubName: Replace this placeholder below with the name you choose for your IoT hub.

MyDevice: This is the name given for the registered device. Use MyDevice as shown. If you choose a different name for your device, you will also need to use that name throughout this article, and update the device name in the sample applications before you run them.

```
az extension add --name azure-cli-iot-ext
az iot hub device-identity create --hub-name YourIoTHubName --device-id MyDevice
```

- Run the following commands in Azure Cloud Shell to get the *device connection string* for the device you just registered:

YourIoTHubName: Replace this placeholder below with the name you choose for your IoT hub.

```
az iot hub device-identity show-connection-string --hub-name YourIoTHubName --device-id MyDevice --output table
```

Make a note of the device connection string, which looks like the following example:

```
HostName={YourIoTHubName}.azure-devices.net;DeviceId=MyDevice;SharedAccessKey={YourSharedAccessKey}
```

You use this value later in the quickstart.

- You also need the *service connection string* from your IoT hub to enable the service-side application to

connect to your IoT hub and establish a device stream. The following command retrieves this value for your IoT hub:

YourIoTHubName: Replace this placeholder below with the name you choose for your IoT hub.

```
az iot hub show-connection-string --policy-name service --hub-name YourIoTHubName
```

Make a note of the returned value, which looks like this:

```
"HostName={YourIoTHubName}.azure-devices.net;SharedAccessKeyName=service;SharedAccessKey={YourSharedAccessKey}"
```

Communicate between device and service via device streams

Run the service-side application

Navigate to `iot-hub/Quickstarts/device-streams-echo/service` in your unzipped project folder. You will need the following information handy:

PARAMETER NAME	PARAMETER VALUE
<code>ServiceConnectionString</code>	Provide the service connection string of your IoT Hub.
<code>DeviceId</code>	Provide the ID of the device you created earlier, for example, MyDevice.

Compile and run the code as follows:

```
cd ./iot-hub/Quickstarts/device-streams-echo/service/  
  
# Build the application  
dotnet build  
  
# Run the application  
# In Linux/MacOS  
dotnet run "<ServiceConnectionString>" "<MyDevice>"  
  
# In Windows  
dotnet run <ServiceConnectionString> <MyDevice>
```

NOTE

A timeout occurs if the device-side application doesn't respond in time.

Run the device-side application

Navigate to `iot-hub/Quickstarts/device-streams-echo/device` directory in your unzipped project folder. You will need the following information handy:

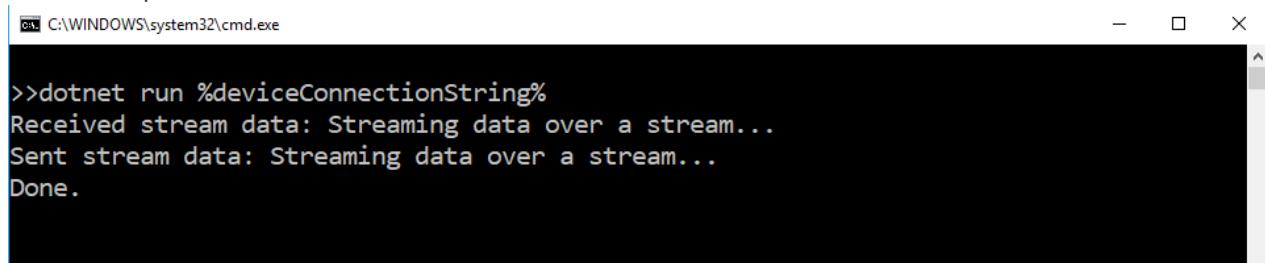
PARAMETER NAME	PARAMETER VALUE
<code>DeviceConnectionString</code>	Provide the device connection string of your IoT Hub.

Compile and run the code as follows:

```
cd ./iot-hub/Quickstarts/device-streams-echo/device/  
  
# Build the application  
dotnet build  
  
# Run the application  
# In Linux/MacOS  
dotnet run "<DeviceConnectionString>"  
  
# In Windows  
dotnet run <DeviceConnectionString>
```

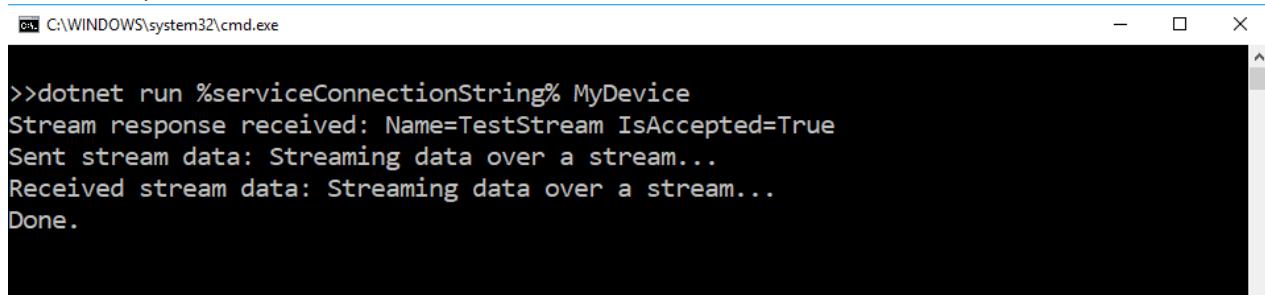
At the end of the last step, the service-side program will initiate a stream to your device and once established will send a string buffer to the service over the stream. In this sample, the service-side program simply echoes back the same data to the device, demonstrating successful bidirectional communication between the two applications. See figure below.

Console output on the device-side:



```
C:\WINDOWS\system32\cmd.exe  
  
>>dotnet run %deviceConnectionString%  
Received stream data: Streaming data over a stream...  
Sent stream data: Streaming data over a stream...  
Done.
```

Console output on the service-side:



```
C:\WINDOWS\system32\cmd.exe  
  
>>dotnet run %serviceConnectionString% MyDevice  
Stream response received: Name=TestStream IsAccepted=True  
Sent stream data: Streaming data over a stream...  
Received stream data: Streaming data over a stream...  
Done.
```

The traffic being sent over the stream will be tunneled through IoT Hub rather than being sent directly. This provides [these benefits](#).

Clean up resources

If you will be continuing to the next recommended article, you can keep the resources you've already created and reuse them.

Otherwise, you can delete the Azure resources created in this article to avoid charges.

IMPORTANT

Deleting a resource group is irreversible. The resource group and all the resources contained in it are permanently deleted. Make sure that you do not accidentally delete the wrong resource group or resources. If you created the IoT Hub inside an existing resource group that contains resources you want to keep, only delete the IoT Hub resource itself instead of deleting the resource group.

To delete a resource group by name:

1. Sign in to the [Azure portal](#) and click **Resource groups**.

2. In the **Filter by name...** textbox, type the name of the resource group containing your IoT Hub.

3. To the right of your resource group in the result list, click ... then **Delete resource group**.

The screenshot shows the Azure Resource Groups blade. On the left, there's a sidebar with 'Create a resource', 'Home', 'Dashboard', 'All services', 'FAVORITES' (which includes 'All resources' and 'Resource groups'), 'App Services', and 'SQL databases'. The 'Resource groups' item is highlighted with a red box. The main area is titled 'Resource groups' and shows a table with one item selected: 'IoTHubDeviceStreamsRG'. The table has columns for NAME, SUBSCRIPTION, and LOCATION. The 'NAME' column shows 'IoTHubDeviceStreamsRG'. The 'SUBSCRIPTION' column shows 'OneDeploy'. The 'LOCATION' column shows 'East US'. A context menu is open over the selected row, with the option 'Delete resource group' highlighted with a red box. Other options in the menu include '...'. At the top of the blade, there are buttons for 'Add', 'Edit columns', 'Refresh', 'Assign tags', and 'Export to CSV'. There are also filters for 'Subscriptions', 'Locations', 'Tags', and 'Grouping'.

4. You will be asked to confirm the deletion of the resource group. Type the name of your resource group again to confirm, and then click **Delete**. After a few moments, the resource group and all of its contained resources are deleted.

Next steps

In this quickstart, you have set up an IoT hub, registered a device, established a device stream between C# applications on the device and service side, and used the stream to send data back and forth between the applications.

Use the links below to learn more about device streams:

[Device streams overview](#)

Quickstart: Communicate to a device application in Node.js via IoT Hub device streams (preview)

1/29/2019 • 6 minutes to read

IoT Hub device streams allow service and device applications to communicate in a secure and firewall-friendly manner. During public preview, Node.js SDK only supports device streams on the service side. As a result, this quickstart only covers instructions to run the service-side application. You should run an accompanying device-side application which is available in [C quickstart](#) or [C# quickstart](#) guides.

The service-side Node.js application in this quickstart has the following functionalities:

- Creates a device stream to an IoT device.
- Reads input from command line and sends it to the device application, which will echo it back.

The code will demonstrate the initiation process of a device stream, as well as how to use to send and receive data.

Open Azure Cloud Shell

Azure Cloud Shell is a free, interactive shell that you can use to run the steps in this article. Common Azure tools are preinstalled and configured in Cloud Shell for you to use with your account. Just select the **Copy** button to copy the code, paste it in Cloud Shell, and then press Enter to run it. There are a few ways to open Cloud Shell:

Select Try It in the upper-right corner of a code block.	
Open Cloud Shell in your browser.	
Select the Cloud Shell button on the menu in the upper-right corner of the Azure portal .	

If you don't have an Azure subscription, create a [free account](#) before you begin.

Prerequisites

To run the service-side application in this quickstart you need Node.js v4.x.x or later on your development machine.

You can download Node.js for multiple platforms from [Node.js.org](#).

You can verify the current version of Node.js on your development machine using the following command:

```
node --version
```

If you haven't already done so, download the sample Node.js project from <https://github.com/Azure-Samples/azure-iot-samples-node/archive/streams-preview.zip> and extract the ZIP archive.

Create an IoT hub

If you completed the previous [Quickstart: Send telemetry from a device to an IoT hub](#), you can skip this step.

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose **+Create a resource**, then choose **Internet of Things**.
3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.

The screenshot shows the 'Basics' step of the IoT hub creation wizard. At the top, there's a breadcrumb navigation: Home > All resources > New > Marketplace > Everything > IoT Hub > IoT hub. Below that, it says 'IoT hub Microsoft'. There are three tabs: Basics (which is selected), Size and scale, and Review + create. A sub-instruction says 'Create an IoT Hub to help you connect, monitor, and manage billions of your IoT assets.' with a 'Learn More' link. The 'PROJECT DETAILS' section contains four fields: 'Subscription' (dropdown), 'Resource Group' (dropdown showing '(New) IoTHubDeviceStreamsRG' with a 'Create new' link), 'Region' (dropdown showing 'Central US EUAP'), and 'IoT Hub Name' (input field containing 'IoTHubDeviceStreams' with a green checkmark). The entire form has a light blue background.

Fill in the fields:

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Ensure you select a supported region (e.g., Central US or Central US EUAP).

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

IoT hub

Microsoft

[Basics](#) [Size and scale](#) [Review + create](#)

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS

* Pricing and scale tier [?](#) F1: Free tier [▼](#)

[Learn how to choose the right IoT Hub tier for your solution](#)

Number of F1 IoT Hub units [?](#) 1

This determines your IoT Hub scale capability and can be changed as your need increases.

Pricing and scale tier ? F1	Device-to-cloud-messages ? Enabled
Messages per day ? 8,000	Message routing ? Enabled
Cost per month	Cloud-to-device commands ? Enabled
	IoT Edge ? Enabled
	Device management ? Enabled

[▼ Advanced Settings](#)

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: Ensure you select one of the standard (S1, S2, S3) or the Free (F1) tier. This choice can also be guided by the size of your fleet and the non-streaming workloads you expect in your hub (e.g., telemetry messages). For example, the free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: This choice depends on non-streaming workload you expect in your hub - you can select 1 for now.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

1. Click **Review + create** to review your choices. You see something similar to this screen.

IoT hub

Microsoft

Basics Size and scale Review + create

BASICS

Subscription	IoTHubDeviceStreamsRG
Resource Group	Central US EUAP
Region	IoTHubDeviceStreams

SIZE AND SCALE

Pricing and scale tier	F1
Number of F1 IoT Hub units	1
Messages per day	8,000
Cost per month	

- Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Register a device

If you completed the previous [Quickstart: Send telemetry from a device to an IoT hub](#), you can skip this step.

A device must be registered with your IoT hub before it can connect. In this quickstart, you use the Azure Cloud Shell to register a simulated device.

- Run the following commands in Azure Cloud Shell to add the IoT Hub CLI extension and to create the device identity.

YourIoTHubName: Replace this placeholder below with the name you chose for your IoT hub.

MyDevice: This is the name given for the registered device. Use MyDevice as shown. If you choose a different name for your device, you will also need to use that name throughout this article, and update the device name in the sample applications before you run them.

```
az extension add --name azure-cli-iot-ext
az iot hub device-identity create --hub-name YourIoTHubName --device-id MyDevice
```

- You also need a *service connection string* to enable the back-end application to connect to your IoT hub and retrieve the messages. The following command retrieves the service connection string for your IoT hub:

YourIoTHubName: Replace this placeholder below with the name you chose for your IoT hub.

```
az iot hub show-connection-string --policy-name service --hub-name YourIoTHubName
```

Make a note of the returned value, which looks like this:

```
"HostName={YourIoTHubName}.azure-devices.net;SharedAccessKeyName=service;SharedAccessKey={YourSharedAccessKey}"
```

Communicate between device and service via device streams

Run the device-side application

As mentioned earlier, IoT Hub Node.js SDK only supports device streams on the service side. For device-side application, use the accompanying device programs available in [C quickstart](#) or [C# quickstart](#) guides. Ensure the device-side application is running before proceeding to the next step.

Run the service-side application

Assuming the device-side application is running, follow the steps below to run the service-side application in Node.js:

- Provide your service credentials and device ID as environment variables.

```
# In Linux
export IOTHUB_CONNECTION_STRING=<provide_your_service_connection_string>
export STREAMING_TARGET_DEVICE="MyDevice"

# In Windows
SET IOTHUB_CONNECTION_STRING=<provide_your_service_connection_string>
SET STREAMING TARGET DEVICE=MyDevice
```

Change `MyDevice` to the device ID you chose for your device.

- Navigate to `Quickstarts/device-streams-service` in your unzipped project folder and run the sample using node.

```
cd azure-iot-samples-node-streams-preview/iot-hub/Quickstarts/device-streams-service

# Install the preview service SDK, and other dependencies
npm install azure-iothub@streams-preview
npm install

node echo.js
```

At the end of the last step, the service-side program will initiate a stream to your device and once established will send a string buffer to the service over the stream. In this sample, the service-side program simply reads the `stdin` on the terminal and sends it to the device, which will then echo it back. This demonstrates successful bidirectional communication between the two applications.

Console output on the service-side:

```
C:\WINDOWS\system32\cmd.exe - node c2d_streaming_echo.js
>>>set STREAMING_TARGET_DEVICE=MyDevice

>>>node c2d_streaming_echo.js
initiating stream
{
  "authorizationToken": "REDACTED"
}
{
  "isAccepted": true,
  "uri": "wss://eastus2euap.eastus2euap-001.streams.azure-devices.net:443/bridges/IoT
HubDeviceStreams/MyDevice/REDACTED",
  "streamName": "TestStream"
}
Hello World
Hello World
```

You can then terminate the program by pressing enter again.

Clean up resources

If you will be continuing to the next recommended article, you can keep the resources you've already created and reuse them.

Otherwise, you can delete the Azure resources created in this article to avoid charges.

IMPORTANT

Deleting a resource group is irreversible. The resource group and all the resources contained in it are permanently deleted. Make sure that you do not accidentally delete the wrong resource group or resources. If you created the IoT Hub inside an existing resource group that contains resources you want to keep, only delete the IoT Hub resource itself instead of deleting the resource group.

To delete a resource group by name:

1. Sign in to the [Azure portal](#) and click **Resource groups**.
2. In the **Filter by name...** textbox, type the name of the resource group containing your IoT Hub.
3. To the right of your resource group in the result list, click ... then **Delete resource group**.

The screenshot shows the Azure Resource Groups blade. On the left sidebar, 'Resource groups' is selected and highlighted with a red box. In the main area, a resource group named 'IoTHubDeviceStreams...' is listed in the results table. A red box highlights the search bar at the top and the 'Delete resource group' button in the table row for the selected resource group.

4. You will be asked to confirm the deletion of the resource group. Type the name of your resource group again to confirm, and then click **Delete**. After a few moments, the resource group and all of its contained resources are deleted.

Next steps

In this quickstart, you have set up an IoT hub, registered a device, established a device stream between applications on the device and service side, and used the stream to send data back and forth between the applications.

Use the links below to learn more about device streams:

[Device streams overview](#)

Quickstart: Communicate to a device application in C via IoT Hub device streams (preview)

2/4/2019 • 7 minutes to read

[IoT Hub device streams](#) allow service and device applications to communicate in a secure and firewall-friendly manner. During public preview, the C SDK only supports device streams on the device side. As a result, this quickstart only covers instructions to run the device-side application. You should run an accompanying service-side application, which is available in [C# quickstart](#) or [Node.js quickstart](#) guides.

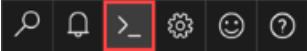
The device-side C application in this quickstart has the following functionality:

- Establish a device stream to an IoT device.
- Receive data sent from the service-side and echo it back.

The code will demonstrate the initiation process of a device stream, as well as how to use to send and receive data.

Open Azure Cloud Shell

Azure Cloud Shell is a free, interactive shell that you can use to run the steps in this article. Common Azure tools are preinstalled and configured in Cloud Shell for you to use with your account. Just select the **Copy** button to copy the code, paste it in Cloud Shell, and then press Enter to run it. There are a few ways to open Cloud Shell:

Select Try It in the upper-right corner of a code block.	
Open Cloud Shell in your browser.	
Select the Cloud Shell button on the menu in the upper-right corner of the Azure portal .	

If you don't have an Azure subscription, create a [free account](#) before you begin.

Prerequisites

- Install [Visual Studio 2017](#) with the '[Desktop development with C++](#)' workload enabled.
- Install the latest version of [Git](#).

Prepare the development environment

For this quickstart, you will be using the [Azure IoT device SDK for C](#). You will prepare a development environment used to clone and build the [Azure IoT C SDK](#) from GitHub. The SDK on GitHub includes the sample code used in this quickstart.

1. Download version 3.11.4 of the [CMake build system](#). Verify the downloaded binary using the corresponding cryptographic hash value. The following example used Windows PowerShell to verify the cryptographic hash for version 3.11.4 of the x64 MSI distribution:

```
PS C:\Downloads> $hash = get-filehash .\cmake-3.11.4-win64-x64.msi
PS C:\Downloads> $hash.Hash -eq "56e3605b8e49cd446f3487da88fcc38cb9c3e9e99a20f5d4bd63e54b7a35f869"
True
```

The following hash values for version 3.11.4 were listed on the CMake site at the time of this writing:

```
6dab016a6b82082b8bcd0f4d1e53418d6372015dd983d29367b9153f1a376435 cmake-3.11.4-Linux-x86_64.tar.gz
72b3b82b6d2c2f3a375c0d2799c01819df8669dc55694c8b8daaf6232e873725 cmake-3.11.4-win32-x86.msi
56e3605b8e49cd446f3487da88fcc38cb9c3e9e99a20f5d4bd63e54b7a35f869 cmake-3.11.4-win64-x64.msi
```

2. Open a command prompt or Git Bash shell. Execute the following command to clone the [Azure IoT C SDK GitHub repository](#):

```
git clone https://github.com/Azure/azure-iot-sdk-c.git --recursive -b public-preview
```

The size of this repository is currently around 220 MB.

3. Create a `cmake` subdirectory in the root directory of the git repository, and navigate to that folder.

```
cd azure-iot-sdk-c
mkdir cmake
cd cmake
```

4. Run the following command that builds a version of the SDK specific to your development client platform.

In Windows, a Visual Studio solution for the simulated device will be generated in the `cmake` directory.

```
# In Linux
cmake ..
make -j
```

In Windows, run the following commands in Developer Command Prompt for your Visual Studio 2015 or 2017 prompt:

```
rem In Windows
rem For VS2015
cmake .. -G "Visual Studio 15 2015"

rem Or for VS2017
cmake .. -G "Visual Studio 15 2017"

rem Then build the project
cmake --build . -- /m /p:Configuration=Release
```

Create an IoT hub

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose **+Create a resource**, then choose **Internet of Things**.
3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.

IoT hub

Microsoft

Basics Size and scale Review + create

Create an IoT Hub to help you connect, monitor, and manage billions of your IoT assets. [Learn More](#)

PROJECT DETAILS

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

* Subscription <small>i</small>	<input type="text"/>	<input type="button" value="▼"/>
* Resource Group <small>i</small>	(New) IoTHubDeviceStreamsRG Create new	<input type="button" value="▼"/>
* Region <small>i</small>	Central US EUAP	<input type="button" value="▼"/>
* IoT Hub Name <small>i</small>	IoTHubDeviceStreams	<input checked="" type="checkbox"/>

Fill in the fields:

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Ensure you select a supported region (e.g., Central US or Central US EUAP).

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

IoT hub

Microsoft

[Basics](#) [Size and scale](#) [Review + create](#)

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS

* Pricing and scale tier [?](#) F1: Free tier [▼](#)

[Learn how to choose the right IoT Hub tier for your solution](#)

Number of F1 IoT Hub units [?](#) 1

This determines your IoT Hub scale capability and can be changed as your need increases.

Pricing and scale tier ? F1	Device-to-cloud-messages ? Enabled
Messages per day ? 8,000	Message routing ? Enabled
Cost per month	Cloud-to-device commands ? Enabled
	IoT Edge ? Enabled
	Device management ? Enabled

[▼ Advanced Settings](#)

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: Ensure you select one of the standard (S1, S2, S3) or the Free (F1) tier. This choice can also be guided by the size of your fleet and the non-streaming workloads you expect in your hub (e.g., telemetry messages). For example, the free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: This choice depends on non-streaming workload you expect in your hub - you can select 1 for now.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

1. Click **Review + create** to review your choices. You see something similar to this screen.

IoT hub

Microsoft

Basics Size and scale Review + create

BASICS

Subscription ?	IoTHubDeviceStreamsRG
Resource Group ?	Central US EUAP
Region ?	IoTHubDeviceStreams
IoT Hub Name ?	

SIZE AND SCALE

Pricing and scale tier ?	F1
Number of F1 IoT Hub units ?	1
Messages per day ?	8,000
Cost per month	

- Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Register a device

A device must be registered with your IoT hub before it can connect. In this section, you will use the Azure Cloud Shell with the [IoT extension](#) to register a simulated device.

- Run the following commands in Azure Cloud Shell to add the IoT Hub CLI extension and to create the device identity.

YourIoTHubName: Replace this placeholder below with the name you choose for your IoT hub.

MyDevice: This is the name given for the registered device. Use MyDevice as shown. If you choose a different name for your device, you will also need to use that name throughout this article, and update the device name in the sample applications before you run them.

```
az extension add --name azure-cli-iot-ext
az iot hub device-identity create --hub-name YourIoTHubName --device-id MyDevice
```

- Run the following commands in Azure Cloud Shell to get the *device connection string* for the device you just registered:

YourIoTHubName: Replace this placeholder below with the name you choose for your IoT hub.

```
az iot hub device-identity show-connection-string --hub-name YourIoTHubName --device-id MyDevice --output table
```

Make a note of the device connection string, which looks like the following example:

```
HostName={YourIoTHubName}.azure-devices.net;DeviceId=MyDevice;SharedAccessKey={YourSharedAccessKey}
```

You use this value later in the quickstart.

Communicate between device and service via device streams

Run the device-side application

To run the device-side application, you need to perform the following steps:

- Set up your development environment by using the instructions in this [article about device streams](#).

- Provide your device credentials by editing the source file

`iothub_client/samples/iothub_client_c2d_streaming_sample/iothub_client_c2d_streaming_sample.c` and provide your device connection string.

```
/* Paste in the your iothub connection string */
static const char* connectionString = "[device connection string]";
```

- Compile the code as follows:

```
# In Linux
# Go to the sample's folder cmake/iothub_client/samples/iothub_client_c2d_streaming_sample
make -j
```

```
# In Windows
# Go to the cmake folder at the root of repo
cmake --build . -- /m /p:Configuration=Release
```

- Run the compiled program:

```
# In Linux
# Go to sample's folder
cmake/iothub_client/samples/iothub_client_c2d_streaming_sample
./iothub_client_c2d_streaming_sample
```

```
# In Windows
# Go to sample's release folder
cmake\iothub_client\samples\iothub_client_c2d_streaming_sample\Release
iothub_client_c2d_streaming_sample.exe
```

Run the service-side application

As mentioned earlier, IoT Hub C SDK only supports device streams on the device side. For the service-side application, use the accompanying service programs available in [C# quickstart](#) or [Node.js quickstart](#) guides.

Clean up resources

If you will be continuing to the next recommended article, you can keep the resources you've already created and reuse them.

Otherwise, you can delete the Azure resources created in this article to avoid charges.

IMPORTANT

Deleting a resource group is irreversible. The resource group and all the resources contained in it are permanently deleted. Make sure that you do not accidentally delete the wrong resource group or resources. If you created the IoT Hub inside an existing resource group that contains resources you want to keep, only delete the IoT Hub resource itself instead of deleting the resource group.

To delete a resource group by name:

1. Sign in to the [Azure portal](#) and click **Resource groups**.
2. In the **Filter by name...** textbox, type the name of the resource group containing your IoT Hub.
3. To the right of your resource group in the result list, click ... then **Delete resource group**.

The screenshot shows the Azure Resource Groups blade. On the left is a navigation menu with options like 'Create a resource', 'Home', 'Dashboard', 'All services', 'FAVORITES' (which includes 'All resources' and 'Resource groups'), 'App Services', and 'SQL databases'. The 'Resource groups' item is highlighted with a red box. The main area is titled 'Resource groups' and shows a table with one item selected: 'IoTHubDeviceStreamsRG'. The 'NAME' column shows 'IoTHubDeviceStreamsRG', the 'SUBSCRIPTION' column shows 'OneDeploy', and the 'LOCATION' column shows 'West US'. To the right of the table is a context menu with options: 'Delete resource group' (which is highlighted with a red box) and '...'. At the top of the blade, there are filter options for 'Subscriptions', 'Subscription settings', and dropdowns for 'All locations', 'All tags', and 'No grouping'.

4. You will be asked to confirm the deletion of the resource group. Type the name of your resource group again to confirm, and then click **Delete**. After a few moments, the resource group and all of its contained resources are deleted.

Next steps

In this quickstart, you have set up an IoT hub, registered a device, sent simulated telemetry to the hub using a C application, and read the telemetry from the hub using the Azure Cloud Shell.

Use the links below to learn more about device streams:

[Device streams overview](#)

Quickstart: SSH/RDP over IoT Hub device streams using C# proxy applications (preview)

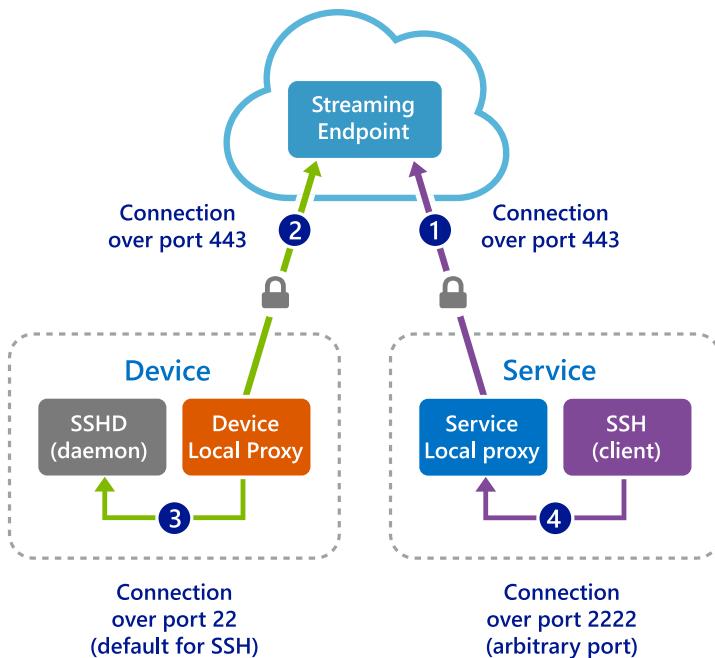
2/6/2019 • 9 minutes to read

IoT Hub device streams allow service and device applications to communicate in a secure and firewall-friendly manner. This quickstart guide involves two C# programs that enable client/server application traffic (such as SSH and RDP) to be sent over a device stream established through IoT Hub. See [here](#) for an overview of the setup.

We first describe the setup for SSH (using port 22). We then describe how to modify the setup's port for RDP. Since device streams are application and protocol agnostic, the same sample can be modified to accommodate other types of application traffic. This usually only involves changing the communication port to the one used by the intended application.

How it works?

Figure below illustrates the setup of how the device- and service-local proxy programs in this sample will enable end-to-end connectivity between SSH client and SSH daemon. Here, we assume that the daemon is running on the same device as the device-local proxy.



1. Service-local proxy connects to IoT hub and initiates a device stream to the target device using its device ID.
2. Device-local proxy completes the stream initiation handshake and establishes an end-to-end streaming tunnel through IoT Hub's streaming endpoint to the service side.

3. Device-local proxy connects to the SSH daemon (SSHD) listening on port 22 on the device (this port is configurable, as described [below](#)).
4. Service-local proxy awaits for new SSH connections from the user by listening on a designated port which in this case is port 2222 (this is also configurable, as described [below](#)). When user connects via SSH client, the tunnel enables application traffic to be exchanged between the SSH client and server programs.

NOTE

SSH traffic being sent over the stream will be tunneled through IoT Hub's streaming endpoint rather than being sent directly between service and device. This provides [these benefits](#).

Open Azure Cloud Shell

Azure Cloud Shell is a free, interactive shell that you can use to run the steps in this article. Common Azure tools are preinstalled and configured in Cloud Shell for you to use with your account. Just select the **Copy** button to copy the code, paste it in Cloud Shell, and then press Enter to run it. There are a few ways to open Cloud Shell:

Select Try It in the upper-right corner of a code block.	
Open Cloud Shell in your browser.	
Select the Cloud Shell button on the menu in the upper-right corner of the Azure portal .	

If you don't have an Azure subscription, create a [free account](#) before you begin.

Prerequisites

The two sample applications you run in this quickstart are written using C#. You need the .NET Core SDK 2.1.0 or greater on your development machine.

You can download the .NET Core SDK for multiple platforms from [.NET](#).

You can verify the current version of C# on your development machine using the following command:

```
dotnet --version
```

Download the sample C# project from <https://github.com/Azure-Samples/azure-iot-samples-csharp/archive/master.zip> and extract the ZIP archive.

Create an IoT hub

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose **+Create a resource**, then choose **Internet of Things**.
3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.

IoT hub

Microsoft

Basics Size and scale Review + create

Create an IoT Hub to help you connect, monitor, and manage billions of your IoT assets. [Learn More](#)

PROJECT DETAILS

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

* Subscription <small>i</small>	<input type="text"/>	<input type="button" value="▼"/>
* Resource Group <small>i</small>	(New) IoTHubDeviceStreamsRG Create new	<input type="button" value="▼"/>
* Region <small>i</small>	Central US EUAP	<input type="button" value="▼"/>
* IoT Hub Name <small>i</small>	IoTHubDeviceStreams	<input checked="" type="checkbox"/>

Fill in the fields:

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Ensure you select a supported region (e.g., Central US or Central US EUAP).

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

IoT hub

Microsoft

[Basics](#) [Size and scale](#) [Review + create](#)

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS* Pricing and scale tier [?](#)

F1: Free tier

[Learn how to choose the right IoT Hub tier for your solution](#)Number of F1 IoT Hub units [?](#)

This determines your IoT Hub scale capability and can be changed as your need increases.

Pricing and scale tier ? F1	Device-to-cloud-messages ? Enabled
Messages per day ? 8,000	Message routing ? Enabled
Cost per month	Cloud-to-device commands ? Enabled
	IoT Edge ? Enabled
	Device management ? Enabled

[Advanced Settings](#)

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: Ensure you select one of the standard (S1, S2, S3) or the Free (F1) tier. This choice can also be guided by the size of your fleet and the non-streaming workloads you expect in your hub (e.g., telemetry messages). For example, the free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: This choice depends on non-streaming workload you expect in your hub - you can select 1 for now.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

1. Click **Review + create** to review your choices. You see something similar to this screen.

IoT hub

Microsoft

Basics Size and scale Review + create

BASICS

Subscription ?	IoTHubDeviceStreamsRG
Resource Group ?	Central US EUAP
Region ?	IoTHubDeviceStreams

SIZE AND SCALE

Pricing and scale tier ?	F1
Number of F1 IoT Hub units ?	1
Messages per day ?	8,000
Cost per month	

- Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Register a device

A device must be registered with your IoT hub before it can connect. In this quickstart, you use the Azure Cloud Shell to register a simulated device.

- Run the following commands in Azure Cloud Shell to add the IoT Hub CLI extension and to create the device identity.

YourIoTHubName: Replace this placeholder below with the name you choose for your IoT hub.

MyDevice: This is the name given for the registered device. Use MyDevice as shown. If you choose a different name for your device, you will also need to use that name throughout this article, and update the device name in the sample applications before you run them.

```
az extension add --name azure-cli-iot-ext
az iot hub device-identity create --hub-name YourIoTHubName --device-id MyDevice
```

- Run the following commands in Azure Cloud Shell to get the *device connection string* for the device you just registered:

YourIoTHubName: Replace this placeholder below with the name you choose for your IoT hub.

```
az iot hub device-identity show-connection-string --hub-name YourIoTHubName --device-id MyDevice --output table
```

Make a note of the device connection string, which looks like the following example:

```
HostName={YourIoTHubName}.azure-devices.net;DeviceId=MyDevice;SharedAccessKey={YourSharedAccessKey}
```

You use this value later in the quickstart.

- You also need the *service connection string* from your IoT hub to enable the service-side application to

connect to your IoT hub and establish a device stream. The following command retrieves this value for your IoT hub:

YourIoTHubName: Replace this placeholder below with the name you choose for your IoT hub.

```
az iot hub show-connection-string --policy-name service --hub-name YourIoTHubName
```

Make a note of the returned value, which looks like this:

```
"HostName={YourIoTHubName}.azure-devices.net;SharedAccessKeyName=service;SharedAccessKey={YourSharedAccessKey}"
```

SSH to a device via device streams

Run the device-local proxy

Navigate to `device-streams-proxy/device` in your unzipped project folder. You will need the following information handy:

ARGUMENT NAME	ARGUMENT VALUE
<code>deviceConnectionString</code>	The connection string of the device you created earlier.
<code>targetServiceHostName</code>	The IP address where SSH server listens on (this would be <code>localhost</code> if the same IP where device-local proxy is running).
<code>targetServicePort</code>	The port used by your application protocol (by default, this would be port 22 for SSH).

Compile and run the code as follows:

```
cd ./iot-hub/Quickstarts/device-streams-proxy/device/  
  
# Build the application  
dotnet build  
  
# Run the application  
# In Linux/MacOS  
dotnet run $deviceConnectionString localhost 22  
  
# In Windows  
dotnet run %deviceConnectionString% localhost 22
```

Run the service-local proxy

Navigate to `device-streams-proxy/service` in your unzipped project folder. You will need the following information handy:

PARAMETER NAME	PARAMETER VALUE
<code>iotHubConnectionString</code>	The service connection string of your IoT Hub.
<code>deviceId</code>	The identifier of the device you created earlier.

PARAMETER NAME	PARAMETER VALUE
localPortNumber	A local port where your SSH client will connect to. We use port 2222 in this sample, but you could modify this to other arbitrary numbers.

Compile and run the code as follows:

```
cd ./iot-hub/Quickstarts/device-streams-proxy/service/

# Build the application
dotnet build

# Run the application
# In Linux/MacOS
dotnet run $serviceConnectionString MyDevice 2222

# In Windows
dotnet run %serviceConnectionString% MyDevice 2222
```

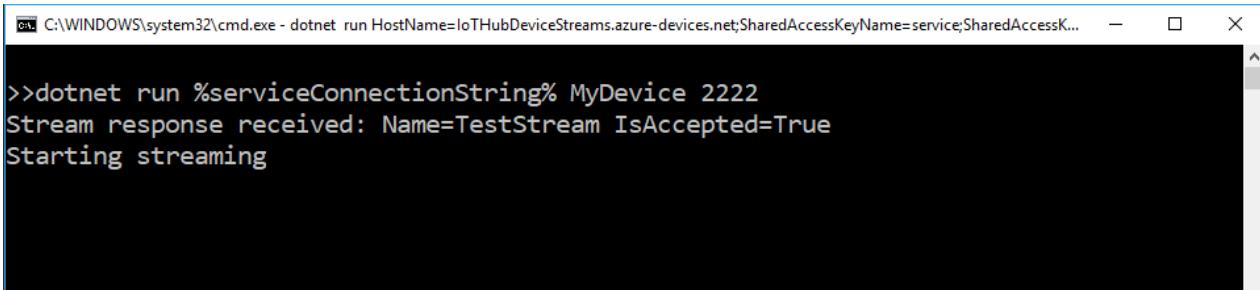
Run SSH client

Now use your SSH client program and connect to service-local proxy on port 2222 (instead of the SSH daemon directly).

```
ssh <username>@localhost -p 2222
```

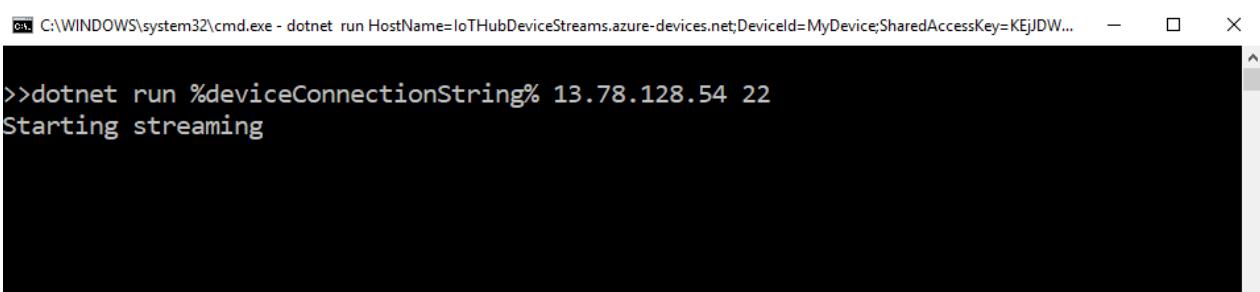
At this point, you will be presented with the SSH login prompt to enter your credentials.

Console output on the service-side (the service-local proxy listens on port 2222):



```
>>dotnet run %serviceConnectionString% MyDevice 2222
Stream response received: Name=TestStream IsAccepted=True
Starting streaming
```

Console output on the device-local proxy which connects to the SSH daemon at `IP_address:22`:



```
>>dotnet run %deviceConnectionString% 13.78.128.54 22
Starting streaming
```

Console output of the SSH client program (SSH client communicates to SSH daemon by connecting to port 22 where service-local proxy is listening on):

```

root@reza@ubuntu-bash-2: ~
root@RezaYoga:/mnt/c/Users/rezas# ssh -p 2222 rezas@localhost
rezas@localhost's password:
Welcome to Ubuntu 16.04.5 LTS (GNU/Linux 4.15.0-1035-azure x8
6_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

Get cloud support with Ubuntu Advantage Cloud Guest:
http://www.ubuntu.com/business/services/cloud

43 packages can be updated.
0 updates are security updates.

New release '18.04.1 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

*** System restart required ***
Last login: Fri Jan 11 00:11:34 2019 from 167.220.2.127
@ubuntu-vm:~$
```

RDP to a device via device streams

The setup for RDP is very similar to SSH (described above). We basically need to use the RDP destination IP and port 3389 instead and use RDP client (instead of SSH client).

Run the device-local proxy (RDP)

Navigate to `device-streams-proxy/device` in your unzipped project folder. You will need the following information handy:

ARGUMENT NAME	ARGUMENT VALUE
<code>DeviceConnectionString</code>	The connection string of the device you created earlier.
<code>targetServiceHostName</code>	The hostname or IP address where RDP server runs (this would be <code>localhost</code> if the same IP where device-local proxy is running).
<code>targetServicePort</code>	The port used by your application protocol (by default, this would be port 3389 for RDP).

Compile and run the code as follows:

```

cd ./iot-hub/Quickstarts/device-streams-proxy/device

# Run the application
# In Linux/MacOS
dotnet run $DeviceConnectionString localhost 3389

# In Windows
dotnet run %DeviceConnectionString% localhost 3389
```

Run the service-local proxy (RDP)

Navigate to `device-streams-proxy/service` in your unzipped project folder. You will need the following information handy:

PARAMETER NAME	PARAMETER VALUE
<code>iotHubConnectionString</code>	The service connection string of your IoT Hub.
<code>deviceId</code>	The identifier of the device you created earlier.
<code>localPortNumber</code>	A local port where your SSH client will connect to. We use port 2222 in this sample, but you could modify this to other arbitrary numbers.

Compile and run the code as follows:

```
cd ./iot-hub/Quickstarts/device-streams-proxy/service/  
  
# Build the application  
dotnet build  
  
# Run the application  
# In Linux/MacOS  
dotnet run $serviceConnectionString MyDevice 2222  
  
# In Windows  
dotnet run %serviceConnectionString% MyDevice 2222
```

Run RDP client

Now use your RDP client program and connect to service-local proxy on port 2222 (this was an arbitrary available port you chose earlier).



Clean up resources

If you will be continuing to the next recommended article, you can keep the resources you've already created and reuse them.

Otherwise, you can delete the Azure resources created in this article to avoid charges.

IMPORTANT

Deleting a resource group is irreversible. The resource group and all the resources contained in it are permanently deleted. Make sure that you do not accidentally delete the wrong resource group or resources. If you created the IoT Hub inside an existing resource group that contains resources you want to keep, only delete the IoT Hub resource itself instead of deleting the resource group.

To delete a resource group by name:

1. Sign in to the [Azure portal](#) and click **Resource groups**.
2. In the **Filter by name...** textbox, type the name of the resource group containing your IoT Hub.
3. To the right of your resource group in the result list, click ... then **Delete resource group**.

The screenshot shows the Azure Resource Groups blade. On the left, there's a sidebar with 'Create a resource' and several navigation items: Home, Dashboard, All services, Favorites (with 'IoT HubDeviceStreams...' listed), All resources, and Resource groups (which is selected and highlighted with a red box). The main area is titled 'Resource groups' under 'Microsoft'. It has buttons for Add, Edit columns, Refresh, Assign tags, and Export to CSV. A 'Subscriptions' section shows '1 of 14 selected - Don't see a subscription? Open Directory + Subscription settings'. Below this, a table lists one item: 'IoTHubDeviceStreamsRG' under 'NAME', 'OneDeploy' under 'SUBSCRIPTION', and 'All locations' under 'LOCATION'. To the right of the table, there are three buttons: 'Delete resource group' (which is highlighted with a red box), '...', and '...'. The entire row for the resource group is also highlighted with a red box.

4. You will be asked to confirm the deletion of the resource group. Type the name of your resource group again to confirm, and then click **Delete**. After a few moments, the resource group and all of its contained resources are deleted.

Next steps

In this quickstart, you have set up an IoT hub, registered a device, deployed a device- and a service-local proxy program to establish a device stream through IoT Hub, and used the proxies to tunnel SSH or RDP traffic. The same paradigm can accommodate other client/server protocols (where server runs on the device, e.g., SSH daemon).

Use the links below to learn more about device streams:

[Device streams overview](#)

Quickstart: SSH/RDP over IoT Hub device streams using Node.js proxy application (preview)

1/24/2019 • 7 minutes to read

IoT Hub device streams allow service and device applications to communicate in a secure and firewall-friendly manner. This quickstart guide describes execution of a Node.js proxy application running on the service side to enable SSH and RDP traffic to be sent to the device over a device stream. See [here](#) for an overview of the setup. During public preview, Node.js SDK only supports device streams on the service side. As a result, this quickstart guide only covers instructions to run the service-local proxy. You should run an accompanying device-local proxy which is available in [C quickstart](#) or [C# quickstart](#) guides.

We first describe the setup for SSH (using port 22). We then describe how to modify the setup for RDP (which uses port 3389). Since device streams are application and protocol agnostic, the same sample can be modified to accommodate other types of client/server application traffic (usually by modifying the communication port).

Open Azure Cloud Shell

Azure Cloud Shell is a free, interactive shell that you can use to run the steps in this article. Common Azure tools are preinstalled and configured in Cloud Shell for you to use with your account. Just select the **Copy** button to copy the code, paste it in Cloud Shell, and then press Enter to run it. There are a few ways to open Cloud Shell:

Select Try It in the upper-right corner of a code block.	
Open Cloud Shell in your browser.	
Select the Cloud Shell button on the menu in the upper-right corner of the Azure portal .	

If you don't have an Azure subscription, create a [free account](#) before you begin.

Prerequisites

To run the service-local application in this quickstart you need Node.js v4.x.x or later on your development machine.

You can download Node.js for multiple platforms from [nodejs.org](#).

You can verify the current version of Node.js on your development machine using the following command:

```
node --version
```

If you haven't already done so, download the sample Node.js project from <https://github.com/Azure-Samples/azure-iot-samples-node/archive/streams-preview.zip> and extract the ZIP archive.

Create an IoT hub

If you completed the previous [Quickstart: Send telemetry from a device to an IoT hub](#), you can skip this step.

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose **+Create a resource**, then choose **Internet of Things**.
3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.

The screenshot shows the 'Basics' step of the IoT hub creation wizard. At the top, there's a breadcrumb navigation: Home > All resources > New > Marketplace > Everything > IoT Hub > IoT hub. Below that, it says 'IoT hub Microsoft'. There are three tabs: Basics (which is selected), Size and scale, and Review + create. A sub-instruction says 'Create an IoT Hub to help you connect, monitor, and manage billions of your IoT assets.' with a 'Learn More' link. The 'PROJECT DETAILS' section contains four fields: 'Subscription' (dropdown), 'Resource Group' (dropdown showing '(New) IoTHubDeviceStreamsRG' with a 'Create new' link), 'Region' (dropdown showing 'Central US EUAP'), and 'IoT Hub Name' (input field containing 'IoTHubDeviceStreams' with a green checkmark). The entire form is enclosed in a light blue border.

Fill in the fields:

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Ensure you select a supported region (e.g., Central US or Central US EUAP).

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

IoT hub

Microsoft

[Basics](#) [Size and scale](#) [Review + create](#)

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS

* Pricing and scale tier [?](#) F1: Free tier [▼](#)

[Learn how to choose the right IoT Hub tier for your solution](#)

Number of F1 IoT Hub units [?](#) 1

This determines your IoT Hub scale capability and can be changed as your need increases.

Pricing and scale tier ? F1	Device-to-cloud-messages ? Enabled
Messages per day ? 8,000	Message routing ? Enabled
Cost per month	Cloud-to-device commands ? Enabled
	IoT Edge ? Enabled
	Device management ? Enabled

[▼ Advanced Settings](#)

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: Ensure you select one of the standard (S1, S2, S3) or the Free (F1) tier. This choice can also be guided by the size of your fleet and the non-streaming workloads you expect in your hub (e.g., telemetry messages). For example, the free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: This choice depends on non-streaming workload you expect in your hub - you can select 1 for now.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

1. Click **Review + create** to review your choices. You see something similar to this screen.

IoT hub

Microsoft

Basics Size and scale Review + create

BASICS

Subscription	IoTHubDeviceStreamsRG
Resource Group	Central US EUAP
Region	IoTHubDeviceStreams

SIZE AND SCALE

Pricing and scale tier	F1
Number of F1 IoT Hub units	1
Messages per day	8,000
Cost per month	

- Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Register a device

If you completed the previous [Quickstart: Send telemetry from a device to an IoT hub](#), you can skip this step.

A device must be registered with your IoT hub before it can connect. In this quickstart, you use the Azure Cloud Shell to register a simulated device.

- Run the following commands in Azure Cloud Shell to add the IoT Hub CLI extension and to create the device identity.

YourIoTHubName: Replace this placeholder below with the name you chose for your IoT hub.

MyDevice: This is the name given for the registered device. Use MyDevice as shown. If you choose a different name for your device, you will also need to use that name throughout this article, and update the device name in the sample applications before you run them.

```
az extension add --name azure-cli-iot-ext
az iot hub device-identity create --hub-name YourIoTHubName --device-id MyDevice
```

- You also need a *service connection string* to enable the back-end application to connect to your IoT hub and retrieve the messages. The following command retrieves the service connection string for your IoT hub:

YourIoTHubName: Replace this placeholder below with the name you chose for your IoT hub.

```
az iot hub show-connection-string --policy-name service --hub-name YourIoTHubName
```

Make a note of the returned value, which looks like this:

```
"HostName={YourIoTHubName}.azure-devices.net;SharedAccessKeyName=service;SharedAccessKey={YourSharedAccessKey}"
```

SSH to a device via device streams

Run the device-local proxy

As mentioned earlier, IoT Hub Node.js SDK only supports device streams on the service side. For device-local application, use the accompanying device proxy programs available in [C quickstart](#) or [C# quickstart](#) guides. Ensure the device-local proxy is running before proceeding to the next step.

Run the service-local proxy

Assuming that the [device-local proxy](#) is running, follow the steps below to run the service-local proxy written in Node.js.

- Provide your service credentials, the target device ID where SSH daemon runs, and the port number for the proxy running on the device as environment variables.

```
# In Linux
export IOTHUB_CONNECTION_STRING=<provide_your_service_connection_string>
export STREAMING_TARGET_DEVICE="MyDevice"
export PROXY_PORT=2222

# In Windows
SET IOTHUB_CONNECTION_STRING=<provide_your_service_connection_string>
SET STREAMING_TARGET_DEVICE=MyDevice
SET PROXY_PORT=2222
```

Change the values above to match your device ID and connection string.

- Navigate to `Quickstarts/device-streams-service` in your unzipped project folder and run the service-local proxy.

```
cd azure-iot-samples-node-streams-preview/iot-hub/Quickstarts/device-streams-service

# Install the preview service SDK, and other dependencies
npm install azure-iothub@streams-preview
npm install

# Run the service-local proxy application
node proxy.js
```

SSH to your device via device streams

In Linux, run SSH using `ssh $USER@localhost -p 2222` on a terminal. In Windows, use your favorite SSH client (e.g., PuTTY).

Console output on the service-local after SSH session is established (the service-local proxy listens on port 2222):



```
C:\WINDOWS\system32\cmd.exe - node c2d_streaming_proxy.js
listening on port: 2222...
initiating stream
results received from the device: {
  "authorizationToken": "REDACTED"
  "isAccepted": true,
  "uri": "wss://eastus2euap.eastus2euap-001.streams.azure-devices.net:443/bridges/IoT
HubDeviceStreams/MyDevice/REDACTED",
  "streamName": "TestStream"
}
got websocket - creating local server on port 2222
```

Console output of the SSH client program (SSH client communicates to SSH daemon by connecting to port 22

where service-local proxy is listening on):

```
rezas@ubuntu-bug-bash-2: ~
root@RezaYoga:/mnt/c/Users/rezas# ssh -p 2222 rezas@localhost
rezas@localhost's password:
Welcome to Ubuntu 16.04.5 LTS (GNU/Linux 4.15.0-1035-azure x8
6_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

 Get cloud support with Ubuntu Advantage Cloud Guest:
 http://www.ubuntu.com/business/services/cloud

43 packages can be updated.
0 updates are security updates.

New release '18.04.1 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

*** System restart required ***
Last login: Fri Jan 11 00:11:34 2019 from 167.220.2.127
@ubuntu-vm:~$
```

RDP to your device via device streams

Now use your RDP client program and connect to service proxy on port 2222 (this was an arbitrary available port you chose earlier).

NOTE

Ensure that your device proxy is configured correctly for RDP and configured with RDP port 3389.



Clean up resources

If you will be continuing to the next recommended article, you can keep the resources you've already created and

reuse them.

Otherwise, you can delete the Azure resources created in this article to avoid charges.

IMPORTANT

Deleting a resource group is irreversible. The resource group and all the resources contained in it are permanently deleted. Make sure that you do not accidentally delete the wrong resource group or resources. If you created the IoT Hub inside an existing resource group that contains resources you want to keep, only delete the IoT Hub resource itself instead of deleting the resource group.

To delete a resource group by name:

1. Sign in to the [Azure portal](#) and click **Resource groups**.
2. In the **Filter by name...** textbox, type the name of the resource group containing your IoT Hub.
3. To the right of your resource group in the result list, click ... then **Delete resource group**.

The screenshot shows the Azure Resource Groups blade. On the left, there's a sidebar with 'Create a resource' and several service links: Home, Dashboard, All services, Favorites, All resources, and Resource groups (which is highlighted with a red box). The main area shows a table titled 'Resource groups' with one item listed: 'IoTHubDeviceStreams...' under 'Subscription settings'. The table has columns for NAME, SUBSCRIPTION, and LOCATION. A row for 'IoTHubDeviceStreamsRG' is selected, and its details are shown in a preview pane on the right. In this preview pane, a 'Delete resource group' button is highlighted with a red box.

4. You will be asked to confirm the deletion of the resource group. Type the name of your resource group again to confirm, and then click **Delete**. After a few moments, the resource group and all of its contained resources are deleted.

Next steps

In this quickstart, you have set up an IoT hub, registered a device, and deployed a service proxy program to enable RDP and SSH to an IoT device. The RDP and SSH traffic will be tunneled through a device stream through IoT Hub. This eliminates the need for direct connectivity to the device.

Use the links below to learn more about device streams:

[Device streams overview](#)

Quickstart: SSH/RDP over IoT Hub device streams using C proxy application (preview)

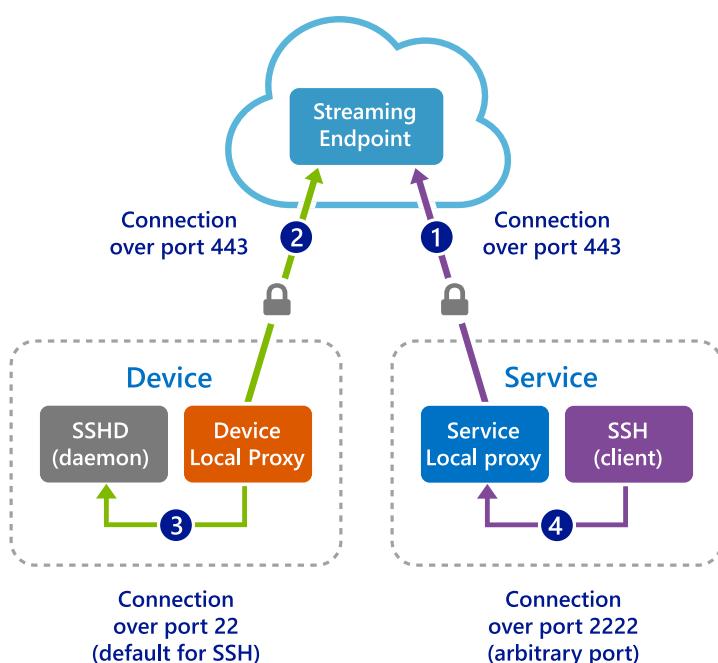
2/4/2019 • 9 minutes to read

IoT Hub device streams allow service and device applications to communicate in a secure and firewall-friendly manner. See [this page](#) for an overview of the setup.

This document describes the setup for tunneling SSH traffic (using port 22) through device streams. The setup for RDP traffic is similar and requires a simple configuration change. Since device streams are application and protocol agnostic, the present quickstart can be modified (by changing the communication ports) to accommodate other types of application traffic.

How it works?

Figure below illustrates the setup of how the device- and service-local proxy programs will enable end-to-end connectivity between SSH client and SSH daemon processes. During public preview, the C SDK only supports device streams on the device side. As a result, this quickstart only covers instructions to run the device-local proxy application. You should run an accompanying service-local proxy application which is available in [C# quickstart](#) or [Nodejs quickstart](#) guides.



1. Service-local proxy connects to IoT hub and initiates a device stream to the target device.
2. Device-local proxy completes the stream initiation handshake and establishes an end-to-end streaming tunnel through IoT Hub's streaming endpoint to the service side.

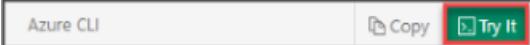
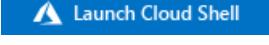
3. Device-local proxy connects to the SSH daemon (SSHD) listening on port 22 on the device (this is configurable, as described [below](#run-the-device-local-proxy-application)).
4. Service-local proxy awaits for new SSH connections from the user by listening on a designated port which in this case is port 2222 (this is also configurable, as described [below](#)). When user connects via SSH client, the tunnel enables SSH application traffic to be transferred between the SSH client and server programs.

NOTE

SSH traffic being sent over a device stream will be tunneled through IoT Hub's streaming endpoint rather than being sent directly between service and device. This provides [these benefits](#). Furthermore, the figure illustrates the SSH daemon running on the same device (or machine) as the device-local proxy. In this quickstart, providing the SSH daemon IP address allows device-local proxy and daemon to run on different machines as well.

Open Azure Cloud Shell

Azure Cloud Shell is a free, interactive shell that you can use to run the steps in this article. Common Azure tools are preinstalled and configured in Cloud Shell for you to use with your account. Just select the **Copy** button to copy the code, paste it in Cloud Shell, and then press Enter to run it. There are a few ways to open Cloud Shell:

Select Try It in the upper-right corner of a code block.	
Open Cloud Shell in your browser.	
Select the Cloud Shell button on the menu in the upper-right corner of the Azure portal .	

If you don't have an Azure subscription, create a [free account](#) before you begin.

Prerequisites

- Install [Visual Studio 2017](#) with the 'Desktop development with C++' workload enabled.
- Install the latest version of [Git](#).

Prepare the development environment

For this quickstart, you will be using the [Azure IoT device SDK for C](#). You will prepare a development environment used to clone and build the [Azure IoT C SDK](#) from GitHub. The SDK on GitHub includes the sample code used in this quickstart.

1. Download the version 3.11.4 of the [CMake build system](#) from [GitHub](#). Verify the downloaded binary using the corresponding cryptographic hash value. The following example used Windows PowerShell to verify the cryptographic hash for version 3.11.4 of the x64 MSI distribution:

```
PS C:\Downloads> $hash = get-filehash .\cmake-3.11.4-win64-x64.msi
PS C:\Downloads> $hash.Hash -eq "56e3605b8e49cd446f3487da88fcc38cb9c3e9e99a20f5d4bd63e54b7a35f869"
True
```

The following hash values for version 3.11.4 were listed on the CMake site at the time of this writing:

```
6dab016a6b82082b8bcd0f4d1e53418d6372015dd983d29367b9153f1a376435 cmake-3.11.4-Linux-x86_64.tar.gz
72b3b82b6d2c2f3a375c0d2799c01819df8669dc55694c8b8daaf6232e873725 cmake-3.11.4-win32-x86.msi
56e3605b8e49cd446f3487da88fcc38cb9c3e9e99a20f5d4bd63e54b7a35f869 cmake-3.11.4-win64-x64.msi
```

It is important that the Visual Studio prerequisites (Visual Studio and the 'Desktop development with C++' workload) are installed on your machine, **before** starting the `cmake` installation. Once the prerequisites are in place, and the download is verified, install the CMake build system.

2. Open a command prompt or Git Bash shell. Execute the following command to clone the [Azure IoT C SDK GitHub repository](#):

```
git clone https://github.com/Azure/azure-iot-sdk-c.git --recursive -b public-preview
```

The size of this repository is currently around 220 MB. You should expect this operation to take several minutes to complete.

3. Create a `cmake` subdirectory in the root directory of the git repository, and navigate to that folder.

```
cd azure-iot-sdk-c
mkdir cmake
cd cmake
```

4. Run the following command that builds a version of the SDK specific to your development client platform.

In Windows, a Visual Studio solution for the simulated device will be generated in the `cmake` directory.

```
# In Linux
cmake ..
make -j
```

In Windows, run the following commands in Developer Command Prompt for your Visual Studio 2015 or 2017 prompt:

```
rem In Windows
rem For VS2015
cmake .. -G "Visual Studio 15 2015"

rem Or for VS2017
cmake .. -G "Visual Studio 15 2017"

rem Then build the project
cmake --build . -- /m /p:Configuration=Release
```

Create an IoT hub

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose +**Create a resource**, then choose **Internet of Things**.
3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.

IoT hub

Microsoft

Basics Size and scale Review + create

Create an IoT Hub to help you connect, monitor, and manage billions of your IoT assets. [Learn More](#)

PROJECT DETAILS

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

* Subscription <small>i</small>	<input type="text"/>	<input type="button" value="▼"/>
* Resource Group <small>i</small>	(New) IoTHubDeviceStreamsRG Create new	<input type="button" value="▼"/>
* Region <small>i</small>	Central US EUAP	<input type="button" value="▼"/>
* IoT Hub Name <small>i</small>	IoTHubDeviceStreams	<input checked="" type="checkbox"/>

Fill in the fields:

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Ensure you select a supported region (e.g., Central US or Central US EUAP).

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

IoT hub

Microsoft

[Basics](#) [Size and scale](#) [Review + create](#)

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS

* Pricing and scale tier [?](#) F1: Free tier [▼](#)

[Learn how to choose the right IoT Hub tier for your solution](#)

Number of F1 IoT Hub units [?](#) 1

This determines your IoT Hub scale capability and can be changed as your need increases.

Pricing and scale tier ? F1	Device-to-cloud-messages ? Enabled
Messages per day ? 8,000	Message routing ? Enabled
Cost per month	Cloud-to-device commands ? Enabled
	IoT Edge ? Enabled
	Device management ? Enabled

[▼ Advanced Settings](#)

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: Ensure you select one of the standard (S1, S2, S3) or the Free (F1) tier. This choice can also be guided by the size of your fleet and the non-streaming workloads you expect in your hub (e.g., telemetry messages). For example, the free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: This choice depends on non-streaming workload you expect in your hub - you can select 1 for now.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

1. Click **Review + create** to review your choices. You see something similar to this screen.

IoT hub

Microsoft

Basics Size and scale Review + create

BASICS

Subscription ?	IoTHubDeviceStreamsRG
Resource Group ?	Central US EUAP
Region ?	IoTHubDeviceStreams
IoT Hub Name ?	

SIZE AND SCALE

Pricing and scale tier ?	F1
Number of F1 IoT Hub units ?	1
Messages per day ?	8,000
Cost per month	

- Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Register a device

A device must be registered with your IoT hub before it can connect. In this section, you will use the Azure Cloud Shell with the [IoT extension](#) to register a simulated device.

- Run the following commands in Azure Cloud Shell to add the IoT Hub CLI extension and to create the device identity.

YourIoTHubName: Replace this placeholder below with the name you choose for your IoT hub.

MyDevice: This is the name given for the registered device. Use MyDevice as shown. If you choose a different name for your device, you will also need to use that name throughout this article, and update the device name in the sample applications before you run them.

```
az extension add --name azure-cli-iot-ext
az iot hub device-identity create --hub-name YourIoTHubName --device-id MyDevice
```

- Run the following commands in Azure Cloud Shell to get the *device connection string* for the device you just registered:

YourIoTHubName: Replace this placeholder below with the name you choose for your IoT hub.

```
az iot hub device-identity show-connection-string --hub-name YourIoTHubName --device-id MyDevice --
output table
```

Make a note of the device connection string, which looks like the following example:

```
HostName={YourIoTHubName}.azure-devices.net;DeviceId=MyDevice;SharedAccessKey={YourSharedAccessKey}
```

You use this value later in the quickstart.

SSH to a device via device streams

Run the device-local proxy application

- Edit the source file

```
iothub_client/samples/iothub_client_c2d_streaming_proxy_sample/iothub_client_c2d_streaming_proxy_sample.c
```

and provide your device connection string, target device IP/hostname as well as the RDP port 22:

```
/* Paste in the your iothub connection string */
static const char* connectionString = "[Connection string of IoT Hub]";
static const char* localHost = "[IP/Host of your target machine]"; // Address of the local server to
connect to.
static const size_t localPort = 22; // Port of the local server to connect to.
```

- Compile the sample as follows:

```
# In Linux
# Go to the sample's folder cmake/iothub_client/samples/iothub_client_c2d_streaming_proxy_sample
make -j
```

```
rem In Windows
rem Go to cmake at root of repository
cmake --build . -- /m /p:Configuration=Release
```

- Run the compiled program on the device:

```
# In Linux # Go to sample's folder cmake/iothub_client/samples/iothub_client_c2d_streaming_proxy_sample
./iothub_client_c2d_streaming_proxy_sample
```

```
rem In Windows
rem Go to sample's release folder
cmake\iothub_client\samples\iothub_client_c2d_streaming_proxy_sample\Release
iothub_client_c2d_streaming_proxy_sample.exe
```

Run the service-local proxy application

As discussed [above](#) establishing an end-to-end stream to tunnel SSH traffic requires a local proxy at each end (i.e., service and device). During public preview, IoT Hub C SDK only supports device streams on the device side however. For the service-local proxy, use the accompanying guides in [C# quickstart](#) or [Nodejs quickstart](#) instead.

Establish an SSH session

Assuming that both the device- and service-local proxies are running, now use your SSH client program and connect to service-local proxy on port 2222 (instead of the SSH daemon directly).

```
ssh <username>@localhost -p 2222
```

At this point, you will be presented with the SSH login prompt to enter your credentials.

Console output on the device-local proxy which connects to the SSH daemon at `IP_address:22`:

```
[reza@ubuntu-vm:~/publicpreview/azure-iot-sdk-c/cmake/iothub_client/samples/iothub_client_c2d_streaming_proxy_sample]@ubuntu-vm:~/publicpreview/azure-iot-sdk-c/cmake/iothub_client/samples/iothub_client_c2d_streaming_proxy_sample$ ./iothub_client_c2d_streaming_proxy_sample
-> 00:24:59 CONNECT | VER: 4 | KEEPALIVE: 240 | FLAGS: 192 | USERNAME: IoTHubDeviceStreams.azure-devices.net/MyDevice/?api-version=2017-11-08-preview&DeviceClientType=iothubclient%2f1.2.11%20(native%3b%20Linux%3b%20x86_64) | PWD: XXXX | CLEAN: 0
<- 00:24:59 CONNACK | SESSION_PRESENT: true | RETURN_CODE: 0x0
-> 00:24:59 SUBSCRIBE | PACKET_ID: 2 | TOPIC_NAME: $iothub/streams/POST/# | QOS: 0 | TOPIC_NAME: $iothub/streams/res/# | QOS: 0
<- 00:24:59 SUBACK | PACKET_ID: 2 | RETURN_CODE: 0 | RETURN_CODE: 0
<- 00:25:20 PUBLISH | IS_DUP: false | RETAIN: 0 | QOS: DELIVER_AT_MOST_ONCE = 0x00 | TOPIC_NAME: $iothub/streams/POST/TestStream/?$rid=1&$url=wss%3A%2Feastus2euap.eastus2euap-001.streams.azure-devices.net%3A443%2Fbridges%2FIoTHubDeviceStreams%2FMyDevice%2Fa58c0fa0bb62409aa15518e609062f86&$auth=eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJleHAiOjE1NDcxNjYzODUsImp0aSI6Ij1kb19pODBsRXJEcS1KdlVpT3pENldTN0NsUQ0dGNkMnpTN2JZY1hsVnMiLCJpb3RodWIiOiJJb1RIdWJEZXZpY2VTdHJ1YW1zIn0._f4SzaxHQ-MgRLwsdjw0SA059n8PydKGdFrjhodYk34&$ip=0.0.0.0 | PAYLOAD_LEN: 0
Received stream request (TestStream)
-> 00:25:20 PUBLISH | IS_DUP: false | RETAIN: 0 | QOS: DELIVER_AT_MOST_ONCE | TOPIC_NAME: $iothub/streams/res/200/?$rid=1
Client connected to the streaming gateway (WS_OPEN_OK)
Reporting traffic statistics every 10 seconds.
[2019-01-11 00:25:21 UTC+0000] Network traffic (in bytes) (sent=41; received=41)
[2019-01-11 00:25:31 UTC+0000] Network traffic (in bytes) (sent=1436; received=1512)
[2019-01-11 00:25:41 UTC+0000] Network traffic (in bytes) (sent=4164; received=1288)
```

Console output of the SSH client program (SSH client communicates to SSH daemon by connecting to port 22 where service-local proxy is listening on):

```
[reza@ubuntu-vm:~]
root@RezaYoga:/mnt/c/Users/reza# ssh -p 2222 reza@localhost
reza@localhost's password:
Welcome to Ubuntu 16.04.5 LTS (GNU/Linux 4.15.0-1035-azure x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

 Get cloud support with Ubuntu Advantage Cloud Guest:
 http://www.ubuntu.com/business/services/cloud

43 packages can be updated.
0 updates are security updates.

New release '18.04.1 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

*** System restart required ***
Last login: Fri Jan 11 00:11:34 2019 from 167.220.2.127
@ubuntu-vm:~$
```

Clean up resources

If you will be continuing to the next recommended article, you can keep the resources you've already created and reuse them.

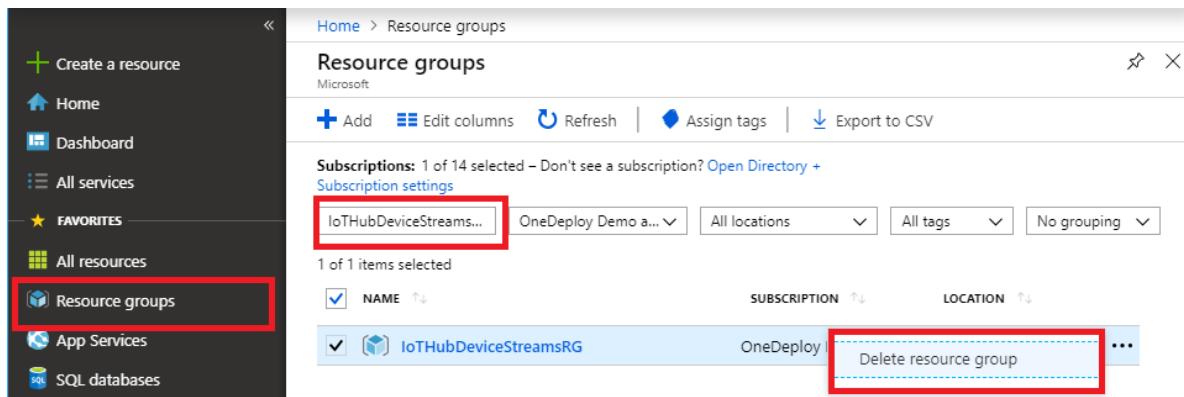
Otherwise, you can delete the Azure resources created in this article to avoid charges.

IMPORTANT

Deleting a resource group is irreversible. The resource group and all the resources contained in it are permanently deleted. Make sure that you do not accidentally delete the wrong resource group or resources. If you created the IoT Hub inside an existing resource group that contains resources you want to keep, only delete the IoT Hub resource itself instead of deleting the resource group.

To delete a resource group by name:

1. Sign in to the [Azure portal](#) and click **Resource groups**.
2. In the **Filter by name...** textbox, type the name of the resource group containing your IoT Hub.
3. To the right of your resource group in the result list, click ... then **Delete resource group**.



The screenshot shows the Azure portal's Resource groups page. On the left, there's a sidebar with links like 'Create a resource', 'Home', 'Dashboard', 'All services', 'FAVORITES', 'All resources', and 'Resource groups'. The 'Resource groups' link is highlighted with a red box. The main area shows a table of resource groups. At the top of the table, there's a 'Filter by name...' input field containing 'IoTHubDeviceStreams...', which is also highlighted with a red box. The table has columns for NAME, SUBSCRIPTION, and LOCATION. A single item is selected: 'IoTHubDeviceStreamsRG' under 'OneDeploy'. To the right of this row, there's a 'Delete resource group' button, which is highlighted with a red box.

4. You will be asked to confirm the deletion of the resource group. Type the name of your resource group again to confirm, and then click **Delete**. After a few moments, the resource group and all of its contained resources are deleted.

Next steps

In this quickstart, you have set up an IoT hub, registered a device, deployed a device- and a service-local proxy program to establish a device stream through IoT Hub, and used the proxies to tunnel SSH traffic.

Use the links below to learn more about device streams:

[Device streams overview](#)

Tutorial: Configure message routing with IoT Hub

3/5/2019 • 23 minutes to read

[Message routing](#) enables sending telemetry data from your IoT devices to built-in Event Hub-compatible endpoints or custom endpoints such as blob storage, Service Bus Queue, Service Bus Topic, and Event Hubs. While configuring message routing, you can create [routing queries](#) to customize the route that matches a certain condition. Once set up, the incoming data is automatically routed to the endpoints by the IoT Hub.

In this tutorial, you learn how to set up and use routing queries with IoT Hub. You will route messages from an IoT device to one of multiple services, including blob storage and a Service Bus queue. Messages to the Service Bus queue will be picked up by a Logic App and sent via e-mail. Messages that do not have routing specifically set up are sent to the default endpoint, and viewed in a Power BI visualization.

In this tutorial, you perform the following tasks:

- Using Azure CLI or PowerShell, set up the base resources -- an IoT hub, a storage account, a Service Bus queue, and a simulated device.
- Configure endpoints and routes in IoT hub for the storage account and Service Bus queue.
- Create a Logic App that is triggered and sends e-mail when a message is added to the Service Bus queue.
- Download and run an app that simulates an IoT Device sending messages to the hub for the different routing options.
- Create a Power BI visualization for data sent to the default endpoint.
- View the results ...
- ...in the Service Bus queue and e-mails.
- ...in the storage account.
- ...in the Power BI visualization.

Prerequisites

NOTE

This article has been updated to use the new Azure PowerShell Az module. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For installation instructions, see [Install Azure PowerShell](#).

- An Azure subscription. If you don't have an Azure subscription, create a [free account](#) before you begin.
- Install [Visual Studio](#).
- A Power BI account to analyze the default endpoint's stream analytics. ([Try Power BI for free](#).)
- An Office 365 account to send notification e-mails.

Open Azure Cloud Shell

Azure Cloud Shell is a free, interactive shell that you can use to run the steps in this article. Common Azure tools are preinstalled and configured in Cloud Shell for you to use with your account. Just select the **Copy** button to copy the code, paste it in Cloud Shell, and then press Enter to run it. There are a few ways to open Cloud Shell:

Select **Try It** in the upper-right corner of a code block.



Open Cloud Shell in your browser.

Launch Cloud Shell

Select the **Cloud Shell** button on the menu in the upper-right corner of the [Azure portal](#).



Set up resources

For this tutorial, you need an IoT hub, a storage account, and a Service Bus queue. These resources can be created using Azure CLI or Azure PowerShell. Use the same resource group and location for all of the resources. Then at the end, you can remove everything in one step by deleting the resource group.

The following sections describe how to do these required steps. Follow the CLI or the PowerShell instructions.

1. Create a [resource group](#).
2. Create an IoT hub in the S1 tier. Add a consumer group to your IoT hub. The consumer group is used by the Azure Stream Analytics when retrieving data.

NOTE

You must use an IoT hub in a paid tier to complete this tutorial. The free tier only allows you to set up one endpoint, and this tutorial requires multiple endpoints.

3. Create a standard V1 storage account with Standard_LRS replication.
4. Create a Service Bus namespace and queue.
5. Create a device identity for the simulated device that sends messages to your hub. Save the key for the testing phase.

Set up your resources using Azure CLI

Copy and paste this script into Cloud Shell. Assuming you are already logged in, it runs the script one line at a time.

The variables that must be globally unique have `$RANDOM` concatenated to them. When the script is run and the variables are set, a random numeric string is generated and concatenated to the end of the fixed string, making it unique.

```
# This is the IOT Extension for Azure CLI.  
# You only need to install this the first time.  
# You need it to create the device identity.  
az extension add --name azure-cli-iot-ext  
  
# Set the values for the resource names that don't have to be globally unique.  
# The resources that have to have unique names are named in the script below  
#   with a random number concatenated to the name so you can probably just  
#   run this script, and it will work with no conflicts.  
location=westus  
resourceGroup=ContosoResources  
iotHubConsumerGroup=ContosoConsumers  
containerName=contosoresults
```

```
iotDeviceName=Contoso-Test-Device

# Create the resource group to be used
#   for all the resources for this tutorial.
az group create --name $resourceGroup \
    --location $location

# The IoT hub name must be globally unique, so add a random number to the end.
iotHubName=ContosoTestHub$RANDOM
echo "IoT hub name = " $iotHubName

# Create the IoT hub.
az iot hub create --name $iotHubName \
    --resource-group $resourceGroup \
    --sku S1 --location $location

# Add a consumer group to the IoT hub.
az iot hub consumer-group create --hub-name $iotHubName \
    --name $iotHubConsumerGroup

# The storage account name must be globally unique, so add a random number to the end.
storageAccountName=contosostorage$RANDOM
echo "Storage account name = " $storageAccountName

# Create the storage account to be used as a routing destination.
az storage account create --name $storageAccountName \
    --resource-group $resourceGroup \
    --location $location \
    --sku Standard_LRS

# Get the primary storage account key.
#   You need this to create the container.
storageAccountKey=$(az storage account keys list \
    --resource-group $resourceGroup \
    --account-name $storageAccountName \
    --query "[0].value" | tr -d "'")

# See the value of the storage account key.
echo "$storageAccountKey"

# Create the container in the storage account.
az storage container create --name $containerName \
    --account-name $storageAccountName \
    --account-key $storageAccountKey \
    --public-access off

# The Service Bus namespace must be globally unique, so add a random number to the end.
sbNameSpace=ContosoSBNamespace$RANDOM
echo "Service Bus namespace = " $sbNameSpace

# Create the Service Bus namespace.
az servicebus namespace create --resource-group $resourceGroup \
    --name $sbNameSpace \
    --location $location

# The Service Bus queue name must be globally unique, so add a random number to the end.
sbQueueName=ContosoSBQueue$RANDOM
echo "Service Bus queue name = " $sbQueueName

# Create the Service Bus queue to be used as a routing destination.
az servicebus queue create --name $sbQueueName \
    --namespace-name $sbNameSpace \
    --resource-group $resourceGroup

# Create the IoT device identity to be used for testing.
az iot hub device-identity create --device-id $iotDeviceName \
    --hub-name $iotHubName

# Retrieve the information about the device identity, then copy the primary key to
```

```
# Notepad. You need this to run the device simulation during the testing phase.  
az iot hub device-identity show --device-id $iotDeviceName \  
--hub-name $iotHubName
```

Set up your resources using Azure PowerShell

Copy and paste this script into Cloud Shell. Assuming you are already logged in, it runs the script one line at a time.

The variables that must be globally unique have `$(Get-Random)` concatenated to them. When the script is run and the variables are set, a random numeric string is generated and concatenated to the end of the fixed string, making it unique.

```
# Log into Azure account.  
Login-AzAccount  
  
# Set the values for the resource names that don't have to be globally unique.  
# The resources that have to have unique names are named in the script below  
#   with a random number concatenated to the name so you can probably just  
#   run this script, and it will work with no conflicts.  
$location = "West US"  
$resourceGroup = "ContosoResources"  
$iotHubConsumerGroup = "ContosoConsumers"  
$containerName = "contosoresults"  
$iotDeviceName = "Contoso-Test-Device"  
  
# Create the resource group to be used  
#   for all resources for this tutorial.  
New-AzResourceGroup -Name $resourceGroup -Location $location  
  
# The IoT hub name must be globally unique, so add a random number to the end.  
$iotHubName = "ContosoTestHub$(Get-Random)"  
Write-Host "IoT hub name is " $iotHubName  
  
# Create the IoT hub.  
New-AzIotHub -ResourceGroupName $resourceGroup `  
    -Name $iotHubName `  
    -SkuName "S1" `  
    -Location $location `  
    -Units 1  
  
# Add a consumer group to the IoT hub for the 'events' endpoint.  
Add-AzIotHubEventHubConsumerGroup -ResourceGroupName $resourceGroup `  
    -Name $iotHubName `  
    -EventHubConsumerGroupName $iotHubConsumerGroup `  
    -EventHubEndpointName "events"  
  
# The storage account name must be globally unique, so add a random number to the end.  
$storageAccountName = "contosostorage$(Get-Random)"  
Write-Host "storage account name is " $storageAccountName  
  
# Create the storage account to be used as a routing destination.  
# Save the context for the storage account  
#   to be used when creating a container.  
$storageAccount = New-AzStorageAccount -ResourceGroupName $resourceGroup `  
    -Name $storageAccountName `  
    -Location $location `  
    -SkuName Standard_LRS `  
    -Kind Storage  
$storageContext = $storageAccount.Context  
  
# Create the container in the storage account.  
New-AzStorageContainer -Name $containerName `  
    -Context $storageContext  
  
# The Service Bus namespace must be globally unique,  
#   with a random number added to the end.
```

```

# so add a random number to the end.
$serviceBusNamespace = "ContosoSBNamespace$(Get-Random)"
Write-Host "Service Bus namespace is " $serviceBusNamespace

# Create the Service Bus namespace.
New-AzServiceBusNamespace -ResourceGroupName $resourceGroup ` 
    -Location $location ` 
    -Name $serviceBusNamespace

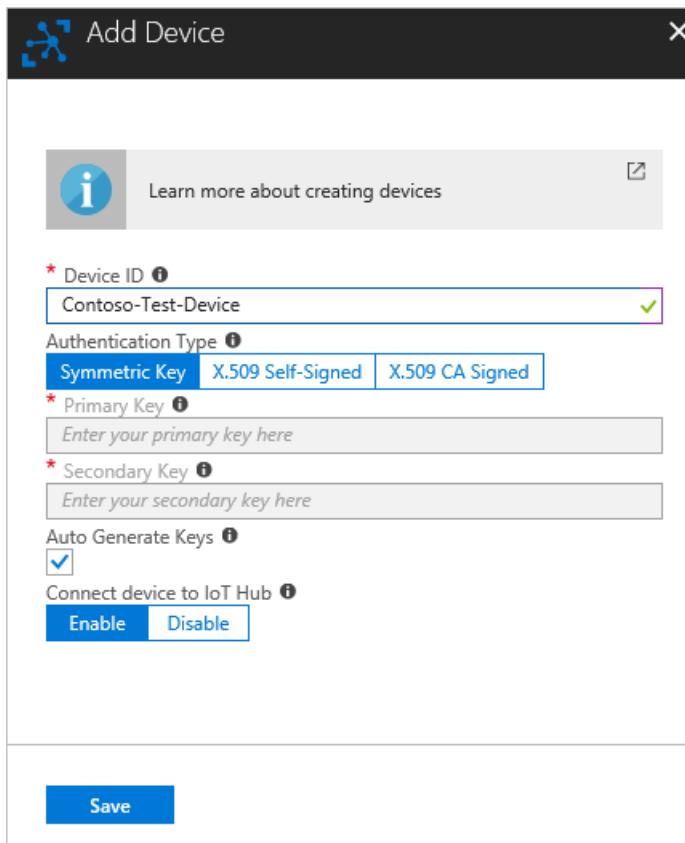
# The Service Bus queue name must be globally unique,
# so add a random number to the end.
$serviceBusQueueName = "ContosoSBQueue$(Get-Random)"
Write-Host "Service Bus queue name is " $serviceBusQueueName

# Create the Service Bus queue to be used as a routing destination.
New-AzServiceBusQueue -ResourceGroupName $resourceGroup ` 
    -Namespace $serviceBusNamespace ` 
    -Name $serviceBusQueueName

```

Next, create a device identity and save its key for later use. This device identity is used by the simulation application to send messages to the IoT hub. This capability is not available in PowerShell, but you can create the device in the [Azure portal](#).

1. Open the [Azure portal](#) and log into your Azure account.
2. Click on **Resource groups** and select your resource group. This tutorial uses **ContosoResources**.
3. In the list of resources, click your IoT hub. This tutorial uses **ContosoTestHub**. Select **IoT Devices** from the Hub pane.
4. Click **+ Add**. On the Add Device pane, fill in the device ID. This tutorial uses **Contoso-Test-Device**. Leave the keys empty, and check **Auto Generate Keys**. Make sure **Connect device to IoT hub** is enabled. Click **Save**.



5. Now that it's been created, click on the device to see the generated keys. Click the Copy icon on the Primary key and save it somewhere such as Notepad for the testing phase of this tutorial.

The screenshot shows the 'Device Details' page for a device named 'Contoso-Test-Device'. It includes fields for 'Device Id' (containing 'Contoso-Test-Device'), 'Primary key' (containing a long hex string), 'Secondary key' (containing another long hex string), 'Connection string—primary key' (containing a connection string for HostName=ContosoTestHub.azure-devices.net), and 'Connection string—secondary key' (containing a connection string for HostName=ContosoTestHub.azure-devices.net). Below these is a section titled 'Connect device to IoT Hub' with a toggle switch currently set to 'Enable'.

Set up message routing

You are going to route messages to different resources based on properties attached to the message by the simulated device. Messages that are not custom routed are sent to the default endpoint (messages/events).

VALUE	RESULT
level="storage"	Write to Azure Storage.
level="critical"	Write to a Service Bus queue. A Logic App retrieves the message from the queue and uses Office 365 to e-mail the message.
default	Display this data using Power BI.

Routing to a storage account

Now set up the routing for the storage account. You go to the Message Routing pane, then add a route. When adding the route, define a new endpoint for the route. After this is set up, messages where the **level** property is set to **storage** are written to a storage account automatically.

The data is written to blob storage in the Avro format by default.

- In the [Azure portal](#), click **Resource Groups**, then select your resource group. This tutorial uses **ContosoResources**.
- Click the IoT hub under the list of resources. This tutorial uses **ContosoTestHub**.
- Click **Message Routing**. In the **Message Routing** pane, click **+Add**. On the **Add a Route** pane, click **+Add** next to the Endpoint field, as displayed in the following picture:

Add a route

* Name

* Endpoint + Add

* Data source ▼

* Enable route ▼

Event hubs

Service bus queue

Service bus topic

Blob storage

Create a query to filter messages before data is routed to an endpoint. [Learn more](#)

Routing query

4. Select **Blob storage**. You see the **Add Storage Endpoint** pane.

Add a storage endpoint

Route your telemetry and device messages to Azure Storage as blobs.

* Endpoint name

Azure Storage account and container

Create a new container, or choose an existing one that shares a subscription with this IoT hub.

Azure Storage container

Batch frequency 100

Chunk size window 100

Encoding AVRO JSON

* Blob file name format

The format must contain {iothub}, {partition}, {YYYY}, {MM}, {DD}, {HH} and {mm} in any order.

If multiple files are created within the same minute, the filename format would be ashitaHub/0/2019/01/29/09/55-01.

5. Enter a name for the endpoint. This tutorial uses **StorageContainer**.

6. Click **Pick a container**. This takes you to a list of your storage accounts. Select the one you set up in the preparation steps. This tutorial uses **contosostorage**. It shows a list of containers in that storage account. Select the container you set up in the preparation steps. This tutorial uses **contosoresults**. Click **Select**. You return to the **Add endpoint** pane.

7. For the purpose of this tutorial, use the defaults for the rest of the fields.

NOTE

You can set the format of the blob name using the **Blob file name format**. The default is

{iothub}/{partition}/{YYYY}/{MM}/{DD}/{HH}/{mm}. The format must contain {iothub}, {partition}, {YYYY}, {MM}, {DD}, {HH}, and {mm} in any order.

For example, using the default blob file name format, if the hub name is ContosoTestHub, and the date/time is October 30, 2018 at 10:56 a.m., the blob name will look like this: ContosoTestHub/0/2018/10/30/10/56.

The blobs are written in the Avro format by default. You can choose to write files in JSON format. The capability to encode JSON format is in preview in all regions IoT Hub is available in, except East US, West US and West Europe. See [guidance on routing to blob storage](#).

When routing to blob storage, we recommend enlisting the blobs and then iterating over them, to ensure all containers are read without making any assumptions of partition. The partition range could potentially change during a [Microsoft-initiated failover](#) or IoT Hub [manual failover](#). To learn how to enumerate the list of blobs see [routing to blob storage](#)

8. Click **Create** to create the storage endpoint and add it to the route. You return to the **Add a route** pane.
9. Now complete the rest of the routing query information. This query specifies the criteria for sending messages to the storage container you just added as an endpoint. Fill in the fields on the screen.

Name: Enter a name for your routing query. This tutorial uses **StorageRoute**.

Endpoint: This shows the endpoint you just set up.

Data source: Select **Device Telemetry Messages** from the dropdown list.

Enable route: Be sure this is enabled.

Routing query: Enter `level="storage"` as the query string.

 Add a storage endpoint

Route your telemetry and device messages to Azure Storage as blobs.

* Endpoint name

Azure Storage account and container

Create a new container, or choose an existing one that shares a subscription with this IoT hub.

Azure Storage container

Pick a container

Batch frequency

Chunk size window

Encoding

* Blob file name format

The format must contain {iothub}, {partition}, {YYYY}, {MM}, {DD}, {HH} and {mm} in any order.

If multiple files are created within the same minute, the filename format would be rk-iot-hub-east/0/2019/01/10/13/30-01.

Click **Save**. When it finishes, it returns to the Message Routing pane, where you can see your new

routing query for storage. Close the Routes pane, which returns you to the Resource group page.

Routing to a Service Bus queue

Now set up the routing for the Service Bus queue. You go to the Message Routing pane, then add a route. When adding the route, define a new endpoint for the route. After this is set up, messages where the **level** property is set to **critical** are written to the Service Bus queue, which triggers a Logic App, which then sends an e-mail with the information.

1. On the Resource group page, click your IoT hub, then click **Message Routing**.
2. In the **Message Routing** pane, click **+ Add**.
3. On the **Add a Route** pane, click **+ Add** next to the Endpoint field. Select **Service Bus Queue**. You see the **Add Service Bus Endpoint** pane.

The screenshot shows the 'Add a service bus endpoint' configuration dialog. It has a title bar with the text 'Add a service bus endpoint'. Below it is a descriptive text: 'Route your telemetry and device messages to Azure Service Bus and add publisher and subscriber capability.' There are three main sections with fields:

- Endpoint name:** A text input field containing 'CriticalQueue'.
- Choose an existing service bus:** A dropdown menu labeled 'Choose an existing service bus'.
- Service bus namespace:** A dropdown menu labeled 'ContosoSBNamespace15129'.
- Service bus queue:** A dropdown menu labeled 'contososbqueue9872'.

At the bottom is a blue 'Create' button.

4. Fill in the fields:

Endpoint Name: Enter a name for the endpoint. This tutorial uses **CriticalQueue**.

Service Bus Namespace: Click on this field to reveal the dropdown list; select the service bus namespace you set up in the preparation steps. This tutorial uses **ContosoSBNamespace**.

Service Bus queue: Click on this field to reveal the dropdown list; select the Service Bus queue from the dropdown list. This tutorial uses **contososbqueue**.

5. Click **Create** to add the Service Bus queue endpoint. You return to the **Add a route** pane.
6. Now you complete the rest of the routing query information. This query specifies the criteria for sending messages to the Service Bus queue you just added as an endpoint. Fill in the fields on the screen.

Name: Enter a name for your routing query. This tutorial uses **SBQueueRoute**.

Endpoint: This shows the endpoint you just set up.

Data source: Select **Device Telemetry Messages** from the dropdown list.

Routing query: Enter `level="critical"` as the query string.

Add a route

* Name [?](#)
SBQueueRoute ✓

* Endpoint [?](#)
CriticalQueue ▼ + Add

* Data source [?](#)
Device Telemetry Messages ▼

* Enable route [?](#)
Enable Disable

Create a query to filter messages before data is routed to an endpoint. [Learn more](#)

Routing query [?](#)
1 level="critical" ✖

- Click **Save**. When it returns to the Routes pane, you see both of your new routes, as displayed here.

Routes Custom endpoints				
Create an endpoint, and then add a route (you can add up to 100 from each IoT hub). Messages that don't match a query are automatically sent to messages/events if you've enabled the fallback route.				
Disable fallback route				
+ Add	Test all routes	Delete		
<input type="checkbox"/>	NAME	DATA SOURCE	ROUTING QUERY	ENDPOINT
	StorageRoute	DeviceMessages	level="storage"	StorageEP
	SBQueueRoute	DeviceMessages	level="critical"	CriticalQueue

- You can see the custom endpoints you set up by clicking on the **Custom Endpoints** tab.

Choose which Azure services will receive your messages. You can add up to 10 endpoints to an IoT hub.

Add **Synchronize keys** **Delete**

- ▽ Event Hubs
- ^ Service Bus queue

Recommended for scenarios that require minimized processing. Messages can be locked or deleted after being read.

<input type="checkbox"/>	NAME	NAMESPACE	QUEUE	STATUS
	CriticalQueue	contososbnames...	contososbqueue...	Unknown
- ▽ Service Bus topic
 - ^ Blob storage

Recommended for storage.

<input type="checkbox"/>	NAME	CONTAINER N...	BATCH FREQU...	FILENAME FOR...	STATUS
	StorageCont...	contosoresults	100	{iothub}://{par...	Unknown

9. Close the Message Routing pane, which returns you to the Resource group pane.

Create a Logic App

The Service Bus queue is to be used for receiving messages designated as critical. Set up a Logic app to monitor the Service Bus queue, and send an e-mail when a message is added to the queue.

1. In the [Azure portal](#), click **+ Create a resource**. Put **logic app** in the search box and click Enter. From the search results displayed, select Logic App, then click **Create** to continue to the **Create logic app** pane. Fill in the fields.

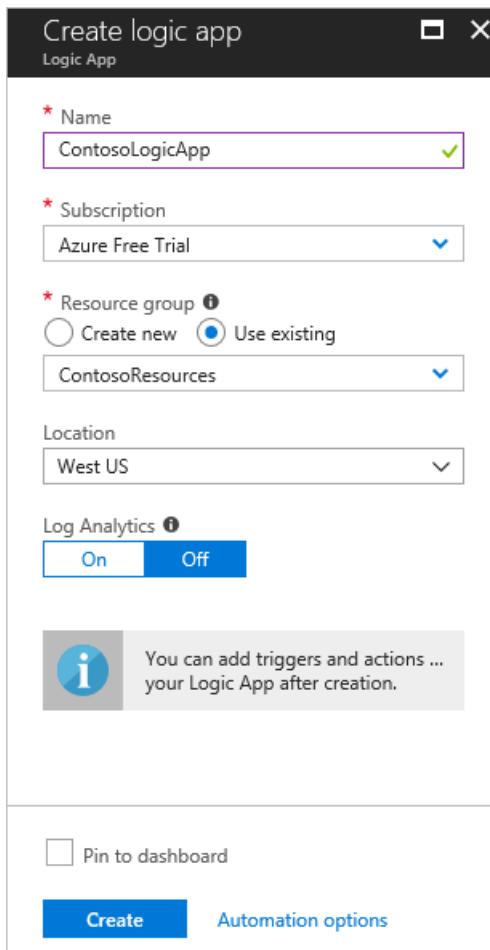
Name: This field is the name of the logic app. This tutorial uses **ContosoLogicApp**.

Subscription: Select your Azure subscription.

Resource group: Click **Use existing** and select your resource group. This tutorial uses **ContosoResources**.

Location: Use your location. This tutorial uses **West US**.

Log Analytics: This toggle should be turned off.



Click **Create**.

2. Now go to the Logic App. The easiest way to get to the Logic App is to click on **Resource groups**, select your resource group (this tutorial uses **ContosoResources**), then select the Logic App from the list of resources. The Logic Apps Designer page appears (you might have to scroll over to the right to see the full page). On the Logic Apps Designer page, scroll down until you see the tile that says **Blank Logic App +** and click it.
3. A list of connectors is displayed. Select **Service Bus**.

The screenshot shows the Microsoft Power Automate interface. At the top, there's a section titled "Connectors" with various icons and names: Request, Schedule, Service Bus (which is highlighted with a red box), Twitter, Office 365 Outlook, SharePoint, FTP, Dynamics 365, SFTP, Salesforce, RSS, OneDrive, Azure Event Grid, and Azure Queues. Below this is a section titled "Triggers (174)" with a list of triggers, each with a preview link and an information icon:

- 10to8 Appointment Scheduling
- Adobe Creative Cloud
- Adobe Creative Cloud
- Appfigures
- Asana
- Azure API Management

4. A list of triggers is displayed. Select **Service Bus - When a message is received in a queue (auto-complete)**.

The screenshot shows the "Search all triggers" screen. It has tabs for "Triggers (8)" and "Actions (22)". The "Triggers (8)" tab is selected. A specific trigger, "Service Bus - When a message is received in a queue (auto-complete)", is highlighted with a red box. Other triggers listed include:

- Service Bus - When a message is received in a queue (peek-lock)
- Service Bus - When a message is received in a topic subscription (auto-complete)
- Service Bus - When a message is received in a topic subscription (peek-lock)
- Service Bus - When one or more messages arrive in a queue (auto-complete)
- Service Bus - When one or more messages arrive in a topic (auto-complete)
- Service Bus - When one or more messages arrive in a queue (peek-lock)
- Service Bus - When one or more messages arrive in a topic (peek-lock)

 At the bottom, there's a "TELL US WHAT YOU NEED" section and a "UserVoice" link.

5. On the next screen, fill in the Connection Name. This tutorial uses **ContosoConnection**.

* Connection Name
ContosoConnection

* Service Bus Namespace

Name	Resource Group	Location
ContosoSBNamespace	ContosoResources	West US

Create Cancel

[Manually enter connection information](#)

Click the Service Bus namespace. This tutorial uses **ContosoSBNamespace**. When you select the namespace, the portal queries the Service Bus namespace to retrieve the keys. Select **RootManageSharedAccessKey** and click **Create**.

* Connection Name
ContosoConnection

* Service Bus Policy

Name	Rights
RootManageSharedAccessKey	Listen, Manage, Send

[Create](#)

[Manually enter connection information](#)

- On the next screen, select the name of the queue (this tutorial uses **contososbqueue**) from the dropdown list. You can use the defaults for the rest of the fields.

* Queue name
contososbqueue

Show advanced options ▾

How often do you want to check for items?

* Interval
3

* Frequency
Minute

Connected to ContosoConnection. [Change connection.](#)

- Now set up the action to send an e-mail when a message is received in the queue. In the Logic Apps Designer, click **+ New step** to add a step, then click **Add an action**. In the **Choose an action** pane, find and click **Office 365 Outlook**. On the triggers screen, select **Office 365 Outlook - Send an email**.

The screenshot shows the Microsoft Flow interface. At the top, a trigger is listed: "When a message is received in a queue (auto-complete)". An arrow points down to the "Actions (34)" section. Within this section, several actions are listed under the heading "Triggers (9) Actions (34)". One specific action, "Office 365 Outlook - Send an email", is highlighted with a red box.

8. Next, log into your Office 365 account to set up the connection. Specify the e-mail addresses for the recipient(s) of the e-mails. Also specify the subject, and type what message you'd like the recipient to see in the body. For testing, fill in your own e-mail address as the recipient.

Click **Add dynamic content** to show the content from the message that you can include. Select **Content** -- it will include the message in the e-mail.

The screenshot shows the Microsoft Flow designer with the "Send an email" step selected. The "To" field is set to <your-email-address@domain>. The "Subject" field is set to "Critical message from IoT Hub!". The "Body" field contains the message: "This is a critical message from an IoT device connected to your IoT Hub." Below the body, there is a "Content" button and an "Add dynamic content" button. To the right, a sidebar titled "Add dynamic content from the apps and connectors used in this flow" lists several options: Content, Content Type, Lock Token, and Session Id. The "Content" option is currently selected.

9. Click **Save**. Then close the Logic App Designer.

Set up Azure Stream Analytics

To see the data in a Power BI visualization, first set up a Stream Analytics job to retrieve the data. Remember that only the messages where the **level** is **normal** are sent to the default endpoint, and will be retrieved by the Stream Analytics job for the Power BI visualization.

Create the Stream Analytics job

1. In the [Azure portal](#), click **Create a resource** > **Internet of Things** > **Stream Analytics job**.

2. Enter the following information for the job.

Job name: The name of the job. The name must be globally unique. This tutorial uses **contosoJob**.

Resource group: Use the same resource group used by your IoT hub. This tutorial uses **ContosoResources**.

Location: Use the same location used in the setup script. This tutorial uses **West US**.

The screenshot shows the 'New Stream Analytics job' configuration interface. It includes fields for Job name (contosoJob), Subscription (Azure Free Trial), Resource group (ContosoResources), Location (West US), Hosting environment (Cloud selected), Streaming units (a slider with a value of 1), and a 'Pin to dashboard' checkbox. At the bottom are 'Create' and 'Automation options' buttons.

Job name	contosoJob
Subscription	Azure Free Trial
Resource group	<input type="radio"/> Create new <input checked="" type="radio"/> Use existing ContosoResources
Location	West US
Hosting environment	<input type="radio"/> Cloud <input checked="" type="radio"/> Edge
Streaming units	1
<input type="checkbox"/> Pin to dashboard	
Create Automation options	

3. Click **Create** to create the job. To get back to the job, click **Resource groups**. This tutorial uses **ContosoResources**. Select the resource group, then click the Stream Analytics job in the list of resources.

Add an input to the Stream Analytics job

1. Under **Job Topology**, click **Inputs**.

2. In the **Inputs** pane, click **Add stream input** and select IoT Hub. On the screen that comes up, fill in the following fields:

Input alias: This tutorial uses **contosoinputs**.

Subscription: Select your subscription.

IoT Hub: Select the IoT Hub. This tutorial uses **ContosoTestHub**.

Endpoint: Select **Messaging**. (If you select Operations Monitoring, you get the telemetry data about the IoT hub rather than the data you're sending through.)

Shared access policy name: Select **iothubowner**. The portal fills in the Shared Access Policy Key for you.

Consumer group: Select the consumer group you created earlier. This tutorial uses **contosoconsumers**.

For the rest of the fields, accept the defaults.

The screenshot shows the 'IoT Hub' configuration dialog for a new input. The 'Input alias' is set to 'contosoinputs'. The 'Provide IoT Hub settings manually' option is unselected, and 'Select IoT Hub from your subscriptions' is selected, with 'ContosoTestHub' chosen. The 'Subscription' is 'Azure Free Trial'. The 'Endpoint' is 'Messaging'. The 'Shared access policy name' is 'iothubowner'. The 'Shared access policy key' field contains a redacted key. The 'Consumer group' is 'contosoconsumers'. The 'Event serialization format' is 'JSON'. The 'Encoding' is 'UTF-8'. The 'Event compression type' is 'None'. At the bottom is a blue 'Save' button.

3. Click **Save**.

Add an output to the Stream Analytics job

1. Under **Job Topology**, click **Outputs**.
2. In the **Outputs** pane, click **Add**, and then select **Power BI**. On the screen that comes up, fill in the following fields:

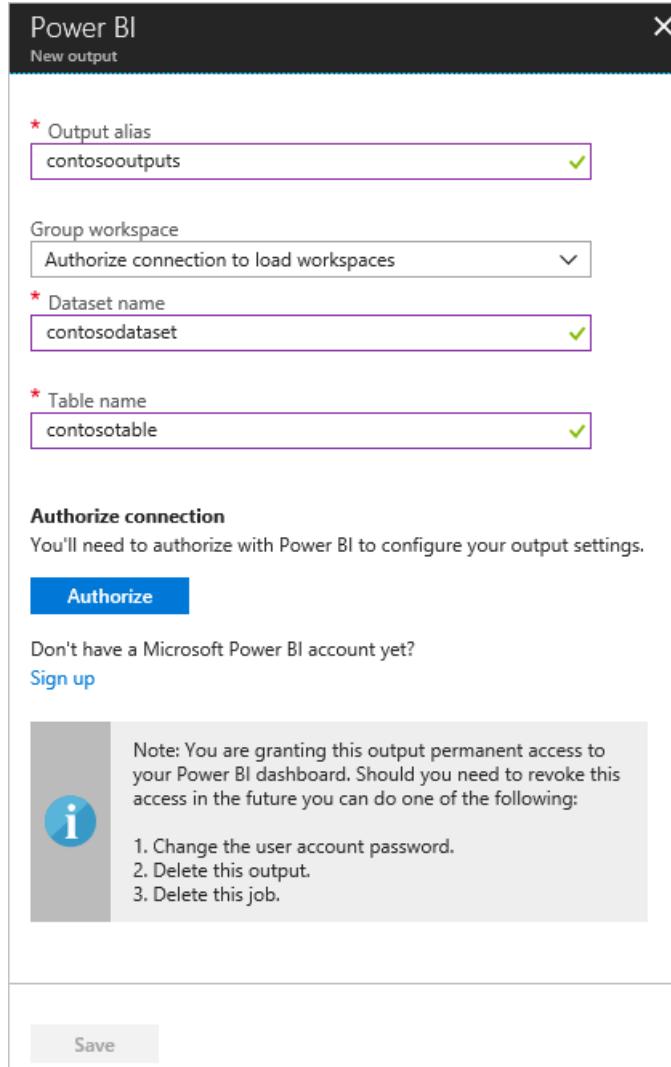
Output alias: The unique alias for the output. This tutorial uses **contosooutputs**.

Dataset name: Name of the dataset to be used in Power BI. This tutorial uses **contosodataset**.

Table name: Name of the table to be used in Power BI. This tutorial uses **contosatable**.

Accept the defaults for the rest of the fields.

3. Click **Authorize**, and sign into your Power BI account.



4. Click **Save**.

Configure the query of the Stream Analytics job

1. Under **Job Topology**, click **Query**.
2. Replace **[YourInputAlias]** with the input alias of the job. This tutorial uses **contosoinputs**.
3. Replace **[YourOutputAlias]** with the output alias of the job. This tutorial uses **contosooutputs**.

Need help with your query? Check out some of the most common Stream Analytics query patterns [here](#).

```
1 SELECT
2 *
3 INTO
4 contosooutputs
5 FROM
6 contosoinputs
```

Your query could be put in logs that are in a potentially different geography.
Missing some language constructs? [Let us know!](#) (Powered by [UserVoice - Privacy Policy](#))

4. Click **Save**.
5. Close the Query pane. This returns you to the view of the resources in the Resource Group. Click the Stream Analytics job. This tutorial calls it **contosoJob**.

Run the Stream Analytics job

In the Stream Analytics job, click **Start > Now > Start**. Once the job successfully starts, the job status changes from **Stopped** to **Running**.

To set up the Power BI report, you need data, so you'll set up Power BI after creating the device and running the device simulation application.

Run Simulated Device app

Earlier in the script setup section, you set up a device to simulate using an IoT device. In this section, you download a .NET console app that simulates a device that sends device-to-cloud messages to an IoT hub. This application sends messages for each of the different routing methods.

Download the solution for the [IoT Device Simulation](#). This downloads a repo with several applications in it; the solution you are looking for is in `iot-hub/Tutorials/Routing/SimulatedDevice/`.

Double-click on the solution file (`SimulatedDevice.sln`) to open the code in Visual Studio, then open `Program.cs`. Substitute `{iot hub hostname}` with the IoT hub host name. The format of the IoT hub host name is **{iot-hub-name}.azure-devices.net**. For this tutorial, the hub host name is **ContosoTestHub.azure-devices.net**. Next, substitute `{device key}` with the device key you saved earlier when setting up the simulated device.

```
static string myDeviceId = "contoso-test-device";
static string iotHubUri = "ContosoTestHub.azure-devices.net";
// This is the primary key for the device. This is in the portal.
// Find your IoT hub in the portal > IoT devices > select your device > copy the key.
static string deviceKey = "{your device key here}";
```

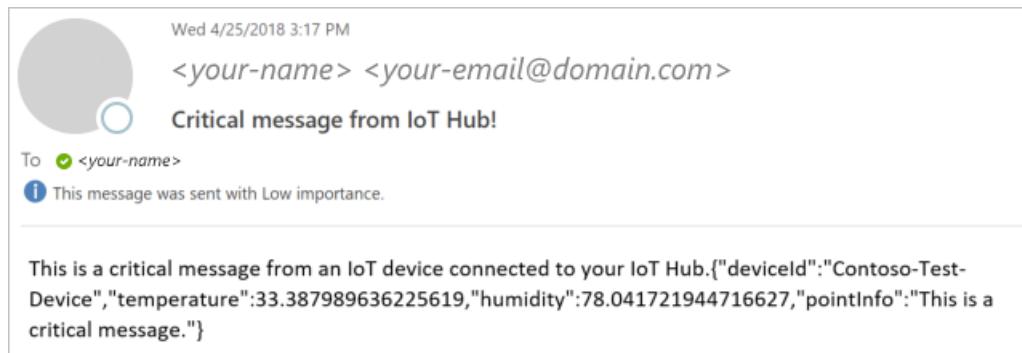
Run and test

Run the console application. Wait a few minutes. You can see the messages being sent on the console screen of the application.

The app sends a new device-to-cloud message to the IoT hub every second. The message contains a JSON-serialized object with the device ID, temperature, humidity, and message level, which defaults to `normal`. It randomly assigns a level of `critical` or `storage`, causing the message to be routed to the storage account or to the Service Bus queue (which triggers your Logic App to send an e-mail). The default (`normal`) readings will be displayed in the BI report you set up next.

If everything is set up correctly, at this point you should see the following results:

1. You start getting e-mails about critical messages.



This means the following:

- The routing to the Service Bus queue is working correctly.
 - The Logic App retrieving the message from the Service Bus queue is working correctly.
 - The Logic App connector to Outlook is working correctly.
2. In the [Azure portal](#), click **Resource groups** and select your Resource Group. This tutorial uses **ContosoResources**. Select the storage account, click **Blobs**, then select the Container. This tutorial uses **contosoresults**. You should see a folder, and you can drill down through the directories until you see one or more files. Open one of those files; they contain the entries routed to the storage account.

The screenshot shows the Azure Storage Blobs interface for the 'contosoresults' container. The left sidebar has 'Overview' selected under 'CONTAINER'. The main area shows a list of blobs. The table has columns: NAME, MODIFIED, BLOB T..., and SIZE. The data is as follows:

NAME	MODIFIED	BLOB T...	SIZE
[..]			
55	4/25/2018 1:57:1...	Block...	6.9 Kib
57	4/25/2018 1:59:1...	Block...	7.72 Kib

This means the following:

- The routing to the storage account is working correctly.

Now with the application still running, set up the Power BI visualization to see the messages coming through the default routing.

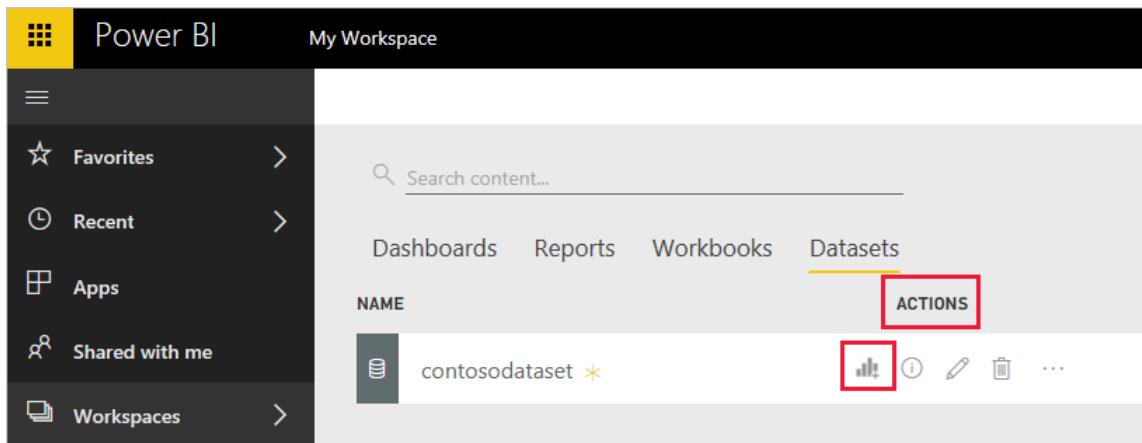
Set up the Power BI Visualizations

1. Sign in to your [Power BI](#) account.
2. Go to **Workspaces** and select the workspace that you set when you created the output for the Stream Analytics job. This tutorial uses **My Workspace**.
3. Click **Datasets**.

You should see the listed dataset that you specified when you created the output for the Stream Analytics job. This tutorial uses **contosodataset**. (It may take 5-10 minutes for the dataset to show up

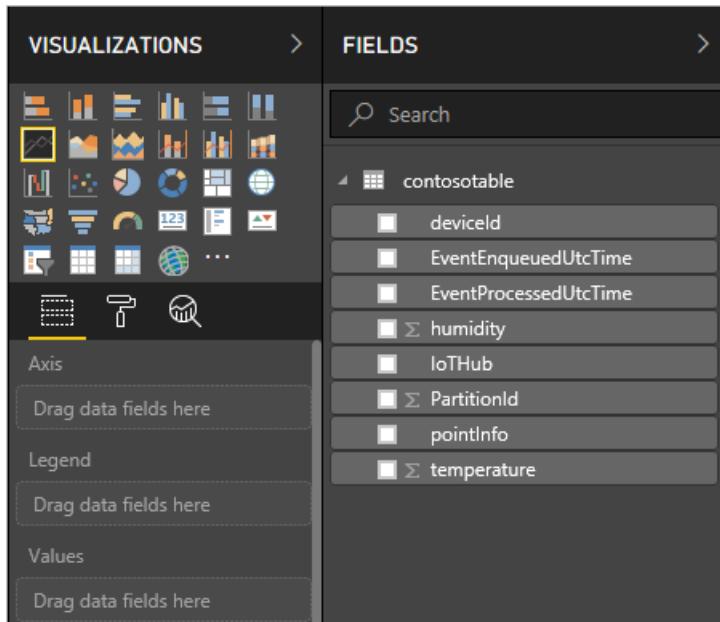
the first time.)

4. Under **ACTIONS**, click the first icon to create a report.



5. Create a line chart to show real-time temperature over time.

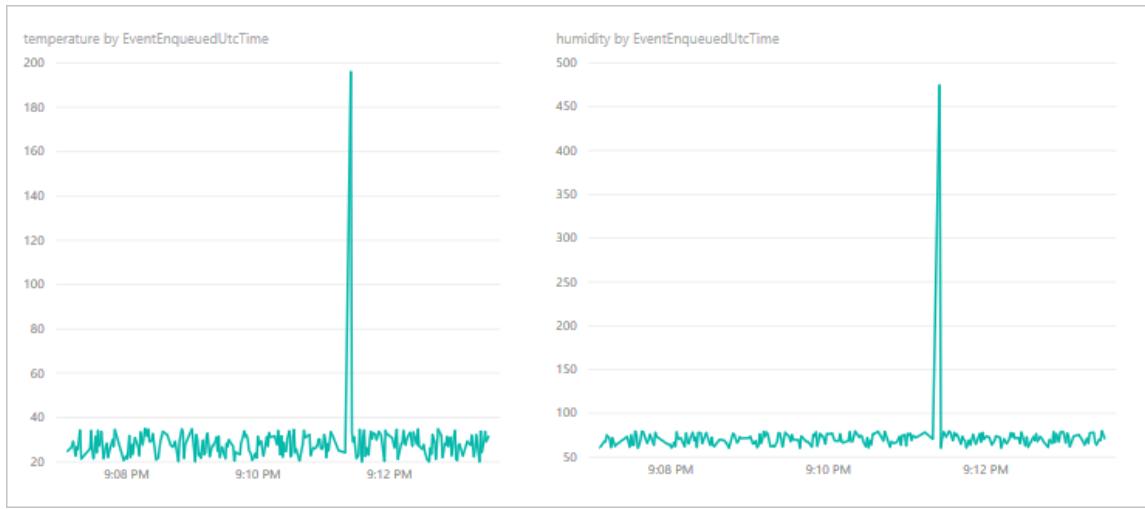
- On the report creation page, add a line chart by clicking the line chart icon.



- On the **Fields** pane, expand the table that you specified when you created the output for the Stream Analytics job. This tutorial uses **contosatable**.
- Drag **EventEnqueuedUtcTime** to **Axis** on the **Visualizations** pane.
- Drag **temperature** to **Values**.

A line chart is created. The x-axis displays date and time in the UTC time zone. The y-axis displays temperature from the sensor.

6. Create another line chart to show real-time humidity over time. To set up the second chart, follow the same steps above and place **EventEnqueuedUtcTime** on the x-axis and **humidity** on the y-axis.



- Click **Save** to save the report.

You should be able to see data on both charts. This means the following:

- The routing to the default endpoint is working correctly.
- The Azure Stream Analytics job is streaming correctly.
- The Power BI Visualization is set up correctly.

You can refresh the charts to see the most recent data by clicking the Refresh button on the top of the Power BI window.

Clean up resources

If you want to remove all of the resources you've created, delete the resource group. This action deletes all resources contained within the group. In this case, it removes the IoT hub, the Service Bus namespace and queue, the Logic App, the storage account, and the resource group itself.

Clean up resources in the Power BI visualization

Log into your [Power BI](#) account. Go to your workspace. This tutorial uses **My Workspace**. To remove the Power BI visualization, go to DataSets and click the trash can icon to delete the dataset. This tutorial uses **contosodataset**. When you remove the dataset, the report is removed as well.

Clean up resources using Azure CLI

To remove the resource group, use the [az group delete](#) command.

```
az group delete --name $resourceGroup
```

Clean up resources using PowerShell

To remove the resource group, use the [Remove-AzResourceGroup](#) command. \$resourceGroup was set to **ContosoloTRG1** back at the beginning of this tutorial.

```
Remove-AzResourceGroup -Name $resourceGroup
```

Next steps

In this tutorial, you learned how to use message routing to route IoT Hub messages to different destinations by performing the following tasks.

- Using Azure CLI or PowerShell, set up the base resources -- an IoT hub, a storage account, a Service Bus

queue, and a simulated device.

- Configure endpoints and routes in IoT hub for the storage account and Service Bus queue.
- Create a Logic App that is triggered and sends e-mail when a message is added to the Service Bus queue.
- Download and run an app that simulates an IoT Device sending messages to the hub for the different routing options.
- Create a Power BI visualization for data sent to the default endpoint.
- View the results ...
- ...in the Service Bus queue and e-mails.
- ...in the storage account.
- ...in the Power BI visualization.

Advance to the next tutorial to learn how to manage the state of an IoT device.

[Set up and use metrics and diagnostics with an IoT Hub](#)

Tutorial: Set up and use metrics and diagnostic logs with an IoT hub

3/14/2019 • 12 minutes to read

If you have an IoT Hub solution running in production, you want to set up some metrics and enable diagnostic logs. Then if a problem occurs, you have data to look at that will help you diagnose the problem and fix it more quickly. In this article, you'll see how to enable the diagnostic logs, and how to check them for errors. You'll also set up some metrics to watch, and alerts that fire when the metrics hit a certain boundary. For example, you could have an e-mail sent to you when the number of telemetry messages sent exceeds a specific boundary, or when the number of messages used gets close to the quota of messages allowed per day for the IoT Hub.

An example use case is a gas station where the pumps are IoT devices that send communicate with an IoT hub. Credit cards are validated, and the final transaction is written to a data store. If the IoT devices stop connecting to the hub and sending messages, it is much more difficult to fix if you have no visibility into what's going on.

This tutorial uses the Azure sample from the [IoT Hub Routing](#) to send messages to the IoT hub.

In this tutorial, you perform the following tasks:

- Using Azure CLI, create an IoT hub, a simulated device, and a storage account.
- Enable diagnostic logs.
- Enable metrics.
- Set up alerts for those metrics.
- Download and run an app that simulates an IoT device sending messages to the hub.
- Run the app until the alerts begin to fire.
- View the metrics results and check the diagnostic logs.

Prerequisites

- An Azure subscription. If you don't have an Azure subscription, create a [free account](#) before you begin.
- Install [Visual Studio](#).
- An email account capable of receiving mail.

Open Azure Cloud Shell

Azure Cloud Shell is a free, interactive shell that you can use to run the steps in this article. Common Azure tools are preinstalled and configured in Cloud Shell for you to use with your account. Just select the **Copy** button to copy the code, paste it in Cloud Shell, and then press Enter to run it. There are a few ways to open Cloud Shell:

Select Try It in the upper-right corner of a code block.	
Open Cloud Shell in your browser.	
Select the Cloud Shell button on the menu in the upper-right corner of the Azure portal .	

Set up resources

For this tutorial, you need an IoT hub, a storage account, and a simulated IoT device. These resources can be created using Azure CLI or Azure PowerShell. Use the same resource group and location for all of the resources. Then at the end, you can remove everything in one step by deleting the resource group.

These are the required steps.

1. Create a [resource group](#).
2. Create an IoT hub.
3. Create a standard V1 storage account with Standard_LRS replication.
4. Create a device identity for the simulated device that sends messages to your hub. Save the key for the testing phase.

Set up resources using Azure CLI

Copy and paste this script into Cloud Shell. Assuming you are already logged in, it runs the script one line at a time. The new resources are created in the resource group ContosoResources.

The variables that must be globally unique have `$RANDOM` concatenated to them. When the script is run and the variables are set, a random numeric string is generated and concatenated to the end of the fixed string, making it unique.

```

# This is the IOT Extension for Azure CLI.
# You only need to install this the first time.
# You need it to create the device identity.
az extension add --name azure-cli-iot-ext

# Set the values for the resource names that don't have to be globally unique.
# The resources that have to have unique names are named in the script below
# with a random number concatenated to the name so you can probably just
# run this script, and it will work with no conflicts.
location=westus
resourceGroup=ContosoResources
iotDeviceName=Contoso-Test-Device

# Create the resource group to be used
# for all the resources for this tutorial.
az group create --name $resourceGroup \
    --location $location

# The IoT hub name must be globally unique, so add a random number to the end.
iotHubName=ContosoTestHub$RANDOM
echo "IoT hub name = " $iotHubName

# Create the IoT hub in the Free tier.
az iot hub create --name $iotHubName \
    --resource-group $resourceGroup \
    --sku F1 --location $location

# The storage account name must be globally unique, so add a random number to the end.
storageAccountName=contosostoragemon$RANDOM
echo "Storage account name = " $storageAccountName

# Create the storage account.
az storage account create --name $storageAccountName \
    --resource-group $resourceGroup \
    --location $location \
    --sku Standard_LRS

# Create the IoT device identity to be used for testing.
az iot hub device-identity create --device-id $iotDeviceName \
    --hub-name $iotHubName

# Retrieve the information about the device identity, then copy the primary key to
# Notepad. You need this to run the device simulation during the testing phase.
az iot hub device-identity show --device-id $iotDeviceName \
    --hub-name $iotHubName

```

NOTE

When creating the device identity, you may get the following error: *No keys found for policy iothubowner of IoT Hub ContosoTestHub*. To fix this error, update the Azure CLI IoT Extension and then run the last two commands in the script again.

Here is the command to update the extension. Run this in your Cloud Shell instance.

```
az extension update --name azure-cli-iot-ext
```

Enable the diagnostic logs

Diagnostic logs are disabled by default when you create a new IoT hub. In this section, enable the diagnostic logs for your hub.

1. First, if you're not already on your hub in the portal, click **Resource groups** and click on the resource group Contoso-Resources. Select the hub from the list of resources displayed.
2. Look for the **Monitoring** section in the IoT Hub blade. Click **Diagnostic settings**.

The screenshot shows the Azure IoT Hub blade for 'ContosoTestHub'. The left sidebar has sections like 'Explorers', 'Automatic Device Management', 'Messaging', 'Resiliency', and 'Monitoring'. Under 'Monitoring', there are four items: 'Alerts', 'Metrics', 'Diagnostic settings' (which is highlighted with a red box), and 'Logs'.

3. Make sure the subscription and resource group are correct. Under **Resource Type**, uncheck **Select All**, then look for and check **IoT Hub**. (It puts the checkmark next to *Select All* again, just ignore it.) Under **Resource**, select the hub name. Your screen should look like this image:

The screenshot shows the 'Turn on diagnostics' configuration pane. It includes fields for 'Subscription' (Microsoft Azure Internal Consumption), 'Resource group' (ContosoResources), 'Resource type' (IoT Hub), and 'Resource' (ContosoTestHub). Below these fields, a link says 'Microsoft Azure Internal Consumption > ContosoResources > ContosoTestHub'. At the bottom, there's a button labeled 'Turn on diagnostics to collect the following data.'

4. Now click **Turn on diagnostics**. The Diagnostics settings pane is displayed. Specify the name of your diagnostic logs settings as "diags-hub".
5. Check **Archive to a storage account**.

Diagnostics settings

Save Discard Delete

* Name
diags-hub

Archive to a storage account

Storage account Configure >

Stream to an event hub

Send to Log Analytics

Click **Configure** to see the **Select a storage account** screen, select the right one (*contosostoragemon*), and click **OK** to return to the Diagnostics settings pane.

Diagnostics settings

Save Discard Delete

i You'll be charged normal data rates for storage and transactions when you send diagnostics to a storage account.

* Name
diags-hub

Archive to a storage account

Storage account contosostoragemon >

Stream to an event hub

Send to Log Analytics

6. Under **LOG**, check **Connections** and **Device Telemetry**, and set the **Retention (days)** to 7 days for each.
Your Diagnostic settings screen should now look like this image:

Diagnostics settings

Save **Discard** **Delete**

i You'll be charged normal data rates for storage and transactions when you send diagnostics to a storage account.

* Name: **diags-hub**

Archive to a storage account

Storage account: **contosostoragemon**

Stream to an event hub

Send to Log Analytics

LOG

<input checked="" type="checkbox"/> Connections	Retention (days) <input type="range" value="7"/> 7
<input checked="" type="checkbox"/> DeviceTelemetry	Retention (days) <input type="range" value="7"/> 7
Retention (days) <input type="range"/>	

7. Click **Save** to save the settings. Close the Diagnostics settings pane.

Later, when you look at the diagnostic logs, you'll be able to see the connect and disconnect logging for the device.

Set up metrics

Now set up some metrics to watch for when messages are sent to the hub.

1. In the settings pane for the IoT hub, click on the **Metrics** option in the **Monitoring** section.
2. At the top of the screen, click **Last 24 hours (Automatic)**. In the dropdown that appears, select **Last 4 hours** for **Time Range**, and set **Time Granularity** to **1 minute**, local time. Click **Apply** to save these settings.

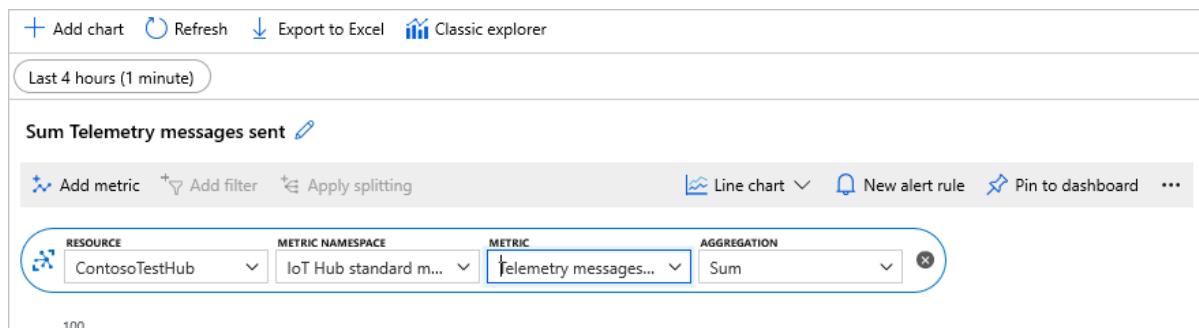
Add chart **Refresh** **Export to Excel** **Classic exp**

Last 24 hours (Automatic)

Time range	Time granularity	
<input type="radio"/> Last 30 minutes	<input type="radio"/> Last 48 hours	<input type="radio"/> 1 minute
<input type="radio"/> Last hour	<input type="radio"/> Last 3 days	<input type="radio"/> UTC/GMT
<input checked="" type="radio"/> Last 4 hours	<input type="radio"/> Last 7 days	<input checked="" type="radio"/> Local
<input type="radio"/> Last 12 hours	<input type="radio"/> Last 30 days	
<input type="radio"/> Last 24 hours	<input type="radio"/> Custom	

Apply **Cancel**

3. There is one metric entry by default. Leave the resource group as the default, and the metric namespace. In the **Metric** dropdown list, select **Telemetry messages sent**. Set **Aggregation** to **Sum**.



- Now click **Add metric** to add another metric to the chart. Select your resource group (**ContosoTestHub**). Under **Metric**, select **Total number of messages used**. For **Aggregation**, select **Avg**.

Now your screen shows the minimized metric for *Telemetry messages sent*, plus the new metric for *Total number of messages used*.



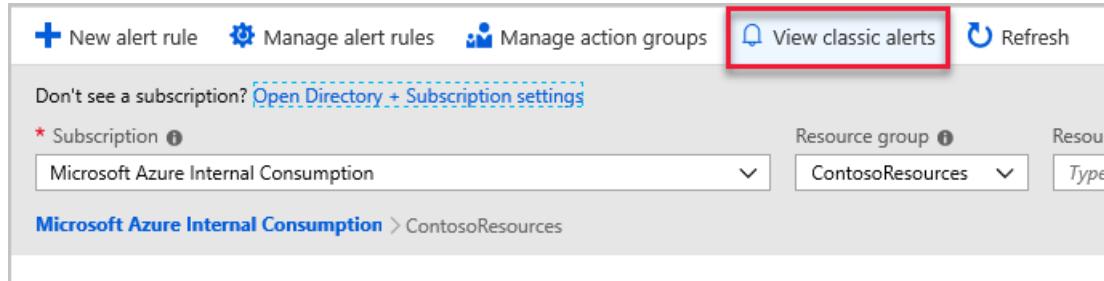
Click **Pin to dashboard**. It will pin it to the dashboard of your Azure portal so you can access it again. If you don't pin it to the dashboard, your settings are not retained.

Set up alerts

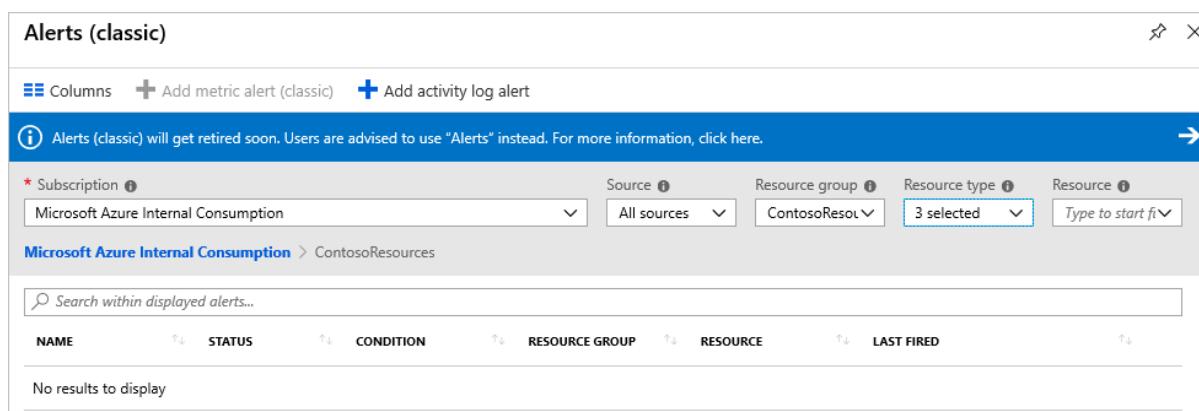
Go to the hub in the portal. Click **Resource Groups**, select *ContosoResources*, then select IoT Hub *ContosoTestHub*.

IoT Hub has not been migrated to the [metrics in Azure Monitor](#) yet; you have to use [classic alerts](#).

- Under **Monitoring**, click **Alerts**. This shows the main alert screen.



- To get to the classic alerts from here, click **View classic alerts**.



Fill in the fields:

Subscription: Leave this field set to your current subscription.

Source: Set this field to *Metrics*.

Resource group: Set this field to your current resource group, *ContosoResources*.

Resource type: Set this field to IoT Hub.

Resource: Select your IoT hub, *ContosoTestHub*.

3. Click **Add metric alert (classic)** to set up a new alert.

Fill in the fields:

Name: Provide a name for your alert rule, such as *telemetry-messages*.

Description: Provide a description of your alert, such as *alert when there are 1000 telemetry messages sent*.

Source: Set this to *Metrics*.

Subscription, Resource group, and Resource should be set to the values you selected on the **View classic alerts** screen.

Set **Metric** to *Telemetry messages sent*.

Add rule

* Name	telemetry-messages
Description	alert when there are 1000 telemetry messages sent
Source	Alert on Metrics
Criteria	Subscription Microsoft Azure Internal Consumption
Resource group	ContosoResources
Resource	ContosoTestHub
* Metric	Telemetry messages sent

4. After the chart, set the following fields:

Condition: Set to *Greater than*.

Threshold: Set to 1000.

Period: Set to *Over the last 5 minutes*.

Notification email recipients: Put your e-mail address here.

Condition: Greater than

* Threshold: 1000

Period: Over the last 5 minutes

Notify via:

Email all users who hold owner, contributor or reader roles for the subscription

Notification email recipients: myemail@mydomain.com

Webhook: HTTP or HTTPS endpoint to route alerts to

Take action: Run a logic app from this alert

OK

Click **OK** to save the alert.

- Now set up another alert for the *Total number of messages used*. This metric is useful if you want to send an alert when the number of messages used is approaching the quota for the IoT hub -- to let you know the hub will soon start rejecting messages.

On the **View classic alerts** screen, click **Add metric alert (classic)**, then fill in these fields on the **Add rule** pane.

Name: Provide a name for your alert rule, such as *number-of-messages-used*.

Description: Provide a description of your alert, such as *alert when getting close to quota*.

Source: Set this field to *Metrics*.

Subscription, Resource group, and Resource should be set to the values you selected on the **View classic alerts** screen.

Set **Metric** to *Total number of messages used*.

- Under the chart, fill in the following fields:

Condition: Set to *Greater than*.

Threshold: Set to 1000.

Period: Set this field to *Over the last 5 minutes*.

Notification email recipients: Put your e-mail address here.

Click **OK** to save the rule.

- You should now see two alerts in the classic alerts pane:

Alerts (classic)

Columns Add metric alert (classic) Add activity log alert

Alerts (classic) will get retired soon. Users are advised to use "Alerts" instead. For more information, click here.

* Subscription Microsoft Azure Internal Consumption Source Metrics Resource group ContosoReso Resource type IoT Hub Resource ContosoTestH

Microsoft Azure Internal Consumption > ContosoResources > ContosoTestHub

Diagnostics settings

Displaying 1 - 2 rules out of total 2 rules

NAME	STATUS	CONDITION	RESOURCE GROUP	RESOURCE	LAST FIRED
number-messages-us...	Active	Total number of mess...	ContosoResources	ContosoTestHub	Never
telemetry-messages...	Active	Telemetry messages s...	ContosoResources	ContosoTestHub	Never

8. Close the alerts pane.

With these settings, you will get an alert when the number of messages sent is greater than 400 and when the total number of messages used exceeds NUMBER.

Run Simulated Device app

Earlier in the script setup section, you set up a device to simulate using an IoT device. In this section, you download a .NET console app that simulates a device that sends device-to-cloud messages to an IoT hub.

Download the solution for the [IoT Device Simulation](#). This link downloads a repo with several applications in it; the solution you are looking for is in `iot-hub/Tutorials/Routing/`.

Double-click on the solution file (`SimulatedDevice.sln`) to open the code in Visual Studio, then open `Program.cs`. Substitute `{iot hub hostname}` with the IoT hub host name. The format of the IoT hub host name is **{iot-hub-name}.azure-devices.net**. For this tutorial, the hub host name is **ContosoTestHub.azure-devices.net**. Next, substitute `{device key}` with the device key you saved earlier when setting up the simulated device.

```
static string myDeviceId = "contoso-test-device";
static string iotHubUri = "ContosoTestHub.azure-devices.net";
// This is the primary key for the device. This is in the portal.
// Find your IoT hub in the portal > IoT devices > select your device > copy the key.
static string deviceKey = "{your device key here}";
```

Run and test

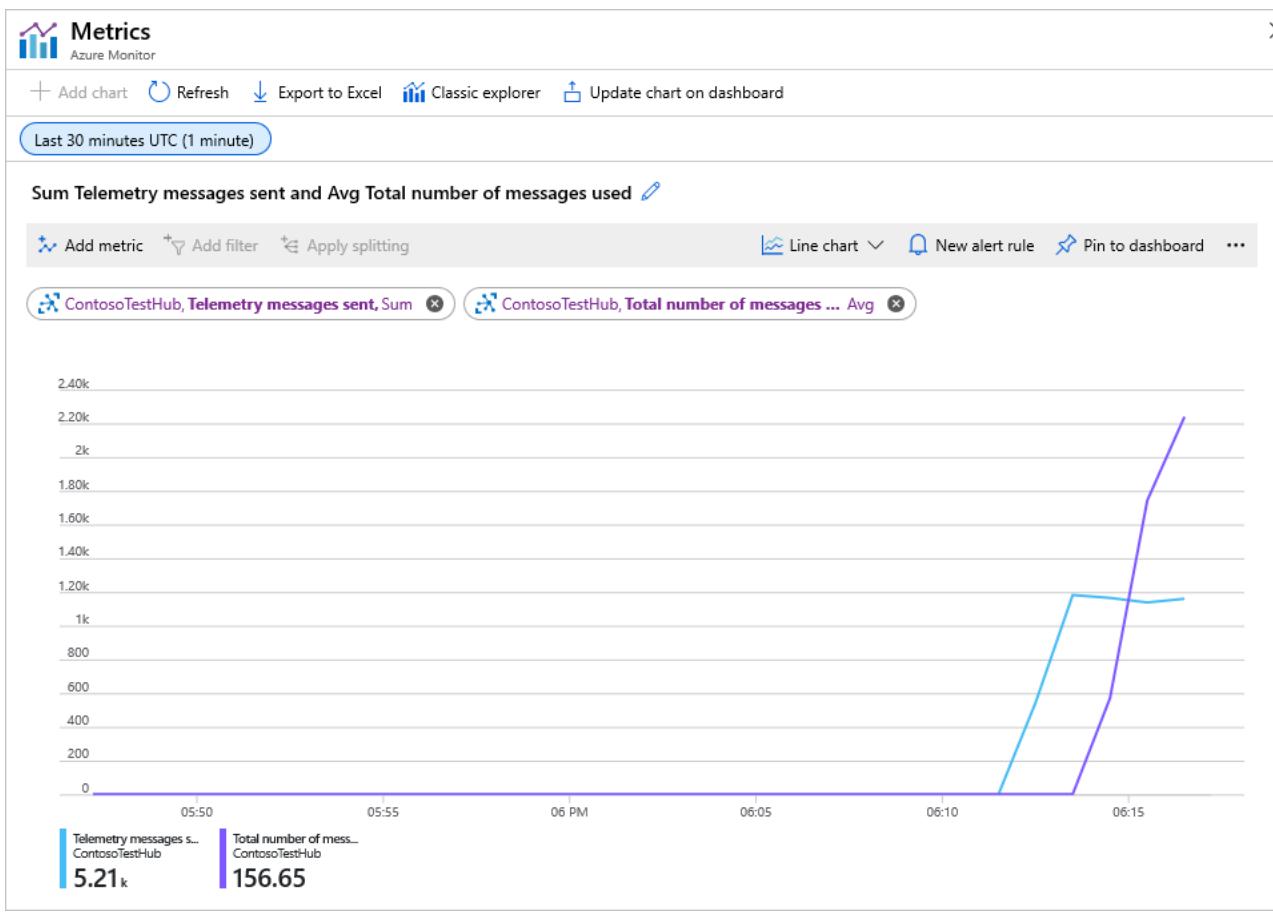
In `Program.cs`, change the `Task.Delay` from 1000 to 10, which reduces the amount of time between sending messages from 1 second to .01 seconds. Shortening this delay increases the number of messages sent.

```
await Task.Delay(10);
```

Run the console application. Wait a few minutes (10-15). You can see the messages being sent from the simulated device to the hub on the console screen of the application.

See the metrics in the portal

Open your metrics from the Dashboard. Change the time values to *Last 30 minutes* with a time granularity of *1 minute*. It shows the telemetry messages sent and the total number of messages used on the chart, with the most recent numbers at the bottom of the chart.



See the alerts

Go back to alerts. Click **Resource groups**, select *ContosoResources*, then select the hub *ContosoTestHub*. In the properties page displayed for the hub, select **Alerts**, then **View classic alerts**.

When the number of messages sent exceeds the limit, you start getting e-mail alerts. To see if there are any active alerts, go to your hub and select **Alerts**. It will show you the alerts that are active, and if there are any warnings.

The screenshot shows the 'Alerts (classic)' blade. It lists two rules:

- number-messages-used-alert**: Status: Warning, Condition: Total number of messages use... (warning threshold), Resource Group: ContosoResources, Resource: ContosoTestHub, Last Fired: 7 min ago.
- telemetry-messages**: Status: Warning, Condition: Telemetry messages sent > 10... (warning threshold), Resource Group: ContosoResources, Resource: ContosoTestHub, Last Fired: 7 min ago.

Click on the alert for telemetry messages. It shows the metric result and a chart with the results. Also, the e-mail sent to warn you of the alert firing looks like this image:

Dear Customer,

**⚠ 'd2c.telemetry.ingress.success
GreaterThan 1000 (Count) in the last 5
minutes' was activated for
contosotesthub**

You can view more details for this alert in the [Microsoft Azure Management Portal](#).

RULE NAME: telemetry-messages

RULE DESCRIPTION: alert when there are 1000 telemetry messages sent

SERVICE: IoTHubs: ContosoTestHub (ContosoResources)

METRIC: Total d2c.telemetry.ingress.success

ALERT ACTIVATED TIME (UTC): 12/17/2018 6:20:25 PM

See the diagnostic logs

You set up your diagnostic logs to be exported to blob storage. Go to your resource group and select your storage account *contosostoragemon*. Select Blobs, then open container *insights-logs-connections*. Drill down until you get to the current date and select the most recent file.

« X		resourceId=/SUBSCRIPTIONS/<Your subscription id> Blob																										
Upload	Refresh	... More																										
Location: insights-logs-connections / resourceId= / SUBSCRIPTIONS / <Your Subscription ID> / RESOURCEGROUPS / CONTOSORESOURCES / PROVIDERS / MICROSOFT.DEVICES / IOTHUBS / CONTOSOTESTHUB / y=2018 / m=12 / d=17 / h=18 / m=00	Save	Discard																										
<input type="text"/> Search blobs by prefix (case-sensitive)	<input type="checkbox"/> Show deleted blobs	<input type="button" value="Download"/> <input type="button" value="Acquire lease"/> <input type="button" value="... More"/>																										
NAME																												
[..]	...	<input type="button" value="Overview"/> <input type="button" value="Snapshots"/> <input type="button" value="Edit blob"/> <input type="button" value="Generate SAS"/>																										
PTIH.json	...	<input type="button" value="Properties"/> <table><tr><td>URL</td><td>https://contosostoragemon.blob.core.windows.net/insights-logs-connections/2018/12/17/PTIH.json </td></tr><tr><td>LAST MODIFIED</td><td>12/17/2018, 10:23:16 AM</td></tr><tr><td>CREATION TIME</td><td>12/17/2018, 10:12:28 AM</td></tr><tr><td>TYPE</td><td>Append blob</td></tr><tr><td>SIZE</td><td>9.03 KiB</td></tr><tr><td>SERVER ENCRYPTED</td><td>true</td></tr><tr><td>ETAG</td><td>0x8D6644CB6895CF8</td></tr><tr><td>CONTENT-MD5</td><td>-</td></tr><tr><td>LEASE STATUS</td><td>Unlocked</td></tr><tr><td>LEASE STATE</td><td>Available</td></tr><tr><td>LEASE DURATION</td><td>-</td></tr><tr><td>COPY STATUS</td><td>-</td></tr><tr><td>COPY COMPLETION TIME</td><td>-</td></tr></table> <input type="button" value="Undelete all snapshots"/>	URL	https://contosostoragemon.blob.core.windows.net/insights-logs-connections/2018/12/17/PTIH.json	LAST MODIFIED	12/17/2018, 10:23:16 AM	CREATION TIME	12/17/2018, 10:12:28 AM	TYPE	Append blob	SIZE	9.03 KiB	SERVER ENCRYPTED	true	ETAG	0x8D6644CB6895CF8	CONTENT-MD5	-	LEASE STATUS	Unlocked	LEASE STATE	Available	LEASE DURATION	-	COPY STATUS	-	COPY COMPLETION TIME	-
URL	https://contosostoragemon.blob.core.windows.net/insights-logs-connections/2018/12/17/PTIH.json																											
LAST MODIFIED	12/17/2018, 10:23:16 AM																											
CREATION TIME	12/17/2018, 10:12:28 AM																											
TYPE	Append blob																											
SIZE	9.03 KiB																											
SERVER ENCRYPTED	true																											
ETAG	0x8D6644CB6895CF8																											
CONTENT-MD5	-																											
LEASE STATUS	Unlocked																											
LEASE STATE	Available																											
LEASE DURATION	-																											
COPY STATUS	-																											
COPY COMPLETION TIME	-																											

Click **Download** to download it and open it. You see the logs of the device connecting and disconnecting as it

sends messages to the hub. Here a sample:

```
{  
    "time": "2018-12-17T18:11:25Z",  
    "resourceId":  
        "/SUBSCRIPTIONS/your-subscription-  
id/RESOURCEGROUPS/CONTOSORESOURCES/PROVIDERS/MICROSOFT.DEVICES/IOTHUBS/CONTOSOTESTHUB",  
    "operationName": "deviceConnect",  
    "category": "Connections",  
    "level": "Information",  
    "properties":  
        {"deviceId": "Contoso-Test-Device",  
            "protocol": "Mqtt",  
            "authType": null,  
            "maskedIpAddress": "73.162.215.XXX",  
            "statusCode": null  
        },  
    "location": "westus"  
}  
{  
    "time": "2018-12-17T18:19:25Z",  
    "resourceId":  
        "/SUBSCRIPTIONS/your-subscription-  
id/RESOURCEGROUPS/CONTOSORESOURCES/PROVIDERS/MICROSOFT.DEVICES/IOTHUBS/CONTOSOTESTHUB",  
    "operationName": "deviceDisconnect",  
    "category": "Connections",  
    "level": "Error",  
    "resultType": "404104",  
    "resultDescription": "DeviceConnectionClosedRemotely",  
    "properties":  
        {"deviceId": "Contoso-Test-Device",  
            "protocol": "Mqtt",  
            "authType": null,  
            "maskedIpAddress": "73.162.215.XXX",  
            "statusCode": "404"  
        },  
    "location": "westus"  
}
```

Clean up resources

To remove all of the resources you've created in this tutorial, delete the resource group. This action deletes all resources contained within the group. In this case, it removes the IoT hub, the storage account, and the resource group itself. If you have pinned metrics to the dashboard, you will have to remove those manually by clicking on the three dots in the upper right-hand corner of each and selecting **Remove**.

To remove the resource group, use the [az group delete](#) command.

```
az group delete --name $resourceGroup
```

Next steps

In this tutorial, you learned how to use metrics and diagnostic logs by performing the following tasks:

- Using Azure CLI, create an IoT hub, a simulated device, and a storage account.
- Enable diagnostic logs.
- Enable metrics.
- Set up alerts for those metrics.
- Download and run an app that simulates an IoT device sending messages to the hub.

- Run the app until the alerts begin to fire.
- View the metrics results and check the diagnostic logs.

Advance to the next tutorial to learn how to manage the state of an IoT device.

[Configure your devices from a back-end service](#)

Tutorial: Perform manual failover for an IoT hub (public preview)

2/6/2019 • 5 minutes to read

Manual failover is a feature of the IoT Hub service that allows customers to [failover](#) their hub's operations from a primary region to the corresponding Azure geo-paired region. Manual failover can be done in the event of a regional disaster or an extended service outage. You can also perform a planned failover to test your disaster recovery capabilities, although we recommend using a test IoT hub rather than one running in production. The manual failover feature is offered to customers at no additional cost.

In this tutorial, you perform the following tasks:

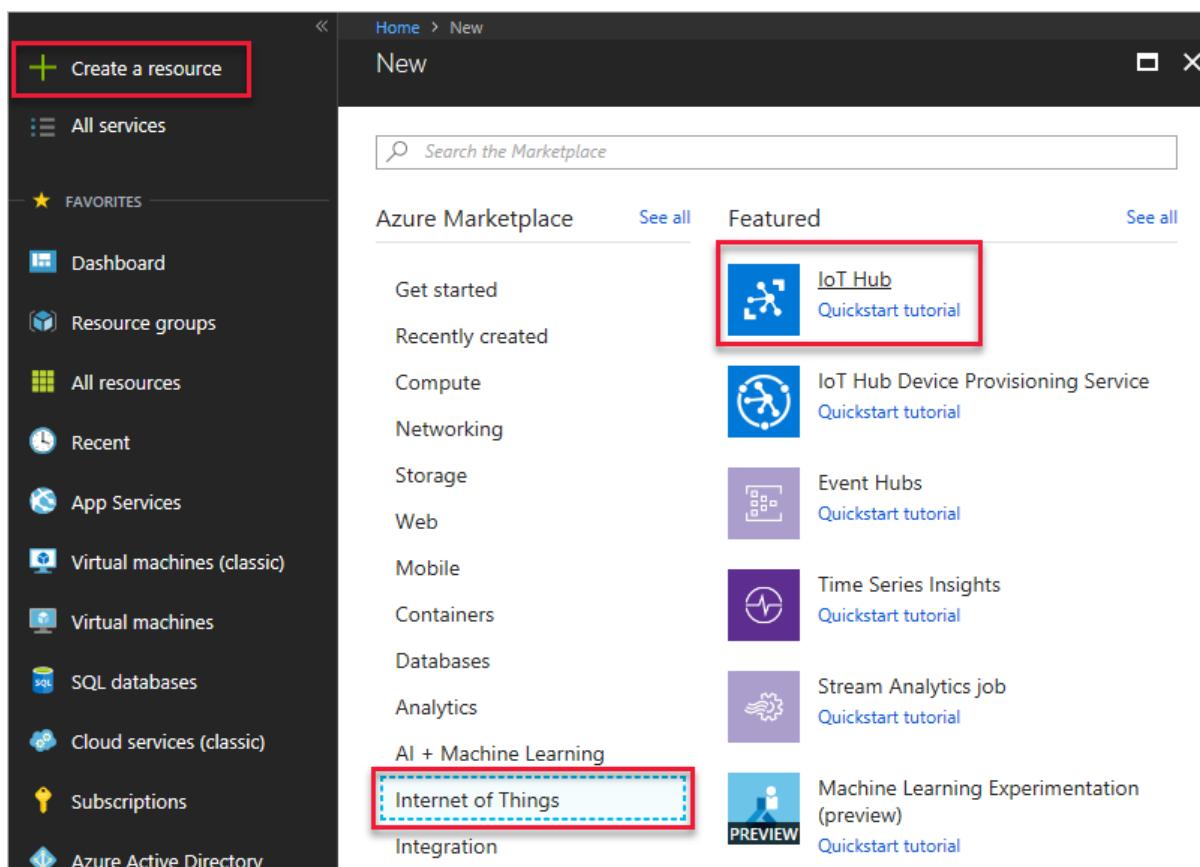
- Using the Azure portal, create an IoT hub.
- Perform a failover.
- See the hub running in the secondary location.
- Perform a fallback to return the IoT hub's operations to the primary location.
- Confirm the hub is running correctly in the right location.

Prerequisites

- An Azure subscription. If you don't have an Azure subscription, create a [free account](#) before you begin.

Create an IoT hub

1. Log into the [Azure portal](#).
2. Click **+ Create a resource** and select **Internet of Things**, then **IoT Hub**.



3. Select the **Basics** tab. Fill in the following fields.

Subscription: select the Azure subscription you want to use.

Resource Group: click **Create new** and specify **ManFailRG** for the resource group name.

Region: select a region close to you that is part of the preview. This tutorial uses **westus2**. A failover can only be performed between Azure geo-paired regions. The region geo-paired with westus2 is WestCentralUS.

NOTE

Manual failover is currently in public preview and is *not* available in the following Azure regions: East US, West US, North Europe, West Europe, Brazil South, and South Central US.

IoT Hub Name: specify a name for your IoT hub. The hub name must be globally unique.

The screenshot shows the 'Basics' tab of the IoT Hub creation wizard. It includes fields for Subscription (set to Azure Free Trial), Resource Group (set to Create new, ManFailRG), Region (set to West US 2), and IoT Hub Name (set to ContosoHubTestFailover2). The 'Review + create' button at the bottom left is highlighted with a red border.

Click **Review + create**. (It uses the defaults for size and scale.)

4. Review the information, then click **Create** to create the IoT hub.

IoT hub

Microsoft

Basics Size and scale Review + create

BASICS

Subscription	DEVICEHUB_DEV1
Resource Group	ManFailRG
Region	West US 2
IoT Hub Name	ContosoHubTestFailover

SIZE AND SCALE

Pricing and scale tier	S1
Number of S1 IoT Hub units	1
Messages per day	400,000
Cost per month	25.00 USD

Create [« Previous: Size and scale](#) [Automation options](#)

Perform a manual failover

Note that there is a limit of two failovers and two fallbacks per day for an IoT hub.

1. Click **Resource groups** and then select the resource group **ManFailRG**. Click on your hub in the list of resources.
2. Under **Resiliency** on the IoT Hub pane, click **Manual failover (preview)**. Note that if your hub is not set up in a valid region, the manual failover option will be disabled.

ContosoHubTestFailover
IoT Hub

Search (Ctrl+/)

IoT Edge (preview)

IoT device configuration (pre...)

MESSAGING

File upload

Endpoints

Routes

RESILIENCY

Manual failover (preview)

MONITORING

Metrics

Metrics (preview)

3. On the Manual failover pane, you see the **IoT Hub Primary Location** and the **IoT Hub Secondary**

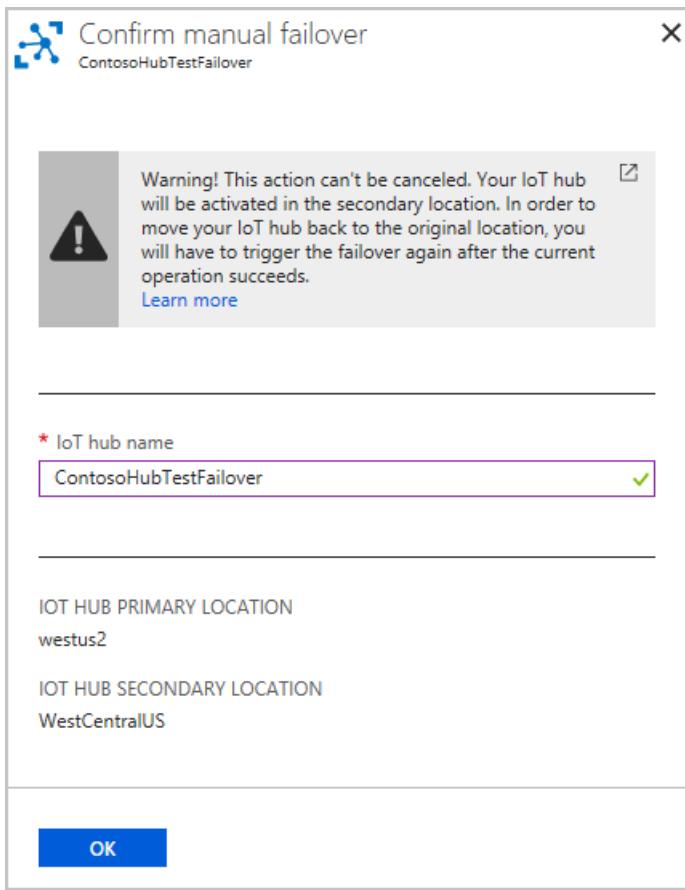
Location. The primary location is initially set to the location you specified when you created the IoT hub, and always indicates the location in which the hub is currently active. The secondary location is the standard [Azure geo-paired region](#) that is paired to the primary location. You cannot change the location values. For this tutorial, the primary location is `westus2` and the secondary location is `WestCentralUS`.

The screenshot shows the 'Manual failover' pane. At the top, there's a button labeled 'Initiate failover'. Below it is a section titled 'Manual failover' with a green cloud icon. A callout box contains an information icon and text: 'Use this feature to failover your IoT hub to the secondary location. This action will cause down time and telemetry loss to your solution. This is a long running operation and could take several minutes to finish. Please exercise with caution when using it.' It also includes a link 'Learn more about manual failover' and a checkbox. Further down, there are two sections: 'IOT HUB PRIMARY LOCATION' with a dropdown containing 'westus2' and a refresh icon, and 'IOT HUB SECONDARY LOCATION' with a dropdown containing 'WestCentralUS' and a refresh icon.

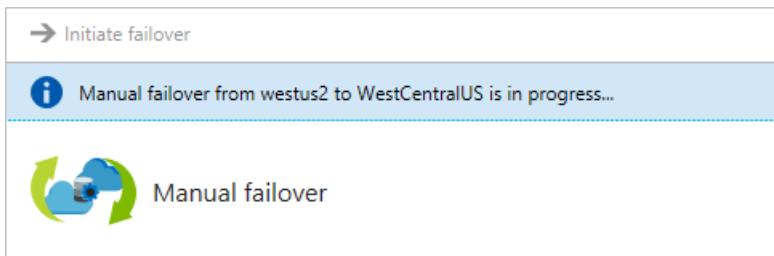
4. At the top of the Manual failover pane, click **Initiate failover**. You see the **Confirm manual failover** pane. Fill in the name of your IoT hub to confirm it's the one you want to failover. Then, to initiate the failover, click **OK**.

The amount of time it takes to perform the manual failover is proportional to the number of devices that are registered for your hub. For example, if you have 100,000 devices, it might take 15 minutes, but if you have five million devices, it might take an hour or longer.

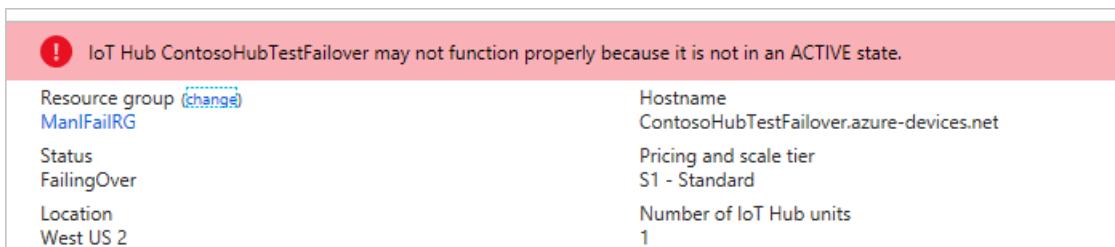
5. In the **Confirm manual failover** pane, fill in the name of your IoT hub to confirm it's the one you want to failover. Then, to initiate the failover, click **OK**.



While the manual failover process is running, there is a banner on the Manual Failover pane that tells you a manual failover is in progress.



If you close the IoT Hub pane and open it again by clicking it on the Resource Group pane, you see a banner that tells you the hub is not active.



After it's finished, the primary and secondary regions on the Manual Failover page are flipped and the hub is active again. In this example, the primary location is now "WestCentralUS" and the secondary location is now "westus2".

→ Initiate failover

 Manual failover

 Use this feature to failover your IoT hub to the secondary location. This action will cause down time and telemetry loss to your solution. This is a long running operation and could take several minutes to finish. Please exercise with caution when using it.

[Learn more about manual failover](#)

IOT HUB PRIMARY LOCATION  WestCentralUS 

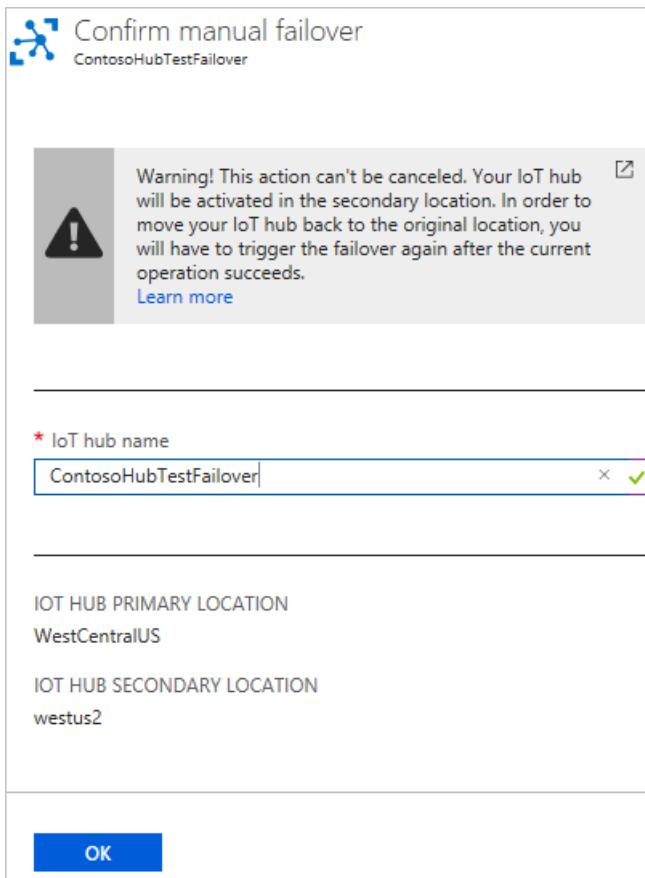
IOT HUB SECONDARY LOCATION  westus2 

Perform a failback

After you have performed a manual failover, you can switch the hub's operations back to the original primary region -- this is called a failback. If you have just performed a failover, you have to wait about an hour before you can request a failback. If you try to perform the failback in a shorter amount of time, an error message is displayed.

A failback is performed just like a manual failover. These are the steps:

1. To perform a failback, return to the IoT Hub pane for your IoT hub.
2. Under **Resiliency** on the IoT Hub pane, click **Manual failover (preview)**.
3. At the top of the Manual failover pane, click **Initiate failover**. You see the **Confirm manual failover** pane.
4. In the **Confirm manual failover** pane, fill in the name of your IoT hub to confirm it's the one you want to failback. To then initiate the failback, click **OK**.



The banners are displayed as explained in the perform a failover section. After the fallback is complete, it again shows `westus2` as the primary location and `WestCentralUS` as the secondary location, as set originally.

Clean up resources

To remove the resources you've created for this tutorial, delete the resource group. This action deletes all resources contained within the group. In this case, it removes the IoT hub and the resource group itself.

1. Click **Resource Groups**.
2. Locate and select the resource group **ManFailRG**. Click on it to open it.
3. Click **Delete resource group**. When prompted, enter the name of the resource group and click **Delete** to confirm.

Next steps

In this tutorial, you learned how to configure and perform a manual failover, and how to request a fallback by performing the following tasks:

- Using the Azure portal, create an IoT hub.
- Perform a failover.
- See the hub running in the secondary location.
- Perform a fallback to return the IoT hub's operations to the primary location.
- Confirm the hub is running correctly in the right location.

Advance to the next tutorial to learn how to manage the state of an IoT device.

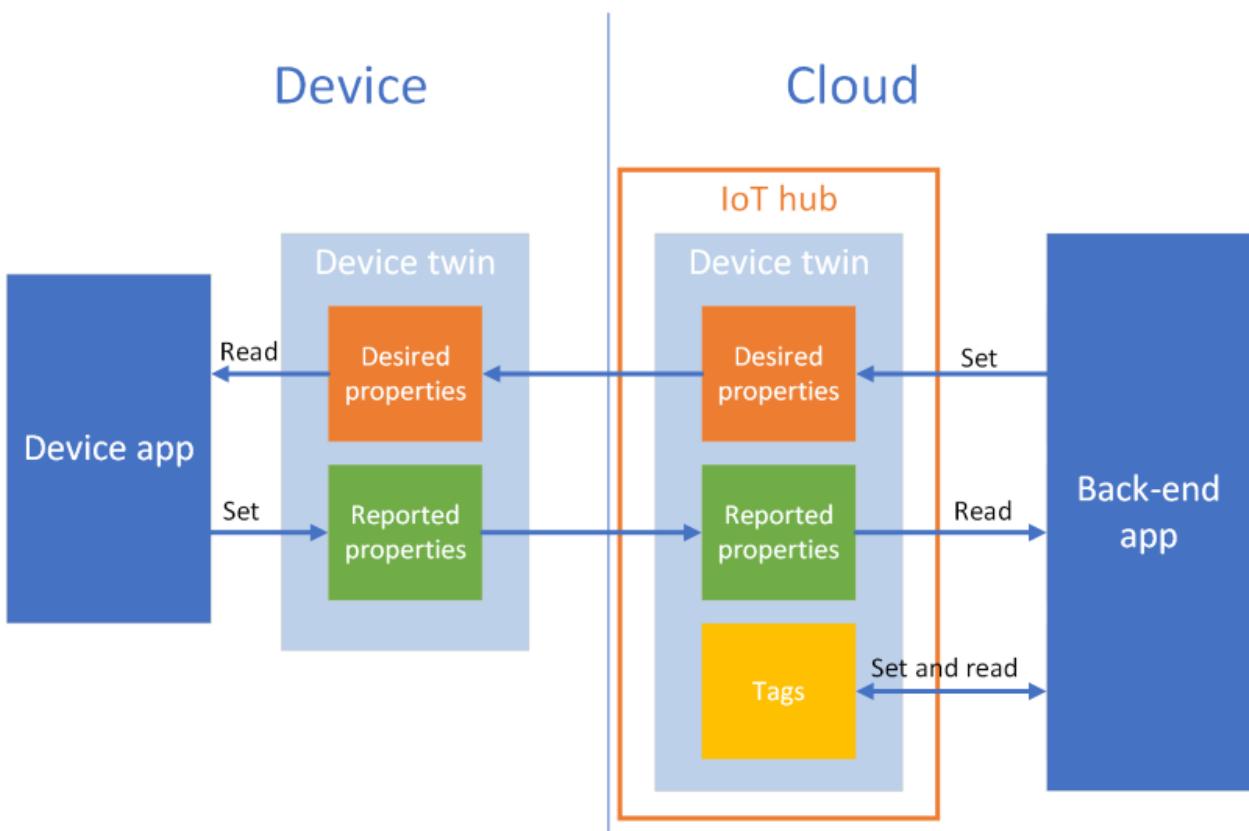
[Manage the state of an IoT device](#)

Tutorial: Configure your devices from a back-end service

3/5/2019 • 12 minutes to read

As well as receiving telemetry from your devices, you may need to configure your devices from your back-end service. When you send a desired configuration to your devices, you may also want to receive status and compliance updates from those devices. For example, you might set a target operational temperature range for a device or collect firmware version information from your devices.

To synchronize state information between a device and an IoT hub, you use *device twins*. A [device twin](#) is a JSON document, associated with a specific device, and stored by IoT Hub in the cloud where you can [query](#) them. A device twin contains *desired properties*, *reported properties*, and *tags*. A desired property is set by a back-end application and read by a device. A reported property is set by a device and read by a back-end application. A tag is set by a back-end application and is never sent to a device. You use tags to organize your devices. This tutorial shows you how to use desired and reported properties to synchronize state information:



In this tutorial, you perform the following tasks:

- Create an IoT hub and add a test device to the identity registry.
- Use desired properties to send state information to your simulated device.
- Use reported properties to receive state information from your simulated device.

Open Azure Cloud Shell

Azure Cloud Shell is a free, interactive shell that you can use to run the steps in this article. Common Azure tools are preinstalled and configured in Cloud Shell for you to use with your account. Just select the **Copy** button to copy the code, paste it in Cloud Shell, and then press Enter to run it. There are a few ways to open Cloud Shell:

Select **Try It** in the upper-right corner of a code block.



Open Cloud Shell in your browser.



Select the **Cloud Shell** button on the menu in the upper-right corner of the [Azure portal](#).



If you don't have an Azure subscription, create a [free account](#) before you begin.

Prerequisites

The two sample applications you run in this quickstart are written using Node.js. You need Node.js v4.x.x or later on your development machine.

You can download Node.js for multiple platforms from nodejs.org.

You can verify the current version of Node.js on your development machine using the following command:

```
node --version
```

Download the sample Node.js project from <https://github.com/Azure-Samples/azure-iot-samples-node/archive/master.zip> and extract the ZIP archive.

Set up Azure resources

To complete this tutorial, your Azure subscription must contain an IoT hub with a device added to the device identity registry. The entry in the device identity registry enables the simulated device you run in this tutorial to connect to your hub.

If you don't already have an IoT hub set up in your subscription, you can set one up with the following CLI script. This script uses the name **tutorial-iot-hub** for the IoT hub, you should replace this name with your own unique name when you run it. The script creates the resource group and hub in the **Central US** region, which you can change to a region closer to you. The script retrieves your IoT hub service connection string, which you use in the back-end sample to connect to your IoT hub:

```
hubname=tutorial-iot-hub
location=centralus

# Install the IoT extension if it's not already installed:
az extension add --name azure-cli-iot-ext

# Create a resource group:
az group create --name tutorial-iot-hub-rg --location $location

# Create your free-tier IoT Hub. You can only have one free IoT Hub per subscription:
az iot hub create --name $hubname --location $location --resource-group tutorial-iot-hub-rg --sku F1

# Make a note of the service connection string, you need it later:
az iot hub show-connection-string --hub-name $hubname -o table
```

This tutorial uses a simulated device called **MyTwinDevice**. The following script adds this device to your identity registry and retrieves its connection string:

```

# Set the name of your IoT hub:
hubname=tutorial-iot-hub

# Create the device in the identity registry:
az iot hub device-identity create --device-id MyTwinDevice --hub-name $hubname --resource-group tutorial-iot-hub-rg

# Retrieve the device connection string, you need this later:
az iot hub device-identity show-connection-string --device-id MyTwinDevice --hub-name $hubname --resource-group tutorial-iot-hub-rg -o table

```

Send state information

You use desired properties to send state information from a back-end application to a device. In this section, you see how to:

- Receive and process desired properties on a device.
- Send desired properties from a back-end application.

To view the simulated device sample code that receives desired properties, navigate to the **iot-hub/Tutorials/DeviceTwins** folder in the sample Node.js project you downloaded. Then open the SimulatedDevice.js file in a text editor.

The following sections describe the code that runs on the simulated device that responds to desired property changes sent from the back end application:

Retrieve the device twin object

The following code connects to your IoT hub using a device connection string:

```
// Get the device connection string from a command line argument
var connectionString = process.argv[2];
```

The following code gets a twin from the client object:

```
// Get the device twin
client.getTwin(function(err, twin) {
  if (err) {
    console.error(chalk.red('Could not get device twin'));
  } else {
    console.log(chalk.green('Device twin created'));
  }
});
```

Sample desired properties

You can structure your desired properties in any way that's convenient to your application. This example uses one top-level property called **fanOn** and groups the remaining properties into separate **components**. The following JSON snippet shows the structure of the desired properties this tutorial uses:

```
{
  "fanOn": "true",
  "components": {
    "system": {
      "id": "17",
      "units": "farenheit",
      "firmwareVersion": "9.75"
    },
    "wifi" : {
      "channel" : "6",
      "ssid": "my_network"
    },
    "climate" : {
      "minTemperature": "68",
      "maxTemperature": "76"
    }
  }
}
```

Create handlers

You can create handlers for desired property updates that respond to updates at different levels in the JSON hierarchy. For example, this handler sees all desired property changes sent to the device from a back-end application. The **delta** variable contains the desired properties sent from the solution back end:

```
// Handle all desired property updates
twin.on('properties.desired', function(delta) {
  console.log(chalk.yellow('\nNew desired properties received in patch:'));
});
```

The following handler only reacts to changes made to the **fanOn** desired property:

```
// Handle changes to the fanOn desired property
twin.on('properties.desired.fanOn', function(fanOn) {
  console.log(chalk.green('\nSetting fan state to ' + fanOn));

  // Update the reported property after processing the desired property
  reportedPropertiesPatch.fanOn = fanOn ? fanOn : '{unknown}';
});
```

Handlers for multiple properties

In the example desired properties JSON shown previously, the **climate** node under **components** contains two properties, **minTemperature** and **maxTemperature**.

A device's local **twin** object stores a complete set of desired and reported properties. The **delta** sent from the back end might update just a subset of desired properties. In the following code snippet, if the simulated device receives an update to just one of **minTemperature** and **maxTemperature**, it uses the value in the local twin for the other value to configure the device:

```

// Handle desired properties updates to the climate component
twin.on('properties.desired.components.climate', function(delta) {
    if (delta.minTemperature || delta.maxTemperature) {
        console.log(chalk.green('\nUpdating desired tempertures in climate component:'));
        console.log('Configuring minimum temperature: ' +
twin.properties.desired.components.climate.minTemperature);
        console.log('Configuring maximum temperture: ' +
twin.properties.desired.components.climate.maxTemperature);

        // Update the reported properties and send them to the hub
        reportedPropertiesPatch.minTemperature = twin.properties.desired.components.climate.minTemperature;
        reportedPropertiesPatch.maxTemperature = twin.properties.desired.components.climate.maxTemperature;
        sendReportedProperties();
    }
});
```

The local **twin** object stores a complete set of desired and reported properties. The **delta** sent from the back end might update just a subset of desired properties.

Handle insert, update, and delete operations

The desired properties sent from the back end don't indicate what operation is being performed on a particular desired property. Your code needs to infer the operation from the current set of desired properties stored locally and the changes sent from the hub.

The following snippet shows how the simulated device handles insert, update, and delete operations on the list of **components** in the desired properties. You can see how to use **null** values to indicate that a component should be deleted:

```

// Keep track of all the components the device knows about
var componentList = {};

// Use this componentList list and compare it to the delta to infer
// if anything was added, deleted, or updated.
twin.on('properties.desired.components', function(delta) {
  if (delta === null) {
    componentList = {};
  }
  else {
    Object.keys(delta).forEach(function(key) {

      if (delta[key] === null && componentList[key]) {
        // The delta contains a null value, and the
        // device has a record of this component.
        // Must be a delete operation.
        console.log(chalk.green('\nDeleting component ' + key));
        delete componentList[key];

      } else if (delta[key]) {
        if (componentList[key]) {
          // The delta contains a component, and the
          // device has a record of it.
          // Must be an update operation.
          console.log(chalk.green('\nUpdating component ' + key + ':'));

          console.log(JSON.stringify(delta[key]));
          // Store the complete object instead of just the delta
          componentList[key] = twin.properties.desired.components[key];

        } else {
          // The delta contains a component, and the
          // device has no record of it.
          // Must be an add operation.
          console.log(chalk.green('\nAdding component ' + key + ':'));

          console.log(JSON.stringify(delta[key]));
          // Store the complete object instead of just the delta
          componentList[key] = twin.properties.desired.components[key];
        }
      }
    });
  }
});

```

Send desired properties to a device from the back end

You've seen how a device implements handlers for receiving desired property updates. This section shows you how to send desired property changes to a device from a back-end application.

To view the simulated device sample code that receives desired properties, navigate to the **iot-hub/Tutorials/DeviceTwins** folder in the sample Node.js project you downloaded. Then open the ServiceClient.js file in a text editor.

The following code snippet shows how to connect to the device identity registry and access the twin for a specific device:

```
// Create a device identity registry object
var registry = Registry.fromConnectionString(connectionString);

// Get the device twin and send desired property update patches at intervals.
// Print the reported properties after some of the desired property updates.
registry.getTwin(deviceId, async (err, twin) => {
  if (err) {
    console.error(err.message);
  } else {
    console.log('Got device twin');
```

The following snippet shows different desired property *patches* the back end application sends to the device:

```

// Turn the fan on
var twinPatchFanOn = {
  properties: {
    desired: {
      patchId: "Switch fan on",
      fanOn: "false",
    }
  }
};

// Set the maximum temperature for the climate component
var twinPatchSetMaxTemperature = {
  properties: {
    desired: {
      patchId: "Set maximum temperature",
      components: {
        climate: {
          maxTemperature: "92"
        }
      }
    }
  }
};

// Add a new component
var twinPatchAddWifiComponent = {
  properties: {
    desired: {
      patchId: "Add WiFi component",
      components: {
        wifi: {
          channel: "6",
          ssid: "my_network"
        }
      }
    }
  }
};

// Update the WiFi component
var twinPatchUpdateWifiComponent = {
  properties: {
    desired: {
      patchId: "Update WiFi component",
      components: {
        wifi: {
          channel: "13",
          ssid: "my_other_network"
        }
      }
    }
  }
};

// Delete the WiFi component
var twinPatchDeleteWifiComponent = {
  properties: {
    desired: {
      patchId: "Delete WiFi component",
      components: {
        wifi: null
      }
    }
  }
};

```

The following snippet shows how the back-end application sends a desired property update to a device:

```
// Send a desired property update patch
async function sendDesiredProperties(twin, patch) {
  twin.update(patch, (err, twin) => {
    if (err) {
      console.error(err.message);
    } else {
      console.log(chalk.green(`\nSent ${twin.properties.desired.patchId} patch:`));
      console.log(JSON.stringify(patch, null, 2));
    }
  });
}
```

Run the applications

In this section, you run two sample applications to observe as a back-end application sends desired property updates to a simulated device application.

To run the simulated device and back-end applications, you need the device and service connection strings. You made a note of the connection strings when you created the resources at the start of this tutorial.

To run the simulated device application, open a shell or command prompt window and navigate to the **iot-hub/Tutorials/DeviceTwins** folder in the Node.js project you downloaded. Then run the following commands:

```
npm install
node SimulatedDevice.js "{your device connection string}"
```

To run the back-end application, open another shell or command prompt window. Then navigate to the **iot-hub/Tutorials/DeviceTwins** folder in the Node.js project you downloaded. Then run the following commands:

```
npm install
node ServiceClient.js "{your service connection string}"
```

The following screenshot shows the output from the simulated device application and highlights how it handles an update to the **maxTemperature** desired property. You can see how both the top-level handler and the climate component handlers run:

```
ca Command Prompt
{
    "firmwareVersion": "1.2.1",
    "lastPatchReceivedId": "Init",
    "fanOn": "false",
    "minTemperature": "68",
    "maxTemperature": "76"
}

New desired properties received in patch:
Switch fan on

Setting fan state to false
New desired properties received in patch:
Set maximum temperature

Updating component climate:
["maxTemperature": "92"]

Updating desired temperatures in climate component:
Configuring minimum temperature: 68
Configuring maximum temperture: 92

Twin state reported
{
    "firmwareVersion": "1.2.1",
    "lastPatchReceivedId": "Set maximum temperature",
    "fanOn": "false",
    "minTemperature": "68",
    "maxTemperature": "92"
}
```

The following screenshot shows the output from the back-end application and highlights how it sends an update to the **maxTemperature** desired property:

```
ca Command Prompt
}

Sent Switch fan on patch:
{
    "properties": [
        "desired": {
            "patchId": "Switch fan on",
            "fanOn": "false"
        }
    ]
}

Sent Set maximum temperature patch:
{
    "properties": [
        "desired": {
            "patchId": "Set maximum temperature",
            "components": {
                "climate": {
                    "maxTemperature": "92"
                }
            }
        }
    ]
}

Sent Add WiFi component patch:
{
    "properties": [
        "desired": {
```

Receive state information

Your back-end application receives state information from a device as reported properties. A device sets the reported properties, and sends them to your hub. A back-end application can read the current values of the reported properties from the device twin stored in your hub.

Send reported properties from a device

You can send updates to reported property values as a patch. The following snippet shows a template for the patch the simulated device sends. The simulated device updates the fields in the patch before sending it to the hub:

```
// Create a patch to send to the hub
var reportedPropertiesPatch = {
    firmwareVersion:'1.2.1',
    lastPatchReceivedId: '',
    fanOn:'',
    minTemperature:'',
    maxTemperature:''
};
```

The simulated device uses the following function to send the patch that contains the reported properties to the hub:

```
// Send the reported properties patch to the hub
function sendReportedProperties() {
    twin.properties.reported.update(reportedPropertiesPatch, function(err) {
        if (err) throw err;
        console.log(chalk.blue('\nTwin state reported'));
        console.log(JSON.stringify(reportedPropertiesPatch, null, 2));
    });
}
```

Process reported properties

A back-end application accesses the current reported property values for a device through the device twin. The following snippet shows you how the back-end application reads the reported property values for the simulated device:

```
// Display the reported properties from the device
function printReportedProperties(twin) {
    console.log("Last received patch: " + twin.properties.reported.lastPatchReceivedId);
    console.log("Firmware version: " + twin.properties.reported.firmwareVersion);
    console.log("Fan status: " + twin.properties.reported.fanOn);
    console.log("Min temperature set: " + twin.properties.reported.minTemperature);
    console.log("Max temperature set: " + twin.properties.reported.maxTemperature);
}
```

Run the applications

In this section, you run two sample applications to observe as a simulated device application sends reported property updates to a back-end application.

You run the same two sample applications that you ran to see how desired properties are sent to a device.

To run the simulated device and back-end applications, you need the device and service connection strings. You made a note of the connection strings when you created the resources at the start of this tutorial.

To run the simulated device application, open a shell or command prompt window and navigate to the **iot-hub/Tutorials/DeviceTwins** folder in the Node.js project you downloaded. Then run the following commands:

```
npm install
node SimulatedDevice.js "{your device connection string}"
```

To run the back-end application, open another shell or command prompt window. Then navigate to the **iot-hub/Tutorials/DeviceTwins** folder in the Node.js project you downloaded. Then run the following commands:

```
npm install
node ServiceClient.js "{your service connection string}"
```

The following screenshot shows the output from the simulated device application and highlights how it sends a reported property update to your hub:

A screenshot of a Windows Command Prompt window titled "Command Prompt". The window contains the following text:

```
Switch fan on
Setting fan state to false
New desired properties received in patch:
Set maximum temperature
Updating component climate:
{"maxTemperature": "92"}

Updating desired temperatures in climate component:
Configuring minimum temperature: 68
Configuring maximum temperture: 92

Twin state reported
{
  "firmwareVersion": "1.2.1",
  "lastPatchReceivedId": "Set maximum temperature",
  "fanOn": "false",
  "minTemperature": "68",
  "maxTemperature": "92"
}

New desired properties received in patch:
Add WiFi component
Adding component wifi:
{"channel": "6", "ssid": "my_network"}
```

The JSON object under "Twin state reported" is highlighted with a red rectangle.

The following screenshot shows the output from the back-end application and highlights how it receives and processes a reported property update from a device:

A screenshot of a Windows Command Prompt window titled "Command Prompt". The window contains the following text:

```
Sent Delete WiFi component patch:
{
  "properties": {
    "desired": {
      "patchId": "Delete WiFi component",
      "components": {
        "wifi": null
      }
    }
  }
}

Final reported properties from the device
Last received patch: Set maximum temperature
Firmware version: 1.2.1
Fan status: false
Min temperature set: 68
Max temperature set: 92

c:\repos\azure-iot-samples-node\Tutorials\DeviceTwins>
```

The "Final reported properties from the device" section is highlighted with a red rectangle.

Clean up resources

If you plan to complete the next tutorial, leave the resource group and IoT hub and reuse them later.

If you don't need the IoT hub any longer, delete it and the resource group in the portal. To do so, select the **tutorial-iot-hub-rg** resource group that contains your IoT hub and click **Delete**.

Alternatively, use the CLI:

```
# Delete your resource group and its contents  
az group delete --name tutorial-iot-hub-rg
```

Next steps

In this tutorial, you learned how to synchronize state information between your devices and your IoT hub. Advance to the next tutorial to learn how to use device twins to implement a firmware update process.

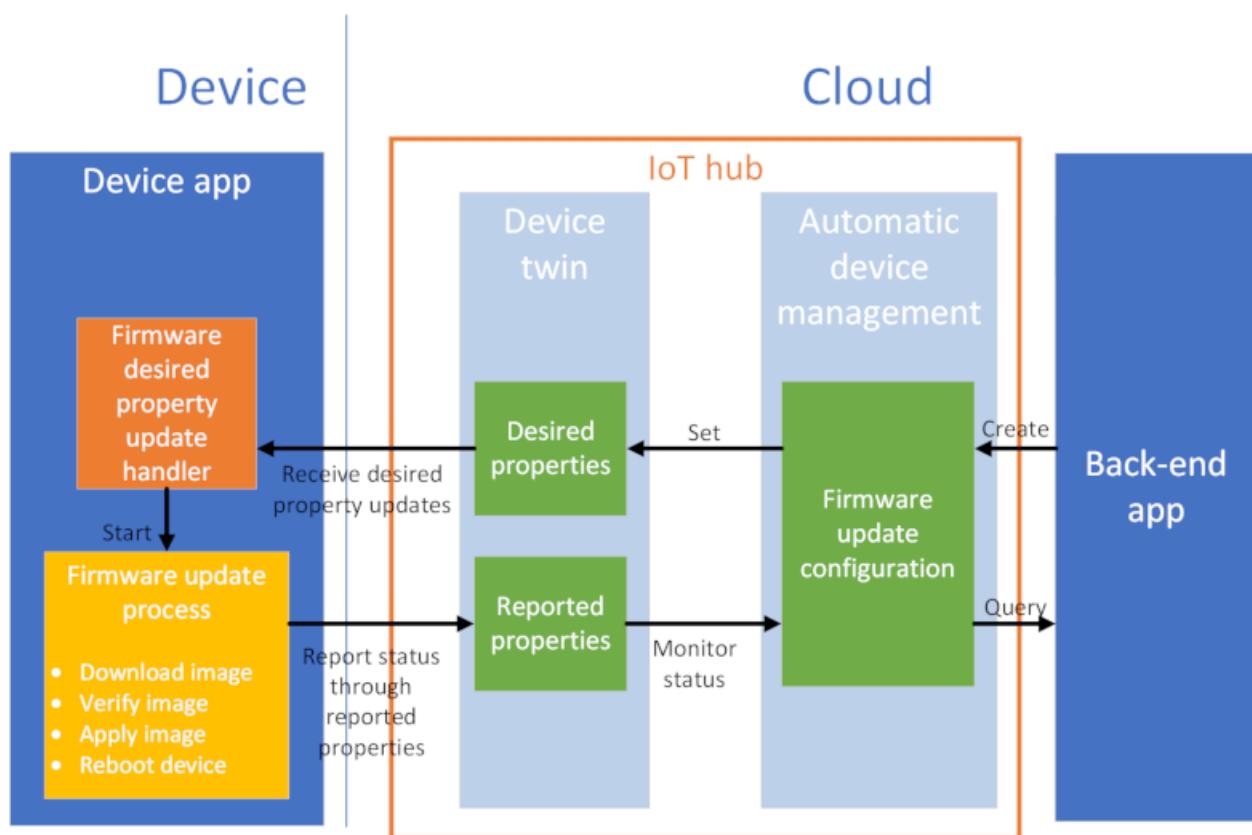
[Implement a device firmware update process](#)

Tutorial: Implement a device firmware update process

2/28/2019 • 11 minutes to read

You may need to update the firmware on the devices connected to your IoT hub. For example, you might want to add new features to the firmware or apply security patches. In many IoT scenarios, it's impractical to physically visit and then manually apply firmware updates to your devices. This tutorial shows how you can start and monitor the firmware update process remotely through a back-end application connected to your hub.

To create and monitor the firmware update process, the back-end application in this tutorial creates a *configuration* in your IoT hub. IoT Hub [automatic device management](#) uses this configuration to update a set of *device twin desired properties* on all your chiller devices. The desired properties specify the details of the firmware update that's required. While the chiller devices are running the firmware update process, they report their status to the back-end application using *device twin reported properties*. The back-end application can use the configuration to monitor the reported properties sent from the device and track the firmware update process to completion:



In this tutorial, you complete the following tasks:

- Create an IoT hub and add a test device to the device identity registry.
- Create a configuration to define the firmware update.
- Simulate the firmware update process on a device.
- Receive status updates from the device as the firmware update progresses.

Open Azure Cloud Shell

Azure Cloud Shell is a free, interactive shell that you can use to run the steps in this article. Common Azure tools

are preinstalled and configured in Cloud Shell for you to use with your account. Just select the **Copy** button to copy the code, paste it in Cloud Shell, and then press Enter to run it. There are a few ways to open Cloud Shell:

Select Try It in the upper-right corner of a code block.	
Open Cloud Shell in your browser.	
Select the Cloud Shell button on the menu in the upper-right corner of the Azure portal .	

If you don't have an Azure subscription, create a [free account](#) before you begin.

Prerequisites

The two sample applications you run in this quickstart are written using Node.js. You need Node.js v4.x.x or later on your development machine.

You can download Node.js for multiple platforms from nodejs.org.

You can verify the current version of Node.js on your development machine using the following command:

```
node --version
```

Download the sample Node.js project from <https://github.com/Azure-Samples/azure-iot-samples-node/archive/master.zip> and extract the ZIP archive.

Set up Azure resources

To complete this tutorial, your Azure subscription must have an IoT hub with a device added to the device identity registry. The entry in the device identity registry enables the simulated device you run in this tutorial to connect to your hub.

If you don't already have an IoT hub set up in your subscription, you can set one up with following CLI script. This script uses the name **tutorial-iot-hub** for the IoT hub, you should replace this name with your own unique name when you run it. The script creates the resource group and hub in the **Central US** region, which you can change to a region closer to you. The script retrieves your IoT hub service connection string, which you use in the back-end sample application to connect to your IoT hub:

```
hubname=tutorial-iot-hub
location=centralus

# Install the IoT extension if it's not already installed
az extension add --name azure-cli-iot-ext

# Create a resource group
az group create --name tutorial-iot-hub-rg --location $location

# Create your free-tier IoT Hub. You can only have one free IoT Hub per subscription
az iot hub create --name $hubname --location $location --resource-group tutorial-iot-hub-rg --sku F1

# Make a note of the service connection string, you need it later
az iot hub show-connection-string --hub-name $hubname -o table
```

This tutorial uses a simulated device called **MyFirmwareUpdateDevice**. The following script adds this device to your device identity registry, sets a tag value, and retrieves its connection string:

```
# Set the name of your IoT hub
hubname=tutorial-iot-hub

# Create the device in the identity registry
az iot hub device-identity create --device-id MyFirmwareUpdateDevice --hub-name $hubname --resource-group
tutorial-iot-hub-rg

# Add a device type tag
az iot hub device-twin update --device-id MyFirmwareUpdateDevice --hub-name $hubname --set
tags='{"devicetype":"chiller"}'

# Retrieve the device connection string, you need this later
az iot hub device-identity show-connection-string --device-id MyFirmwareUpdateDevice --hub-name $hubname --
resource-group tutorial-iot-hub-rg -o table
```

TIP

If you run these commands at a Windows command prompt or Powershell prompt, see the [azure-iot-cli-extension tips](#) page for information about how to quote JSON strings.

Start the firmware update

You create an [automatic device management configuration](#) in the back-end application to begin the firmware update process on all devices tagged with a **devicetype** of chiller. In this section, you see how to:

- Create a configuration from a back-end application.
- Monitor the job to completion.

Use desired properties to start the firmware upgrade from the back-end application

To view the back-end application code that creates the configuration, navigate to the **iot-hub/Tutorials/FirmwareUpdate** folder in the sample Node.js project you downloaded. Then open the ServiceClient.js file in a text editor.

The back-end application creates the following configuration:

```

var firmwareConfig = {
  id: sampleConfigId,
  content: {
    deviceContent: {
      'properties.desired.firmware': {
        fwVersion: fwVersion,
        fwPackageURI: fwPackageURI,
        fwPackageCheckValue: fwPackageCheckValue
      }
    }
  },
  // Maximum of 5 metrics per configuration
  metrics: {
    queries: {
      current: 'SELECT deviceId FROM devices WHERE configurations.[[firmware285]].status=\''Applied\' AND properties.reported.firmware/fwUpdateStatus=\''current\'',
      applying: 'SELECT deviceId FROM devices WHERE configurations.[[firmware285]].status=\''Applied\' AND (properties.reported.firmware/fwUpdateStatus=\''downloading\' OR properties.reported.firmware/fwUpdateStatus=\''verifying\' OR properties.reported.firmware/fwUpdateStatus=\''applying\')',
      rebooting: 'SELECT deviceId FROM devices WHERE configurations.[[firmware285]].status=\''Applied\' AND properties.reported.firmware/fwUpdateStatus=\''rebooting\'',
      error: 'SELECT deviceId FROM devices WHERE configurations.[[firmware285]].status=\''Applied\' AND properties.reported.firmware/fwUpdateStatus=\''error\'',
      rolledback: 'SELECT deviceId FROM devices WHERE configurations.[[firmware285]].status=\''Applied\' AND properties.reported.firmware/fwUpdateStatus=\''rolledback\''
    }
  },
  // Specify the devices the firmware update applies to
  targetCondition: 'tags.devicetype = \'chiller\'',
  priority: 20
};

```

The configuration includes the following sections:

- `content` specifies the firmware desired properties sent to the selected devices.
- `metrics` specifies the queries to run that report the status of the firmware update.
- `targetCondition` selects the devices to receive the firmware update.
- `priority` sets the relative priority of this configuration to other configurations.

The back-end application uses the following code to create the configuration to set the desired properties:

```

var createConfiguration = function(done) {
  console.log();
  console.log('Add new configuration with id ' + firmwareConfig.id + ' and priority ' +
  firmwareConfig.priority);

  registry.addConfiguration(firmwareConfig, function(err) {
    if (err) {
      console.log('Add configuration failed: ' + err);
      done();
    } else {
      console.log('Add configuration succeeded');
      done();
    }
  });
};


```

After it creates the configuration, the application monitors the firmware update:

```
var monitorConfiguration = function(done) {
  console.log('Monitor metrics for configuration: ' + sampleConfigId);
  setInterval(function(){
    registry.getConfiguration(sampleConfigId, function(err, config) {
      if (err) {
        console.log('getConfiguration failed: ' + err);
      } else {
        console.log('System metrics:');
        console.log(JSON.stringify(config.systemMetrics.results, null, ' '));
        console.log('Custom metrics:');
        console.log(JSON.stringify(config.metrics.results, null, ' '));
      }
    });
  }, 20000);
  done();
};
```

A configuration reports two types of metrics:

- System metrics that report how many devices are targeted and how many devices have the update applied.
- Custom metrics generated by the queries you add to the configuration. In this tutorial, the queries report how many devices are at each stage of the update process.

Respond to the firmware upgrade request on the device

To view the simulated device code that handles the firmware desired properties sent from the back-end application, navigate to the **iot-hub/Tutorials/FirmwareUpdate** folder in the sample Node.js project you downloaded. Then open the SimulatedDevice.js file in a text editor.

The simulated device application creates a handler for updates to the **properties.desired.firmware** desired properties in the device twin. In the handler, it carries out some basic checks on the desired properties before launching the update process:

```

// Handle firmware desired property updates - this triggers the firmware update flow
twin.on('properties.desired.firmware', function(fwUpdateDesiredProperties) {
  console.log(chalk.green('\nCurrent firmware version: ' +
  twin.properties.reported.firmware.currentFwVersion));
  console.log(chalk.green('Starting firmware update flow using this data:'));
  console.log(JSON.stringify(fwUpdateDesiredProperties, null, 2));
  desiredFirmwareProperties = twin.properties.desired.firmware;

  if (fwUpdateDesiredProperties.fwVersion == twin.properties.reported.firmware.currentFwVersion) {
    sendStatusUpdate('current', 'Firmware already up to date', function (err) {
      if (err) {
        console.error(chalk.red('Error occurred sending status update : ' + err.message));
      }
      return;
    });
  }
  if (fwUpdateInProgress) {
    sendStatusUpdate('current', 'Firmware update already running', function (err) {
      if (err) {
        console.error(chalk.red('Error occurred sending status update : ' + err.message));
      }
      return;
    });
  }
  if (!fwUpdateDesiredProperties.fwPackageURI.startsWith('https')) {
    sendStatusUpdate('error', 'Insecure package URI', function (err) {
      if (err) {
        console.error(chalk.red('Error occurred sending status update : ' + err.message));
      }
      return;
    });
  }
}

fwUpdateInProgress = true;

sendStartingUpdate(fwUpdateDesiredProperties.fwVersion, function (err) {
  if (err) {
    console.error(chalk.red('Error occurred sending starting update : ' + err.message));
  }
  return;
});
initiateFirmwareUpdateFlow(function(err, result) {
  fwUpdateInProgress = false;
  if (!err) {
    console.log(chalk.green('Completed firmwareUpdate flow. New version: ' + result));
    sendFinishedUpdate(result, function (err) {
      if (err) {
        console.error(chalk.red('Error occurred sending finished update : ' + err.message));
      }
      return;
    });
  }
}, twin.properties.reported.firmware.currentFwVersion);
});

```

Update the firmware

The **initiateFirmwareUpdateFlow** function runs the update. This function uses the **waterfall** function to run the phases of the update process in sequence. In this example, the firmware update has four phases. The first phase downloads the image, the second phase verifies the image using a checksum, the third phase applies the image, and the last phase reboots the device:

```

// Implementation of firmwareUpdate flow
function initiateFirmwareUpdateFlow(callback, currentVersion) {

    async.waterfall([
        downloadImage,
        verifyImage,
        applyImage,
        reboot
    ], function(err, result) {
        if (err) {
            console.error(chalk.red('Error occurred firmwareUpdate flow : ' + err.message));
            sendStatusUpdate('error', err.message, function (err) {
                if (err) {
                    console.error(chalk.red('Error occurred sending status update : ' + err.message));
                }
            });
            setTimeout(function() {
                console.log('Simulate rolling back update due to error');
                sendStatusUpdate('rolledback', 'Rolled back to: ' + currentVersion, function (err) {
                    if (err) {
                        console.error(chalk.red('Error occurred sending status update : ' + err.message));
                    }
                });
                callback(err, result);
            }, 5000);
        } else {
            callback(null, result);
        }
    });
}

```

During the update process, the simulated device reports on progress using reported properties:

```

// Firmware update patch
// currentFwVersion: The firmware version currently running on the device.
// pendingFwVersion: The next version to update to, should match what's
//                   specified in the desired properties. Blank if there
//                   is no pending update (fwUpdateStatus is 'current').
// fwUpdateStatus: Defines the progress of the update so that it can be
//                   categorized from a summary view. One of:
//                   - current: There is no pending firmware update. currentFwVersion should
//                               match fwVersion from desired properties.
//                   - downloading: Firmware update image is downloading.
//                   - verifying: Verifying image file checksum and any other validations.
//                   - applying: Update to the new image file is in progress.
//                   - rebooting: Device is rebooting as part of update process.
//                   - error: An error occurred during the update process. Additional details
//                             should be specified in fwUpdateSubstatus.
//                   - rolledback: Update rolled back to the previous version due to an error.
// fwUpdateSubstatus: Any additional detail for the fwUpdateStatus . May include
//                   reasons for error or rollback states, or download %.
//
// var twinPatchFirmwareUpdate = {
//   firmware: {
//     currentFwVersion: '1.0.0',
//     pendingFwVersion: '',
//     fwUpdateStatus: 'current',
//     fwUpdateSubstatus: '',
//     lastFwUpdateStartTime: '',
//     lastFwUpdateEndTime: ''
//   }
// };

```

The following snippet shows the implementation of the download phase. During the download phase, the

simulated device uses reported properties to send status information to the back-end application:

```
// Simulate downloading an image
function downloadImage(callback) {
    console.log('Simulating image download from: ' + desiredFirmwareProperties.fwPackageURI);

    async.waterfall([
        function(callback) {
            sendStatusUpdate('downloading', 'Start downloading', function (err) {
                if (err) {
                    console.error(chalk.red('Error occurred sending status update : ' + err.message));
                }
            });
            callback(null);
        },
        function(callback) {
            // Simulate a delay downloading the image.
            setTimeout(function() {
                // Simulate some firmware image data
                var imageData = '[Fake firmware image data]';
                callback(null, imageData);
            }, 3000);
        },
        function(imageData, callback) {
            console.log('Downloaded image data: ' + imageData);
            sendStatusUpdate('downloading', 'Finished downloading', function (err) {
                if (err) {
                    console.error(chalk.red('Error occurred sending status update : ' + err.message));
                }
            });
            callback(null, imageData);
        }
    ], function (err, result) {
        callback(err, result);
    });
}
```

Run the sample

In this section, you run two sample applications to observe as a back-end application creates the configuration to manage the firmware update process on the simulated device.

To run the simulated device and back-end applications, you need the device and service connection strings. You made a note of the connection strings when you created the resources at the start of this tutorial.

To run the simulated device application, open a shell or command prompt window and navigate to the **iot-hub/Tutorials/FirmwareUpdate** folder in the Node.js project you downloaded. Then run the following commands:

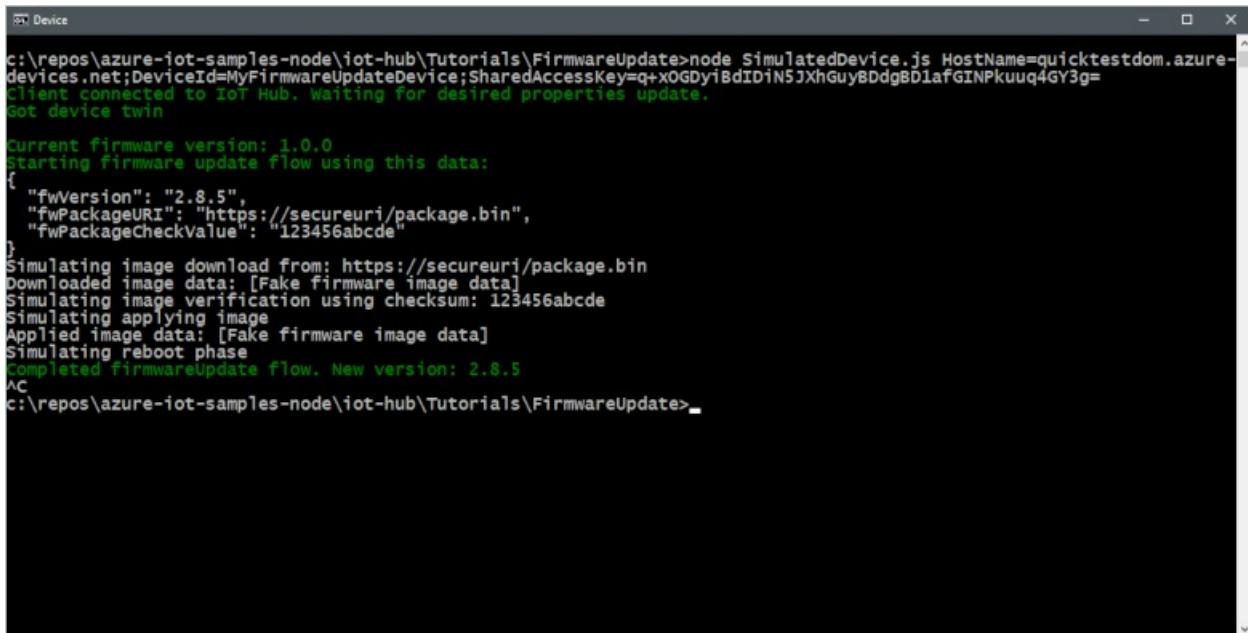
```
npm install
node SimulatedDevice.js "{your device connection string}"
```

To run the back-end application, open another shell or command prompt window. Then navigate to the **iot-hub/Tutorials/FirmwareUpdate** folder in the Node.js project you downloaded. Then run the following commands:

```
npm install
node ServiceClient.js "{your service connection string}"
```

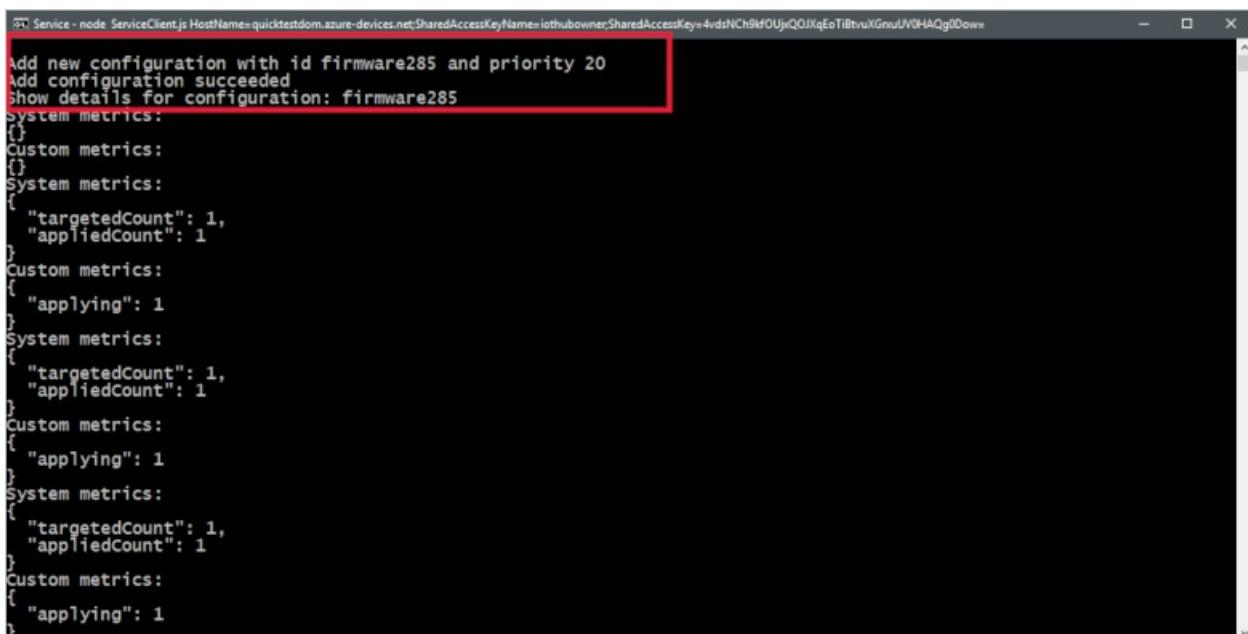
The following screenshot shows the output from the simulated device application and shows how it responds to

the firmware desired properties update from the back-end application:



```
Device
c:\repos\azure-iot-samples-node\iot-hub\Tutorials\FirmwareUpdate>node SimulatedDevice.js HostName=quicktestdom.azure-devices.net;DeviceId=MyFirmwareUpdateDevice;SharedAccessKey=q+x0GDyiBdIDiN5JXhGuy8DdgBD1afGINPkuuq4GY3g=Client connected to IoT Hub. Waiting for desired properties update.
Got device twin
Current firmware version: 1.0.0
Starting firmware update flow using this data:
{
  "fwVersion": "2.8.5",
  "fwPackageURI": "https://secureuri/package.bin",
  "fwPackageCheckValue": "123456abcde"
}
Simulating image download from: https://secureuri/package.bin
Downloaded image data: [Fake firmware image data]
Simulating image verification using checksum: 123456abcde
Simulating applying image
Applied image data: [Fake firmware image data]
Simulating reboot phase
Completed firmwareUpdate flow. New version: 2.8.5
^C
c:\repos\azure-iot-samples-node\iot-hub\Tutorials\FirmwareUpdate>
```

The following screenshot shows the output from the back-end application and highlights how it creates the configuration to update the firmware desired properties:



```
Service - node ServiceClient.js HostName=quicktestdom.azure-devices.net;SharedAccessKeyName=iothubowner;SharedAccessKey=4vdshCh9kfOUjrQOJxqEoTbtvXGnxUV3HAQg0Down
Add new configuration with id firmware285 and priority 20
Add configuration succeeded
Show details for configuration: firmware285
System metrics:
{}
Custom metrics:
{}
System metrics:
[
  {
    "targetedCount": 1,
    "appliedCount": 1
  }
]
Custom metrics:
[
  {
    "applying": 1
  }
]
System metrics:
[
  {
    "targetedCount": 1,
    "appliedCount": 1
  }
]
Custom metrics:
[
  {
    "applying": 1
  }
]
System metrics:
[
  {
    "targetedCount": 1,
    "appliedCount": 1
  }
]
Custom metrics:
[
  {
    "applying": 1
  }
]
```

The following screenshot shows the output from the back-end application and highlights how it monitors the firmware update metrics from the simulated device:

```
Service - node. ServiceClient.js HostName=quicktestdom.azure-devices.net;SharedAccessKeyName=iothubowner;SharedAccessKey=4vdshCh9kfOUjxQOJxqEoTbtvuxGnxuUVlHAQg0Down
Add new configuration with id firmware285 and priority 20
Add configuration succeeded
Show details for configuration: firmware285
System metrics:
[]
Custom metrics:
[]
System metrics:
[
  "targetedCount": 1,
  "appliedCount": 1
]
Custom metrics:
[
  "applying": 1
]
System metrics:
[
  "targetedCount": 1,
  "appliedCount": 1
]
Custom metrics:
[
  "applying": 1
]
System metrics:
[
  "targetedCount": 1,
  "appliedCount": 1
]
Custom metrics:
[
  "applying": 1
]
```

Because of latency in the IoT Hub device identity registry, you may not see every status update sent to the back-end application. You can also view the metrics in the portal in the **Automatic device management -> IoT device configuration** section of your IoT hub:

The screenshot shows the 'Device configurations' blade in the Azure portal. On the left, there's a sidebar with navigation links: Certificates, Properties, Locks, Automation script, EXPLORERS (Query explorer, IoT devices), and AUTOMATIC DEVICE MANAGEMENT (IoT Edge (preview), IoT device configuration (preview)). The main area has a search bar and buttons for 'Add Configuration', 'Refresh', and 'Delete'. A descriptive message about device configurations is displayed. Below is a table showing the configuration details:

CONFIGURATI...	TARGET CONDI...	PRIORITY	CREATION TIME	SYSTEM METRI...	CUSTOM METR...
firmware285	tags.devicety...	20	Mon Jun 25...	1 Targeted 1 Applied	0 current 1 applying 0 rebooting 0 error 0 rolledback

Clean up resources

If you plan to complete the next tutorial, leave the resource group and IoT hub and reuse them later.

If you don't need the IoT hub any longer, delete it and the resource group in the portal. To do so, select the **tutorial-iot-hub-rg** resource group that contains your IoT hub and click **Delete**.

Alternatively, use the CLI:

```
# Delete your resource group and its contents
az group delete --name tutorial-iot-hub-rg
```

Next steps

In this tutorial, you learned how to implement a firmware update process for your connected devices. Advance to the next tutorial to learn how to use Azure IoT Hub portal tools and Azure CLI commands to test device

connectivity.

[Use a simulated device to test connectivity with your IoT hub](#)

Tutorial: Use a simulated device to test connectivity with your IoT hub

2/28/2019 • 8 minutes to read

In this tutorial, you use Azure IoT Hub portal tools and Azure CLI commands to test device connectivity. This tutorial also uses a simple device simulator that you run on your desktop machine.

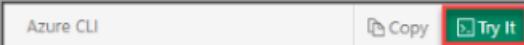
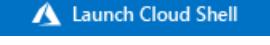
If you don't have an Azure subscription, [create a free account](#) before you begin.

In this tutorial, you learn how to:

- Check your device authentication
- Check device-to-cloud connectivity
- Check cloud-to-device connectivity
- Check device twin synchronization

Open Azure Cloud Shell

Azure Cloud Shell is a free, interactive shell that you can use to run the steps in this article. Common Azure tools are preinstalled and configured in Cloud Shell for you to use with your account. Just select the **Copy** button to copy the code, paste it in Cloud Shell, and then press Enter to run it. There are a few ways to open Cloud Shell:

Select Try It in the upper-right corner of a code block.	
Open Cloud Shell in your browser.	
Select the Cloud Shell button on the menu in the upper-right corner of the Azure portal .	

Prerequisites

The CLI scripts you run in this tutorial use the [Microsoft Azure IoT Extension for Azure CLI](#). To install this extension, run the following CLI command:

```
az extension add --name azure-cli-iot-ext
```

The device simulator application you run in this tutorial is written using Node.js. You need Node.js v4.x.x or later on your development machine.

You can download Node.js for multiple platforms from [nodejs.org](#).

You can verify the current version of Node.js on your development machine using the following command:

```
node --version
```

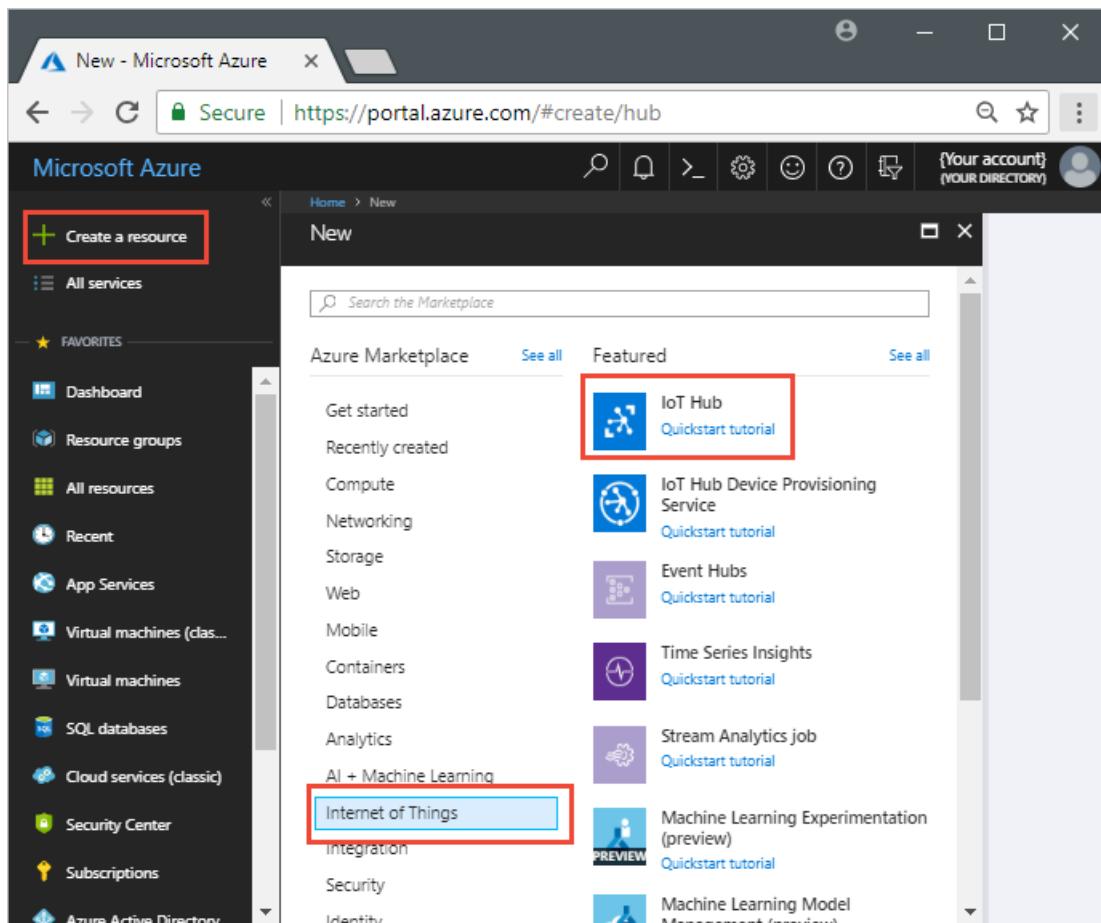
Download the sample device simulator Node.js project from <https://github.com/Azure-Samples/azure-iot-samples-node/archive/master.zip> and extract the ZIP archive.

Create an IoT hub

If you created a free or standard tier IoT hub in a previous quickstart or tutorial, you can skip this step.

To create an IoT Hub using the Azure portal:

1. Sign in to the [Azure portal](#).
2. Select **Create a resource > Internet of Things > IoT Hub**.



The screenshot shows the Microsoft Azure portal's 'New' blade. On the left, the 'Create a resource' button is highlighted with a red box. In the center, the 'Internet of Things' category is also highlighted with a red box. The 'IoT Hub' service is listed under 'Featured' with its icon and 'Quickstart tutorial' link.

3. To create your free-tier IoT hub, use the values in the following tables:

SETTING	VALUE
Subscription	Select your Azure subscription in the drop-down.
Resource group	Create new. This tutorial uses the name tutorials-iot-hub-rg .
Region	This tutorial uses West US . You can choose the region closest to you.
Name	The following screenshot uses the name tutorials-iot-hub . You must choose your own unique name when you create your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics **Size and scale** **Review + create**

Create an IoT Hub to help you connect, monitor, and manage billions of your IoT assets. [Learn More](#)

PROJECT DETAILS

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

* Subscription ?	Pay-As-You-Go
* Resource Group ?	<input checked="" type="radio"/> Create new <input type="radio"/> Use existing tutorials-iot-hub-rg
* Region ?	West US
* IoT Hub Name ?	tutorials-iot-hub

Review + create **Next: Size and scale »** [Automation options](#)

SETTING	VALUE
Pricing and scale tier	F1 Free. You can only have one free tier hub in a subscription.
IoT Hub units	1

Home > New > IoT hubs

IoT hub

Microsoft

Basics **Size and scale** **Review + create**

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS

* Pricing and scale tier ?	F1: Free tier
--	---------------

[Learn how to choose the right IoT Hub tier for your solution](#)

Number of F1 IoT Hub units [?](#) 1

This determines your IoT Hub scale capability and can be changed as your need increases.

Pricing and scale tier ? F1	Device-to-cloud-messages ? Enabled
Messages per day ? 8,000	Message routing ? Enabled
Cost per month 0.00 GBP	Cloud-to-device commands ? Enabled
	IoT Edge ? Enabled
	Device management ? Enabled

Advanced Settings

Review + create [Previous: Basics](#) [Automation options](#)

4. Click **Create**. It can take several minutes for the hub to be created.

The screenshot shows the 'Review + create' step of the IoT hub creation wizard. It's divided into two sections: 'BASICS' and 'SIZE AND SCALE'. In the 'BASICS' section, the subscription is 'Pay-As-You-Go', resource group is 'tutorials-iot-hub-rg', region is 'West US', and the IoT Hub Name is 'tutorials-iot-hub'. In the 'SIZE AND SCALE' section, the pricing tier is 'F1', there is 1 F1 IoT Hub unit, 8,000 messages per day, and the cost per month is 0.00 GBP. At the bottom, there are three buttons: 'Create' (highlighted with a red box), '< Previous: Size and scale', and 'Automation options'.

5. Make a note of the IoT hub name you chose. You use this value later in the tutorial.

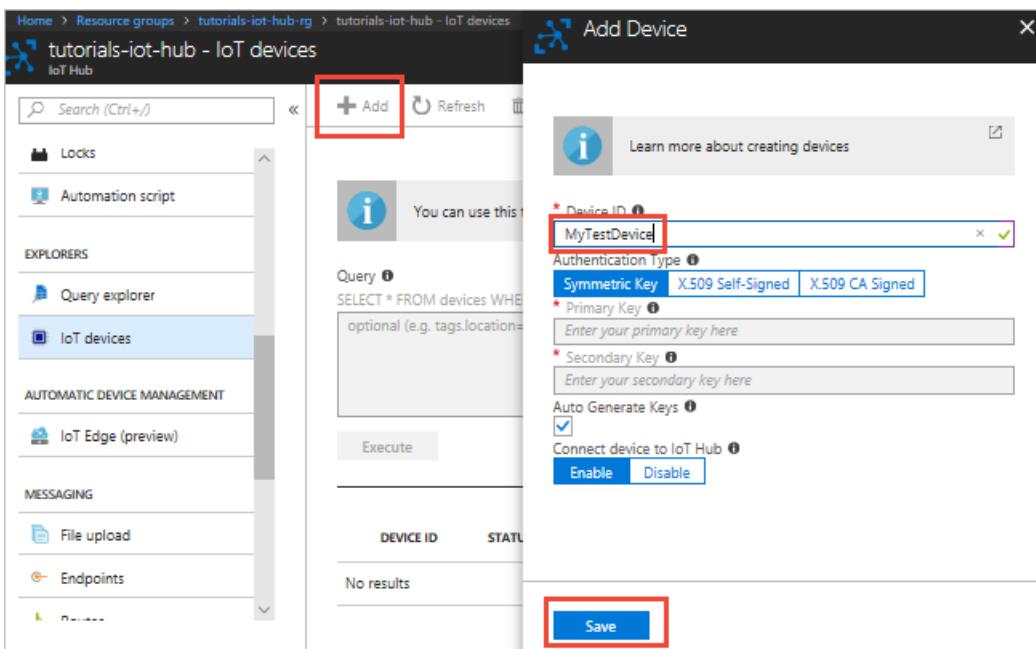
Check device authentication

A device must authenticate with your hub before it can exchange any data with the hub. You can use the **IoT Devices** tool in the **Device Management** section of the portal to manage your devices and check the authentication keys they're using. In this section of the tutorial, you add a new test device, retrieve its key, and check that the test device can connect to the hub. Later you reset the authentication key to observe what happens when a device tries to use an outdated key. This section of the tutorial uses the Azure portal to create, manage, and monitor a device, and the sample Node.js device simulator.

Sign in to the portal and navigate to your IoT hub. Then navigate to the **IoT Devices** tool:

The screenshot shows the IoT hub management blade for 'tutorials-iot-hub'. The left sidebar has sections for Locks, Automation script, EXPLORERS (Query explorer, IoT devices selected and highlighted with a red box), AUTOMATIC DEVICE MANAGEMENT (IoT Edge (preview)), and MESSAGING (File upload, Endpoints). The main area shows the IoT device configuration: Resource group 'tutorials-iot-hub-rg', Status 'Activating', Location 'West US', Subscription 'Pay-As-You-Go', Subscription ID '1f09d235-d756-45bc-97b3-ffdca8ed7e5a', Hostname '--', Pricing and scale tier 'F1 - Free', and Number of IoT Hub units '1'. A callout box in the bottom right says 'Need a way to provision millions of devices? IoT Hub Device Provisioning Service enables zero-touch, just-in-time provisioning to the right IoT hub without requiring human intervention.'

To register a new device, click **+ Add**, set **Device ID** to **MyTestDevice**, and click **Save**:



To retrieve the connection string for **MyTestDevice**, click on it in the list of devices and then copy the **Connection string-primary key** value. The connection string includes the *shared access key* for the device.

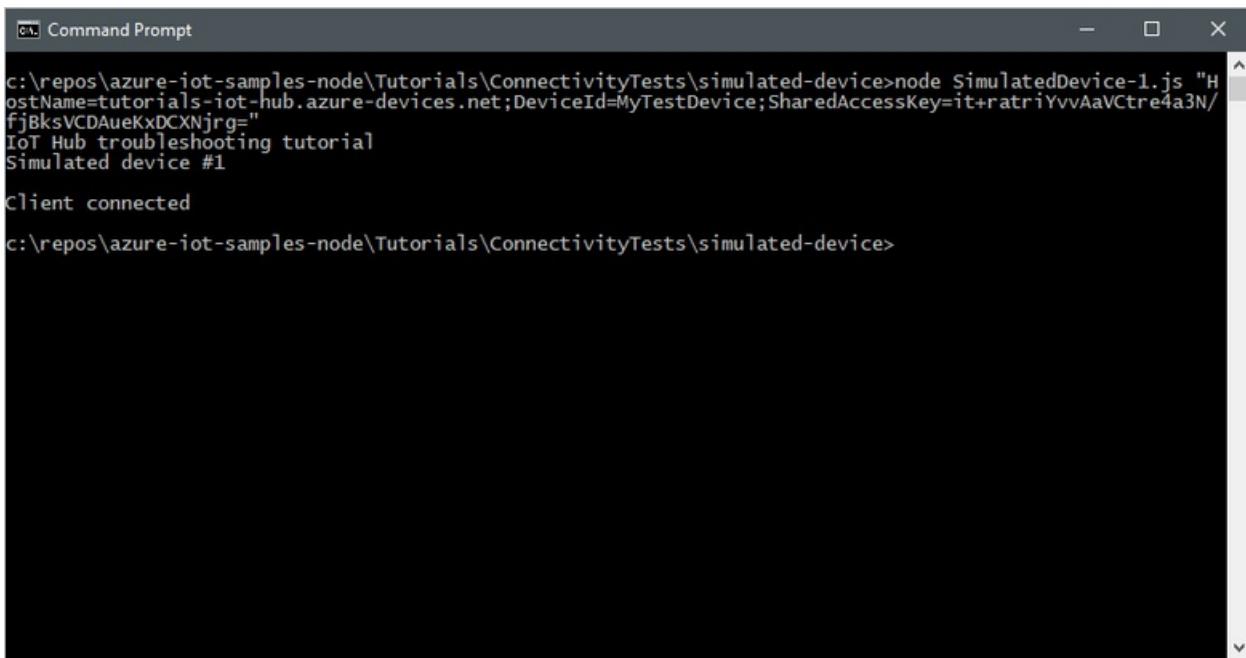
To simulate **MyTestDevice** sending telemetry to your IoT hub, run the Node.js simulated device application you downloaded previously.

In a terminal window on your development machine, navigate to the root folder of the sample Node.js project you downloaded. Then navigate to the **iot-hub\Tutorials\ConnectivityTests** folder.

In the terminal window, run the following commands to install the required libraries and run the simulated device application. Use the device connection string you made a note of when you added the device in the portal.

```
npm install
node SimulatedDevice-1.js "{your device connection string}"
```

The terminal window displays information as it tries to connect to your hub:



```
c:\repos\azure-iot-samples-node\Tutorials\ConnectivityTests\simulated-device>node SimulatedDevice-1.js "H
ostName=tutorials-iot-hub.azure-devices.net;DeviceId=MyTestDevice;SharedAccessKey=it+ratriYvvAaVCtre4a3N/
fjBksVCDAuexKxDXNjrg="
IoT Hub troubleshooting tutorial
Simulated device #1

Client connected

c:\repos\azure-iot-samples-node\Tutorials\ConnectivityTests\simulated-device>
```

You've now successfully authenticated from a device using a device key generated by your IoT hub.

Reset keys

In this section, you reset the device key and observe the error when the simulated device tries to connect.

To reset the primary device key for **MyTestDevice**, run the following commands:

```
# Generate a new Base64 encoded key using the current date
read key < <(date +%s | sha256sum | base64 | head -c 32)

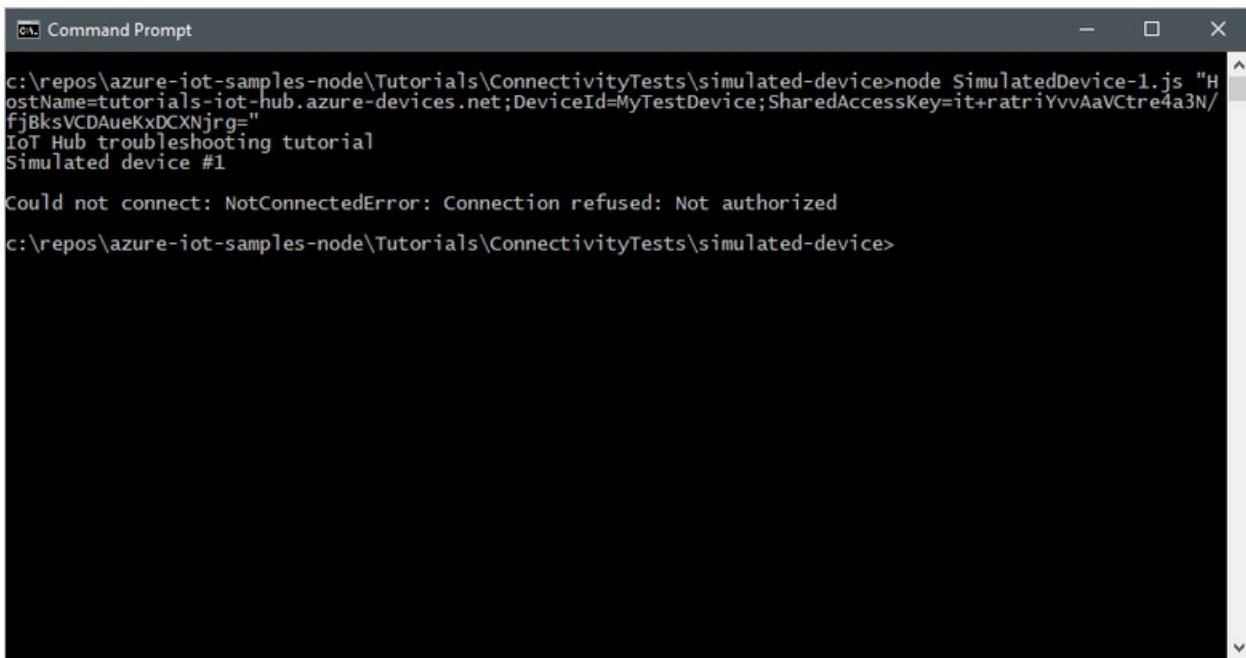
# Requires the IoT Extension for Azure CLI
# az extension add --name azure-cli-iot-ext

# Reset the primary device key for MyTestDevice
az iot hub device-identity update --device-id MyTestDevice --set authentication.symmetricKey.primaryKey=$key -
--hub-name {YourIoTHubName}
```

In the terminal window on your development machine, run the simulated device application again:

```
npm install
node SimulatedDevice-1.js "{your device connection string}"
```

This time you see an authentication error when the application tries to connect:



```
c:\repos\azure-iot-samples-node\Tutorials\ConnectivityTests\simulated-device>node SimulatedDevice-1.js "hostName=tutorials-iot-hub.azure-devices.net;DeviceId=MyTestDevice;SharedAccessKey=it+ratriYvvAaVCTre4a3N/fjBksVCDAuexKxDXNjrg=" IoT Hub troubleshooting tutorial  
Simulated device #1  
Could not connect: NotConnectedError: Connection refused: Not authorized  
c:\repos\azure-iot-samples-node\Tutorials\ConnectivityTests\simulated-device>
```

Generate shared access signature (SAS) token

If your device uses one of the IoT Hub device SDKs, the SDK library code generates the SAS token used to authenticate with the hub. A SAS token is generated from the name of your hub, the name of your device, and the device key.

In some scenarios, such as in a cloud protocol gateway or as part of a custom authentication scheme, you may need to generate the SAS token yourself. To troubleshoot issues with your SAS generation code, it's useful to generate a known-good SAS token to use during testing.

NOTE

The SimulatedDevice-2.js sample includes examples of generating a SAS token both with and without the SDK.

To generate a known-good SAS token using the CLI, run the following command:

```
az iot hub generate-sas-token --device-id MyTestDevice --hub-name {YourIoTHubName}
```

Make a note of the full text of the generated SAS token. A SAS token looks like the following:

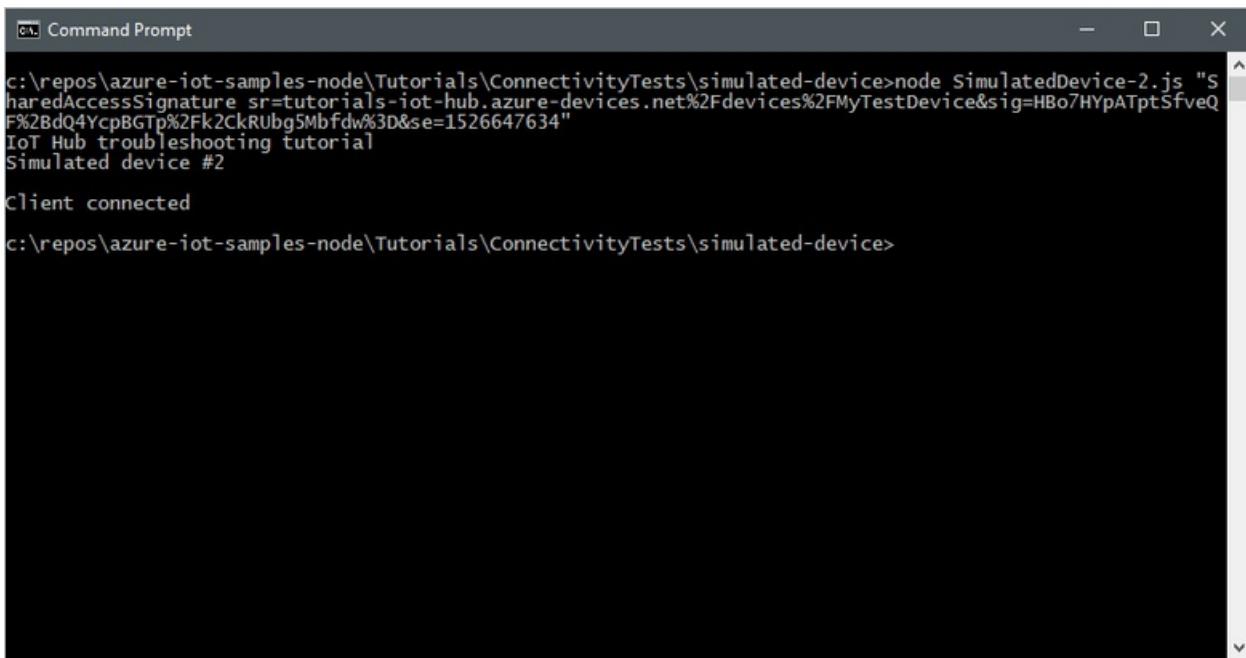
```
SharedAccessSignature sr=tutorials-iot-hub.azure-devices.net%2Fdevices%2FMyTestDevice&sig=....&se=1524155307
```

In a terminal window on your development machine, navigate to the root folder of the sample Node.js project you downloaded. Then navigate to the **iot-hub\Tutorials\ConnectivityTests\simulated-device** folder.

In the terminal window, run the following commands to install the required libraries and run the simulated device application:

```
npm install  
node SimulatedDevice-2.js "{Your SAS token}"
```

The terminal window displays information as it tries to connect to your hub using the SAS token:



```
c:\repos\azure-iot-samples-node\Tutorials\ConnectivityTests\simulated-device>node SimulatedDevice-2.js "S
haredAccessSignature sr=tutorials-iot-hub.azure-devices.net%2Fdevices%2FMyTestDevice&sig=HBo7HYpATptSfveQ
F%2BdQ4YcpBGTp%2Fk2CkRUBg5Mbfdw%3D&se=1526647634"
IoT Hub troubleshooting tutorial
Simulated device #2

Client connected

c:\repos\azure-iot-samples-node\Tutorials\ConnectivityTests\simulated-device>
```

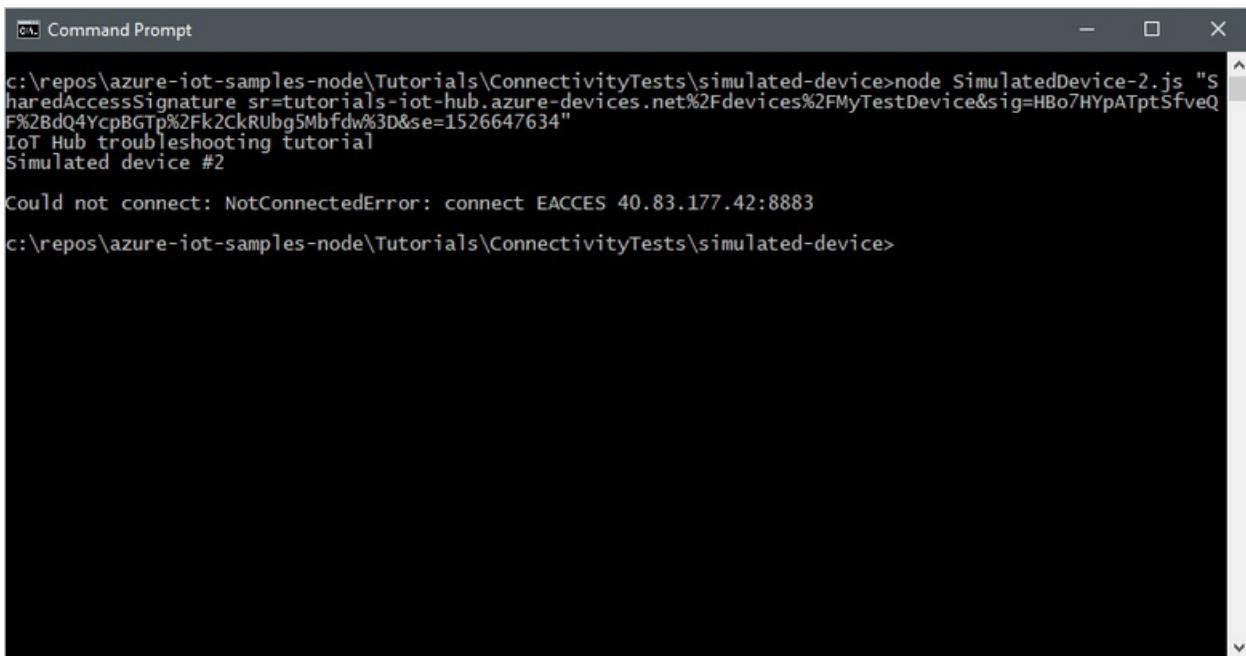
You've now successfully authenticated from a device using a test SAS token generated by a CLI command. The **SimulatedDevice-2.js** file includes sample code that shows you how to generate a SAS token in code.

Protocols

A device can use any of the following protocols to connect to your IoT hub:

PROTOCOL	OUTBOUND PORT
MQTT	8883
MQTT over WebSockets	443
AMQP	5671
AMQP over WebSockets	443
HTTPS	443

If the outbound port is blocked by a firewall, the device can't connect:



```
c:\repos\azure-iot-samples-node\Tutorials\ConnectivityTests\simulated-device>node SimulatedDevice-2.js "S
haredAccessSignature sr=tutorials-iot-hub.azure-devices.net%2Fdevices%2FMyTestDevice&sig=HBo7HYpATptSfveQ
F%2BdQ4YcpBGTp%2Fk2CkRUBg5Mbfdw%3D&se=1526647634"
IoT Hub troubleshooting tutorial
Simulated device #2

Could not connect: NotConnectedError: connect EACCES 40.83.177.42:8883
c:\repos\azure-iot-samples-node\Tutorials\ConnectivityTests\simulated-device>
```

Check device-to-cloud connectivity

After a device connects, it typically tries to send telemetry to your IoT hub. This section shows you how you can verify that the telemetry sent by the device reaches your hub.

First, retrieve the current connection string for your simulated device using the following command:

```
az iot hub device-identity show-connection-string --device-id MyTestDevice --output table --hub-name
{YourIoTHubName}
```

To run a simulated device that sends messages, navigate to the **iot-hub\Tutorials\ConnectivityTests\simulated-device** folder in the code you downloaded.

In the terminal window, run the following commands to install the required libraries and run the simulated device application:

```
npm install
node SimulatedDevice-3.js "{your device connection string}"
```

The terminal window displays information as it sends telemetry to your hub:

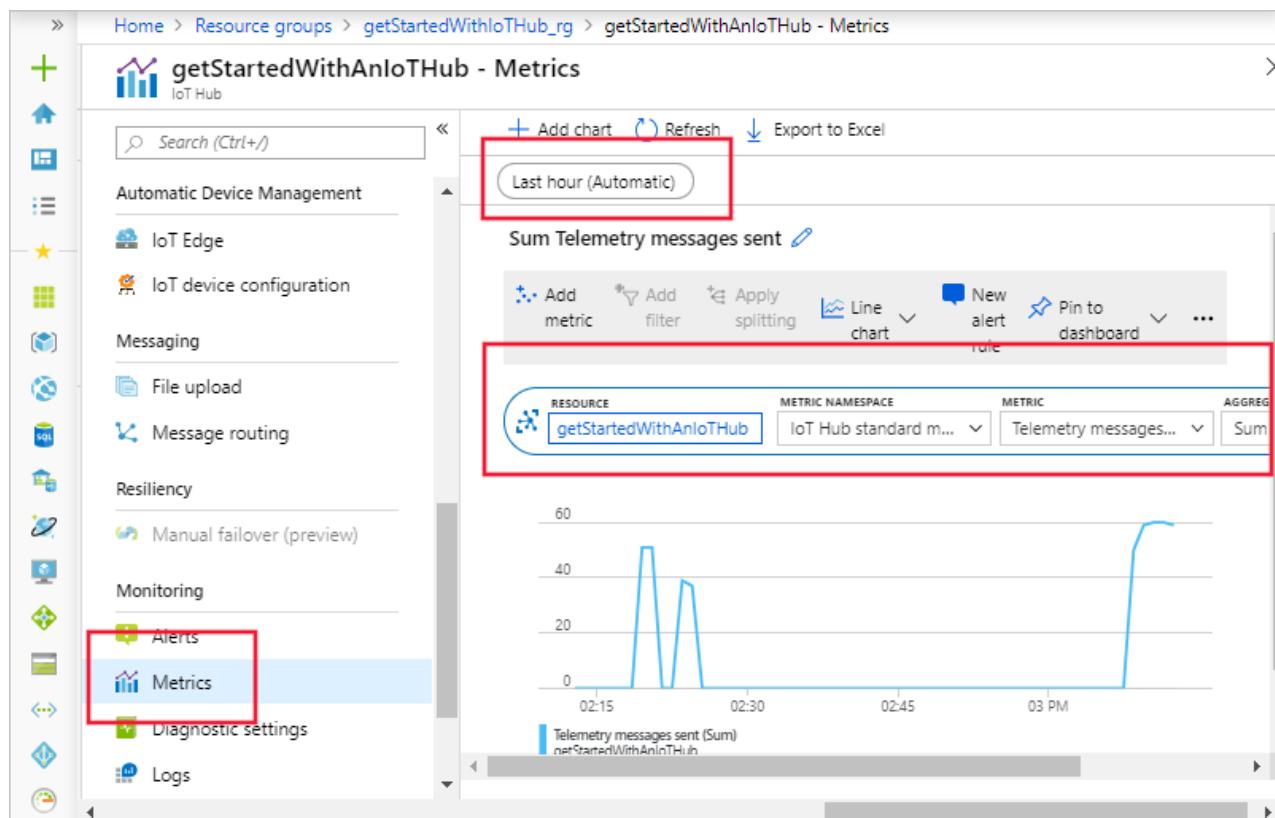
```

Command Prompt - node SimulatedDevice-3.js "HostName=tutorials-iot-hub.azure-devices.net;DeviceId=MyTestDevice;SharedAccessKey=MzNkZjFiOWNiMDZiMTYwMjFjNTAZZmu0"
c:\repos\azure-iot-samples-node\Tutorials\ConnectivityTests\simulated-device>node SimulatedDevice-3.js "HostName=tutorials-iot-hub.azure-devices.net;DeviceId=MyTestDevice;SharedAccessKey=MzNkZjFiOWNiMDZiMTYwMjFjNTAZZmu0"
IoT Hub troubleshooting tutorial
Simulated device #3

Sending message: {"temperature":29.945783468818455,"humidity":66.37628382483341}
Twin: Received desired properties:
{"$version":1}
Twin: Sent reported properties
Send telemetry status: MessageEnqueued
Sending message: {"temperature":30.189002299500295,"humidity":68.12187747761465}
Send telemetry status: MessageEnqueued
Sending message: {"temperature":28.79742406336066,"humidity":74.68073535919167}
Send telemetry status: MessageEnqueued
Sending message: {"temperature":29.337230090760432,"humidity":79.67996039914175}
Send telemetry status: MessageEnqueued
Sending message: {"temperature":28.0769178246929,"humidity":77.4532254513089}
Send telemetry status: MessageEnqueued
Sending message: {"temperature":34.14997607571426,"humidity":61.07608556471763}
Send telemetry status: MessageEnqueued
Sending message: {"temperature":23.68493830589282,"humidity":68.47271270904707}
Send telemetry status: MessageEnqueued
Sending message: {"temperature":33.23827056057915,"humidity":79.32459475445849}
Send telemetry status: MessageEnqueued
Sending message: {"temperature":27.338240267021146,"humidity":78.95655528128572}
Send telemetry status: MessageEnqueued

```

You can use **Metrics** in the portal to verify that the telemetry messages are reaching your IoT hub. Select your IoT hub in the **Resource** drop-down, select **Telemetry messages sent** as the metric, and set the time range to **Past hour**. The chart shows the aggregate count of messages sent by the simulated device:



It takes a few minutes for the metrics to become available after you start the simulated device.

Check cloud-to-device connectivity

This section shows how you can make a test direct method call to a device to check cloud-to-device connectivity. You run a simulated device on your development machine to listen for direct method calls from your hub.

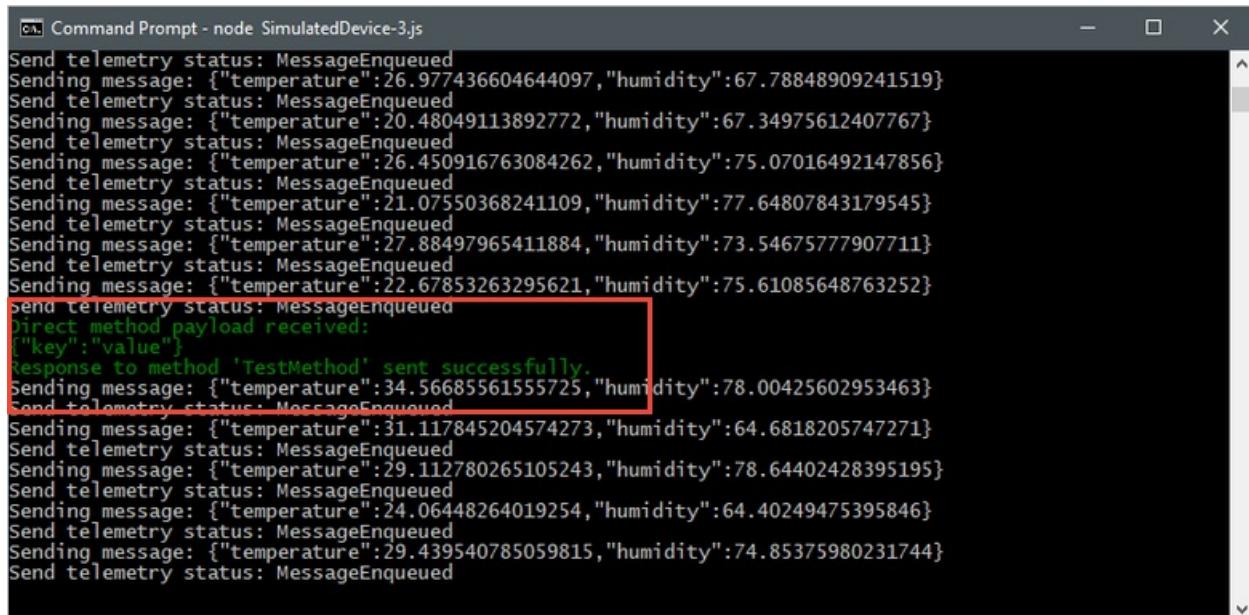
In a terminal window, use the following command to run the simulated device application:

```
node SimulatedDevice-3.js "{your device connection string}"
```

Use a CLI command to call a direct method on the device:

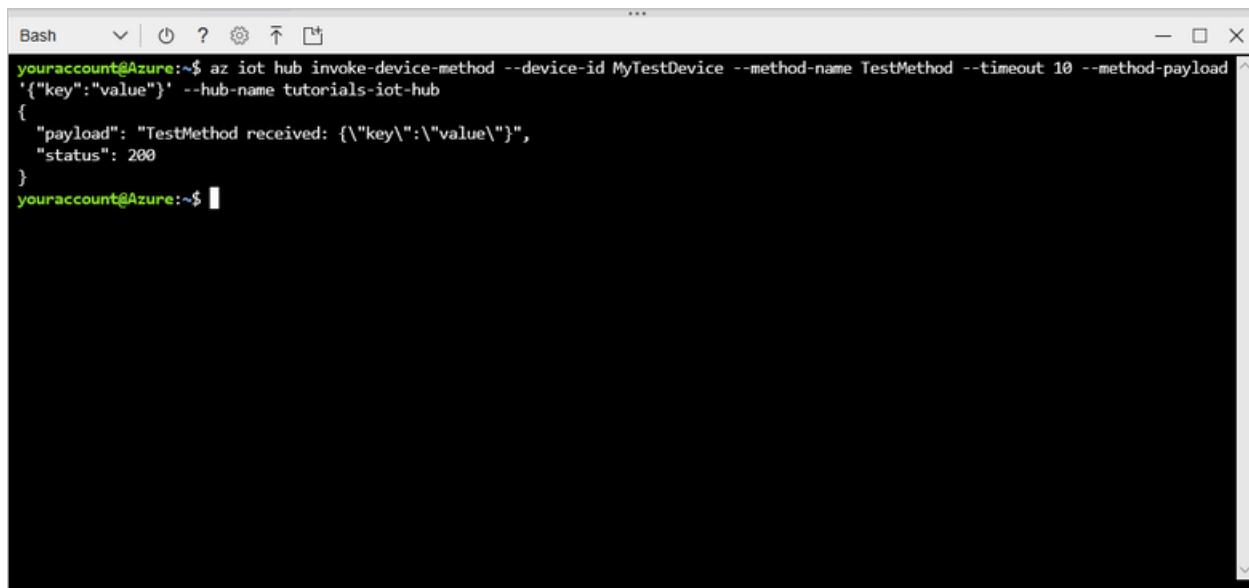
```
az iot hub invoke-device-method --device-id MyTestDevice --method-name TestMethod --timeout 10 --method-payload '{"key":"value"}' --hub-name {YourIoTHubName}
```

The simulated device prints a message to the console when it receives a direct method call:



```
Send telemetry status: MessageEnqueued
Sending message: {"temperature":26.977436604644097,"humidity":67.78848909241519}
Send telemetry status: MessageEnqueued
Sending message: {"temperature":20.48049113892772,"humidity":67.34975612407767}
Send telemetry status: MessageEnqueued
Sending message: {"temperature":26.450916763084262,"humidity":75.07016492147856}
Send telemetry status: MessageEnqueued
Sending message: {"temperature":21.07550368241109,"humidity":77.64807843179545}
Send telemetry status: MessageEnqueued
Sending message: {"temperature":27.88497965411884,"humidity":73.54675777907711}
Send telemetry status: MessageEnqueued
Sending message: {"temperature":22.67853263295621,"humidity":75.61085648763252}
Send telemetry status: MessageEnqueued
Direct method payload received:
["key":"value"]
Response to method 'TestMethod' sent successfully.
Sending message: {"temperature":34.56685561555725,"humidity":78.00425602953463}
Send telemetry status: MessageEnqueued
Sending message: {"temperature":31.117845204574273,"humidity":64.6818205747271}
Send telemetry status: MessageEnqueued
Sending message: {"temperature":29.112780265105243,"humidity":78.64402428395195}
Send telemetry status: MessageEnqueued
Sending message: {"temperature":24.06448264019254,"humidity":64.40249475395846}
Send telemetry status: MessageEnqueued
Sending message: {"temperature":29.439540785059815,"humidity":74.85375980231744}
Send telemetry status: MessageEnqueued
```

When the simulated device successfully receives the direct method call, it sends an acknowledgement back to the hub:



```
youraccount@Azure:~$ az iot hub invoke-device-method --device-id MyTestDevice --method-name TestMethod --timeout 10 --method-payload '{"key":"value"}' --hub-name tutorials-iot-hub
{
  "payload": "TestMethod received: {\"key\":\"value\"}",
  "status": 200
}
youraccount@Azure:~$
```

Check twin synchronization

Devices use twins to synchronize state between the device and the hub. In this section, you use CLI commands to send *desired properties* to a device and read the *reported properties* sent by the device.

The simulated device you use in this section sends reported properties to the hub whenever it starts up, and prints desired properties to the console whenever it receives them.

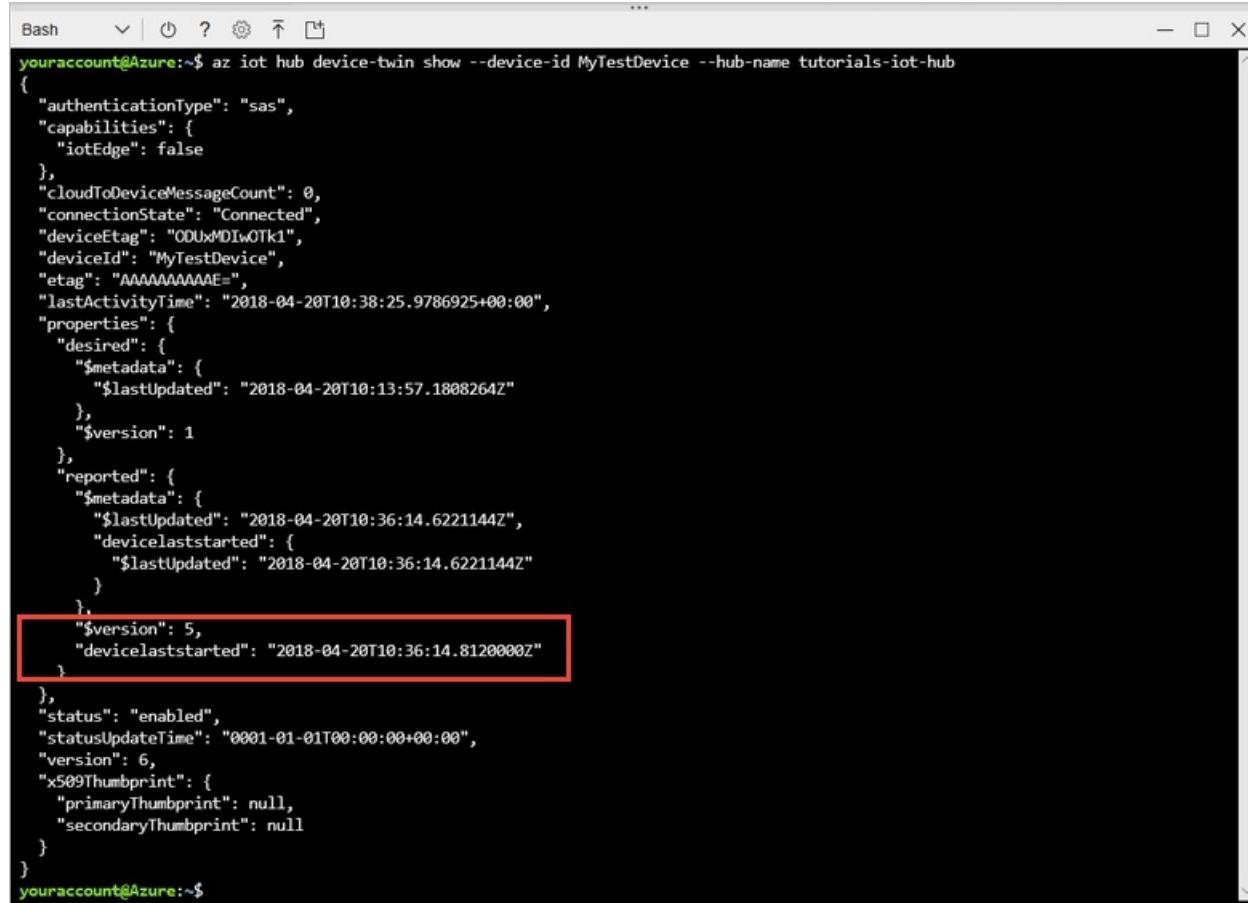
In a terminal window, use the following command to run the simulated device application:

```
node SimulatedDevice-3.js "{your device connection string}"
```

To verify that the hub received the reported properties from the device, use the following CLI command:

```
az iot hub device-twin show --device-id MyTestDevice --hub-name {YourIoTHubName}
```

In the output from the command, you can see the **devicelaststarted** property in the reported properties section. This property shows the date and time you last started the simulated device.

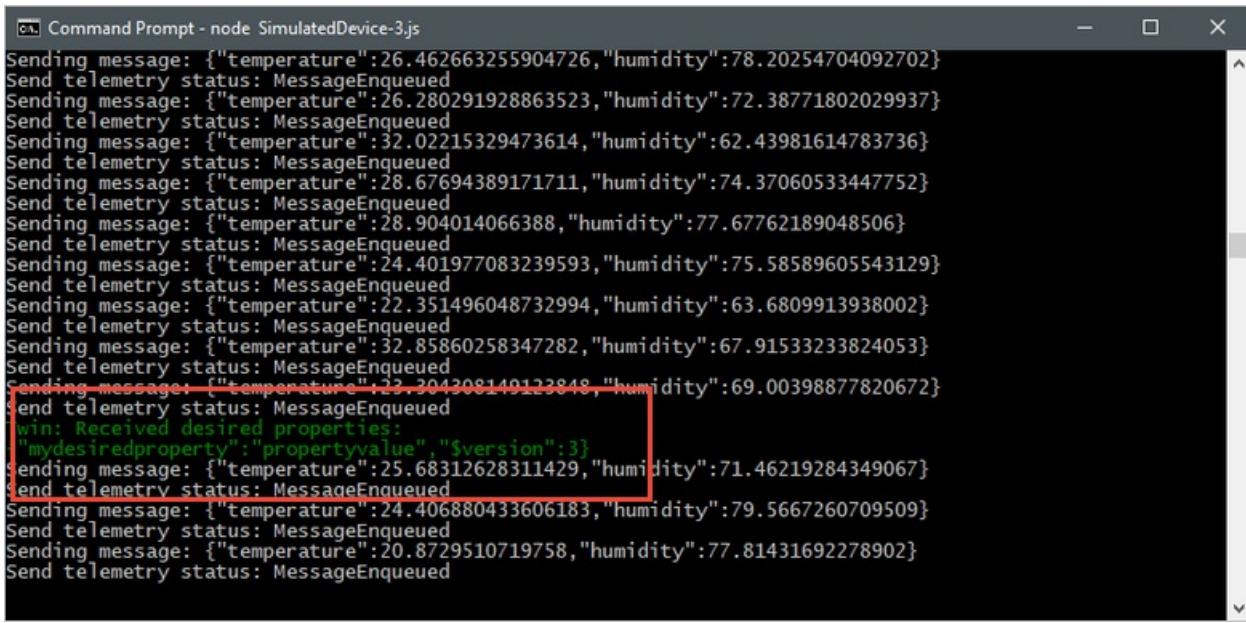


```
youraccount@Azure:~$ az iot hub device-twin show --device-id MyTestDevice --hub-name tutorials-iot-hub
{
  "authenticationType": "sas",
  "capabilities": {
    "iotEdge": false
  },
  "cloudToDeviceMessageCount": 0,
  "connectionState": "Connected",
  "deviceEtag": "ODUxMDIwOTk1",
  "deviceId": "MyTestDevice",
  "etag": "AAAAAAAAAAE=",
  "lastActivityTime": "2018-04-20T10:38:25.9786925+00:00",
  "properties": {
    "desired": {
      "$metadata": {
        "$lastUpdated": "2018-04-20T10:13:57.1808264Z"
      },
      "$version": 1
    },
    "reported": {
      "$metadata": {
        "$lastUpdated": "2018-04-20T10:36:14.6221144Z",
        "devicelaststarted": {
          "$lastUpdated": "2018-04-20T10:36:14.6221144Z"
        }
      },
      "$version": 5,
      "devicelaststarted": "2018-04-20T10:36:14.8120000Z"
    }
  },
  "status": "enabled",
  "statusUpdateTime": "0001-01-01T00:00:00+00:00",
  "version": 6,
  "x509Thumbprint": {
    "primaryThumbprint": null,
    "secondaryThumbprint": null
  }
}
youraccount@Azure:~$
```

To verify that the hub can send desired property values to the device, use the following CLI command:

```
az iot hub device-twin update --set properties.desired='{"mydesiredproperty":"propertyvalue"}' --device-id MyTestDevice --hub-name {YourIoTHubName}
```

The simulated device prints a message when it receives a desired property update from the hub:



```
Command Prompt - node SimulatedDevice-3.js
Sending message: {"temperature":26.462663255904726,"humidity":78.20254704092702}
Send telemetry status: MessageEnqueued
Sending message: {"temperature":26.280291928863523,"humidity":72.38771802029937}
Send telemetry status: MessageEnqueued
Sending message: {"temperature":32.02215329473614,"humidity":62.43981614783736}
Send telemetry status: MessageEnqueued
Sending message: {"temperature":28.67694389171711,"humidity":74.37060533447752}
Send telemetry status: MessageEnqueued
Sending message: {"temperature":28.904014066388,"humidity":77.67762189048506}
Send telemetry status: MessageEnqueued
Sending message: {"temperature":24.401977083239593,"humidity":75.58589605543129}
Send telemetry status: MessageEnqueued
Sending message: {"temperature":22.351496048732994,"humidity":63.6809913938002}
Send telemetry status: MessageEnqueued
Sending message: {"temperature":32.85860258347282,"humidity":67.91533233824053}
Send telemetry status: MessageEnqueued
Sending message: {"temperature":23.304308149123848,"humidity":69.00398877820672}
Send telemetry status: MessageEnqueued
win: Received desired properties:
"mydesiredproperty": "propertyvalue", "$version": 3
Sending message: {"temperature":25.68312628311429,"humidity":71.46219284349067}
Send telemetry status: MessageEnqueued
Sending message: {"temperature":24.406880433606183,"humidity":79.5667260709509}
Send telemetry status: MessageEnqueued
Sending message: {"temperature":20.8729510719758,"humidity":77.81431692278902}
Send telemetry status: MessageEnqueued
```

In addition to receiving desired property changes as they're made, the simulated device automatically checks for desired properties when it starts up.

Clean up resources

If you don't need the IoT hub any longer, delete it and the resource group in the portal. To do so, select the **tutorials-iot-hub-rg** resource group that contains your IoT hub and click **Delete**.

Next steps

In this tutorial, you've seen how to check your device keys, check device-to-cloud connectivity, check cloud-to-device connectivity, and check device twin synchronization. To learn more about how to monitor your IoT hub, visit the how-to article for IoT Hub monitoring.

[Monitor with diagnostics](#)

Overview of device management with IoT Hub

10/2/2018 • 5 minutes to read

Azure IoT Hub provides the features and an extensibility model that enable device and back-end developers to build robust device management solutions. Devices range from constrained sensors and single purpose microcontrollers, to powerful gateways that route communications for groups of devices. In addition, the use cases and requirements for IoT operators vary significantly across industries. Despite this variation, device management with IoT Hub provides the capabilities, patterns, and code libraries to cater to a diverse set of devices and end users.

NOTE

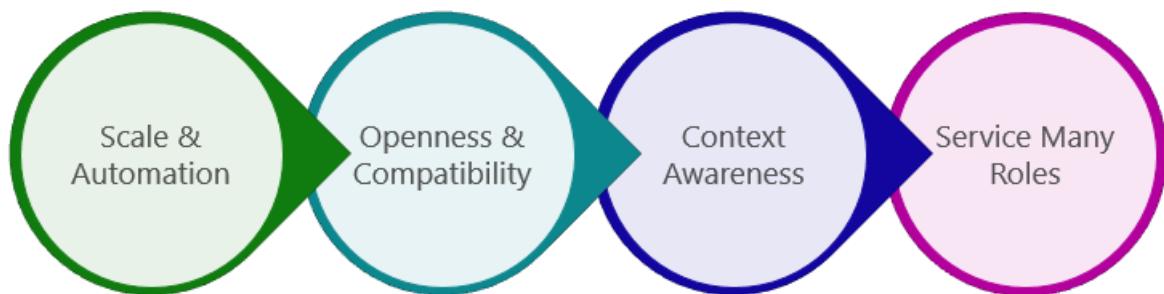
Some of the features mentioned in this article, like cloud-to-device messaging, device twins, and device management, are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

A crucial part of creating a successful enterprise IoT solution is to provide a strategy for how operators handle the ongoing management of their collection of devices. IoT operators require simple and reliable tools and applications that enable them to focus on the more strategic aspects of their jobs. This article provides:

- A brief overview of Azure IoT Hub approach to device management.
- A description of common device management principles.
- A description of the device lifecycle.
- An overview of common device management patterns.

Device management principles

IoT brings with it a unique set of device management challenges and every enterprise-class solution must address the following principles:

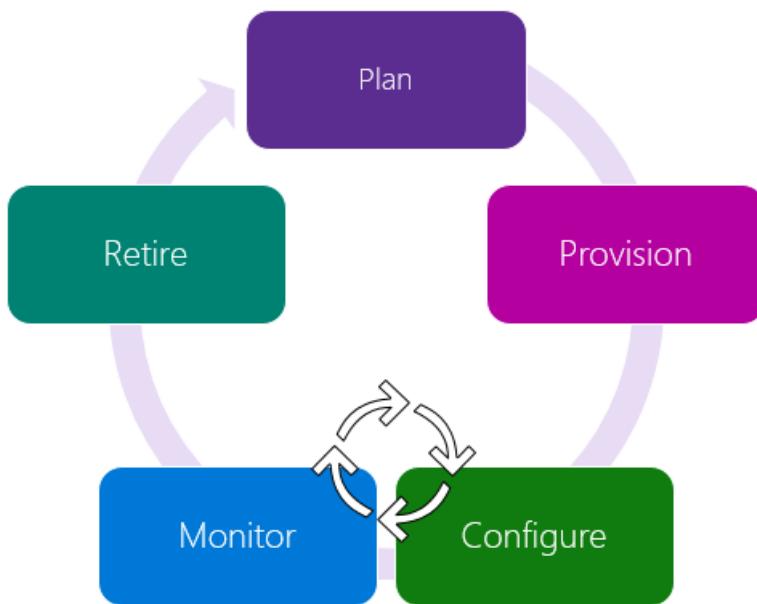


- **Scale and automation:** IoT solutions require simple tools that can automate routine tasks and enable a relatively small operations staff to manage millions of devices. Day-to-day, operators expect to handle device operations remotely, in bulk, and to only be alerted when issues arise that require their direct attention.
- **Openness and compatibility:** The device ecosystem is extraordinarily diverse. Management tools must be tailored to accommodate a multitude of device classes, platforms, and protocols. Operators must be able to support many types of devices, from the most constrained embedded single-process chips, to powerful and fully functional computers.

- **Context awareness:** IoT environments are dynamic and ever-changing. Service reliability is paramount. Device management operations must take into account the following factors to ensure that maintenance downtime doesn't affect critical business operations or create dangerous conditions:
 - SLA maintenance windows
 - Network and power states
 - In-use conditions
 - Device geolocation
- **Service many roles:** Support for the unique workflows and processes of IoT operations roles is crucial. The operations staff must work harmoniously with the given constraints of internal IT departments. They must also find sustainable ways to surface realtime device operations information to supervisors and other business managerial roles.

Device lifecycle

There is a set of general device management stages that are common to all enterprise IoT projects. In Azure IoT, there are five stages within the device lifecycle:



Within each of these five stages, there are several device operator requirements that should be fulfilled to provide a complete solution:

- **Plan:** Enable operators to create a device metadata scheme that enables them to easily and accurately query for, and target a group of devices for bulk management operations. You can use the device twin to store this device metadata in the form of tags and properties.

Further reading:

- [Get started with device twins](#)
- [Understand device twins](#)
- [How to use device twin properties](#)
- [Best practices for device configuration within an IoT solution](#)
- **Provision:** Securely provision new devices to IoT Hub and enable operators to immediately discover device capabilities. Use the IoT Hub identity registry to create flexible device identities and credentials, and perform this operation in bulk by using a job. Build devices to report their capabilities and conditions through device properties in the device twin.

Further reading:

- [Manage device identities](#)
- [Bulk management of device identities](#)
- [How to use device twin properties](#)
- [Best practices for device configuration within an IoT solution](#)
- [Azure IoT Hub Device Provisioning Service](#)

- **Configure:** Facilitate bulk configuration changes and firmware updates to devices while maintaining both health and security. Perform these device management operations in bulk by using desired properties or with direct methods and broadcast jobs.

Further reading:

- [How to use device twin properties](#)
- [Configure and monitor IoT devices at scale](#)
- [Best practices for device configuration within an IoT solution](#)

- **Monitor:** Monitor overall device collection health, the status of ongoing operations, and alert operators to issues that might require their attention. Apply the device twin to allow devices to report realtime operating conditions and status of update operations. Build powerful dashboard reports that surface the most immediate issues by using device twin queries.

Further reading:

- [How to use device twin properties](#)
- [IoT Hub query language for device twins, jobs, and message routing](#)
- [Configure and monitor IoT devices at scale](#)
- [Best practices for device configuration within an IoT solution](#)

- **Retire:** Replace or decommission devices after a failure, upgrade cycle, or at the end of the service lifetime. Use the device twin to maintain device info if the physical device is being replaced, or archived if being retired. Use the IoT Hub identity registry for securely revoking device identities and credentials.

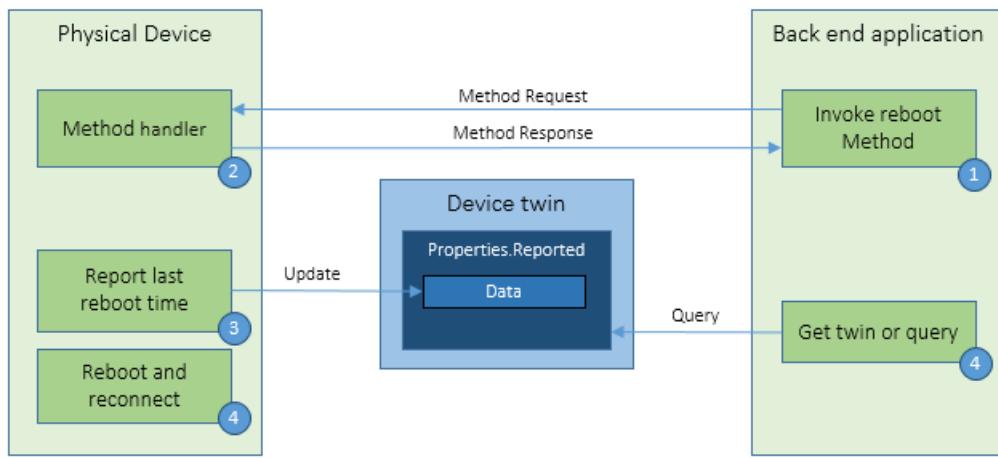
Further reading:

- [How to use device twin properties](#)
- [Manage device identities](#)

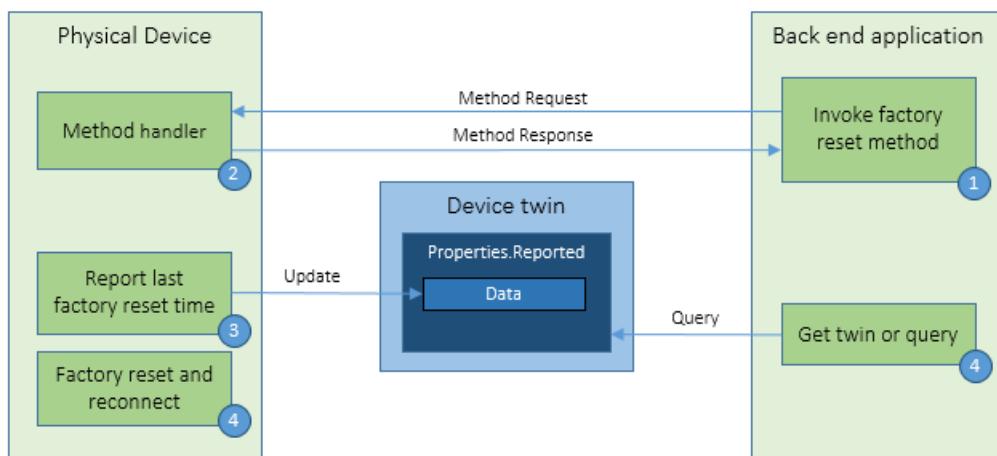
Device management patterns

IoT Hub enables the following set of device management patterns. The [device management tutorials](#) show you in more detail how to extend these patterns to fit your exact scenario and how to design new patterns based on these core templates.

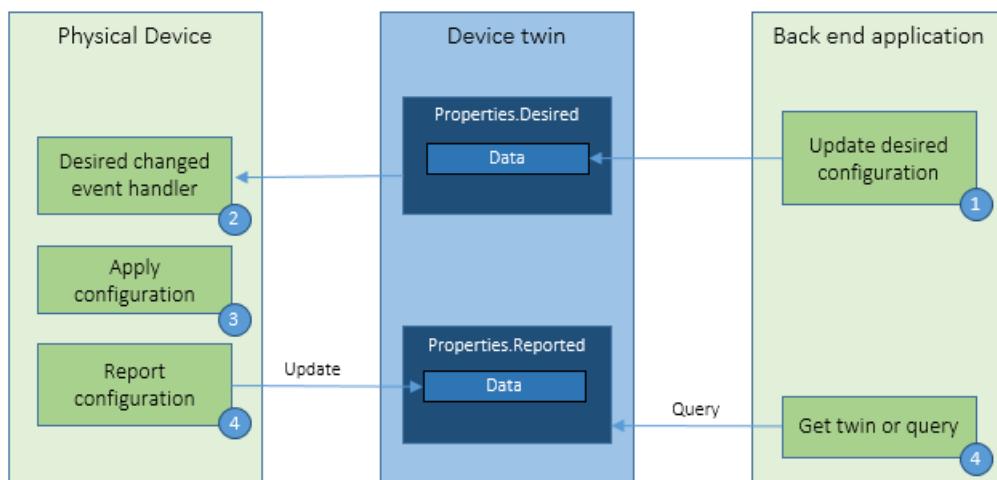
- **Reboot:** The back-end app informs the device through a direct method that it has initiated a reboot. The device uses the reported properties to update the reboot status of the device.



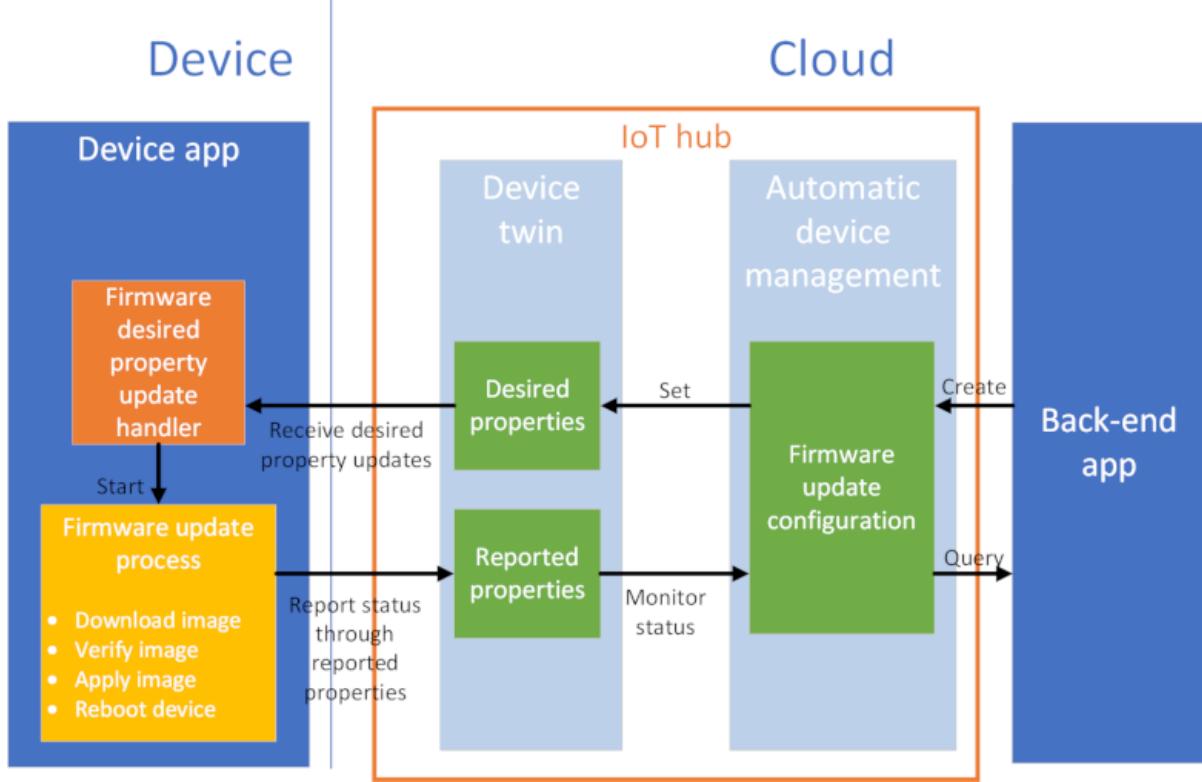
- **Factory Reset:** The back-end app informs the device through a direct method that it has initiated a factory reset. The device uses the reported properties to update the factory reset status of the device.



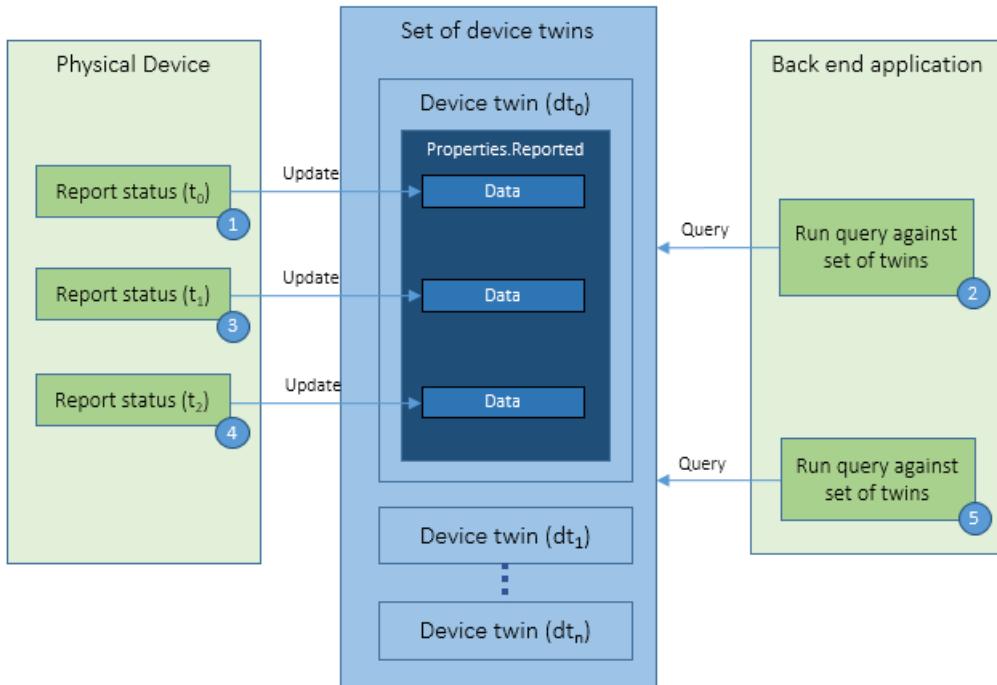
- **Configuration:** The back-end app uses the desired properties to configure software running on the device. The device uses the reported properties to update configuration status of the device.



- **Firmware Update:** The back-end app uses an automatic device management configuration to select the devices to receive the update, to tell the devices where to find the update, and to monitor the update process. The device initiates a multistep process to download, verify, and apply the firmware image, and then reboot the device before reconnecting to the IoT Hub service. Throughout the multistep process, the device uses the reported properties to update the progress and status of the device.



- **Reporting progress and status:** The solution back end runs device twin queries, across a set of devices, to report on the status and progress of actions running on the devices.



Next Steps

The capabilities, patterns, and code libraries that IoT Hub provides for device management, enable you to create IoT applications that fulfill enterprise IoT operator requirements within each device lifecycle stage.

To continue learning about the device management features in IoT Hub, see the [Get started with device management](#) tutorial.

Connecting IoT Devices to Azure: IoT Hub and Event Hubs

2/22/2019 • 2 minutes to read

Azure provides services specifically developed for diverse types of connectivity and communication to help you connect your data to the power of the cloud. Both Azure IoT Hub and Azure Event Hubs are cloud services that can ingest large amounts of data and process or store that data for business insights. The two services are similar in that they both support ingestion of data with low latency and high reliability, but they are designed for different purposes. IoT Hub was developed to address the unique requirements of connecting IoT devices to the Azure cloud while Event Hubs was designed for big data streaming. Microsoft recommends using Azure IoT Hub to connect IoT devices to Azure.

Azure IoT Hub is the cloud gateway that connects IoT devices to gather data and drive business insights and automation. In addition, IoT Hub includes features that enrich the relationship between your devices and your backend systems. Bi-directional communication capabilities mean that while you receive data from devices you can also send commands and policies back to devices. For example, use cloud-to-device messaging to update properties or invoke device management actions. Cloud-to-device communication also enables you to send cloud intelligence to your edge devices with Azure IoT Edge. The unique device-level identity provided by IoT Hub helps better secure your IoT solution from potential attacks.

[Azure Event Hubs](#) is the big data streaming service of Azure. It is designed for high throughput data streaming scenarios where customers may send billions of requests per day. Event Hubs uses a partitioned consumer model to scale out your stream and is integrated into the big data and analytics services of Azure including Databricks, Stream Analytics, ADLS, and HDInsight. With features like Event Hubs Capture and Auto-Inflate, this service is designed to support your big data apps and solutions. Additionally, IoT Hub uses Event Hubs for its telemetry flow path, so your IoT solution also benefits from the tremendous power of Event Hubs.

To summarize, both solutions are designed for data ingestion at a massive scale. Only IoT Hub provides the rich IoT-specific capabilities that are designed for you to maximize the business value of connecting your IoT devices to the Azure cloud. If your IoT journey is just beginning, starting with IoT Hub to support your data ingestion scenarios will assure that you have instant access to the full-featured IoT capabilities once your business and technical needs require them.

The following table provides details about how the two tiers of IoT Hub compare to Event Hubs when you're evaluating them for IoT capabilities. For more information about the standard and basic tiers of IoT Hub, see [How to choose the right IoT Hub tier](#).

IOT CAPABILITY	IOT HUB STANDARD TIER	IOT HUB BASIC TIER	EVENT HUBS
Device-to-cloud messaging	✓	✓	✓
Protocols: HTTPS, AMQP, AMQP over webSockets	✓	✓	✓
Protocols: MQTT, MQTT over webSockets	✓	✓	
Per-device identity	✓	✓	

IOT CAPABILITY	IOT HUB STANDARD TIER	IOT HUB BASIC TIER	EVENT HUBS
File upload from devices	✓	✓	
Device Provisioning Service	✓	✓	
Cloud-to-device messaging	✓		
Device twin and device management	✓		
Device streams (preview)	✓		
IoT Edge	✓		

Even if the only use case is device-to-cloud data ingestion, we highly recommend using IoT Hub as it provides a service that is designed for IoT device connectivity.

Next steps

To further explore the capabilities of IoT Hub, see the [IoT Hub developer guide](#).

Choose the right IoT Hub tier for your solution

3/6/2019 • 5 minutes to read

Every IoT solution is different, so Azure IoT Hub offers several options based on pricing and scale. This article is meant to help you evaluate your IoT Hub needs. For pricing information about IoT Hub tiers, see [IoT Hub pricing](#).

To decide which IoT Hub tier is right for your solution, ask yourself two questions:

What features do I plan to use? Azure IoT Hub offers two tiers, basic and standard, that differ in the number of features they support. If your IoT solution is based around collecting data from devices and analyzing it centrally, then the basic tier is probably right for you. If you want to use more advanced configurations to control IoT devices remotely or distribute some of your workloads onto the devices themselves, then you should consider the standard tier. For a detailed breakdown of which features are included in each tier continue to [Basic and standard tiers](#).

How much data do I plan to move daily? Each IoT Hub tier is available in three sizes, based around how much data throughput they can handle in any given day. These sizes are numerically identified as 1, 2, and 3. For example, each unit of a level 1 IoT hub can handle 400 thousand messages a day, while a level 3 unit can handle 300 million. For more details about the data guidelines, continue to [Message throughput](#).

Basic and standard tiers

The standard tier of IoT Hub enables all features, and is required for any IoT solutions that want to make use of the bi-directional communication capabilities. The basic tier enables a subset of the features and is intended for IoT solutions that only need uni-directional communication from devices to the cloud. Both tiers offer the same security and authentication features.

Only one type of [edition](#) within a tier can be chosen per IoT Hub. For example, you can create an IoT Hub with multiple units of S1, but not with a mix of units from different editions, such as S1 and B3, or S1 and S2.

CAPABILITY	BASIC TIER	FREE/STANDARD TIER
Device-to-cloud telemetry	Yes	Yes
Per-device identity	Yes	Yes
Message routing and Event Grid integration	Yes	Yes
HTTP, AMQP, and MQTT protocols	Yes	Yes
Device Provisioning Service	Yes	Yes
Monitoring and diagnostics	Yes	Yes

CAPABILITY	BASIC TIER	FREE/STANDARD TIER
Cloud-to-device messaging		Yes
Device twins, Module twins, and Device management		Yes
Device streams (preview)		Yes
Azure IoT Edge		Yes

IoT Hub also offers a free tier that is meant for testing and evaluation. It has all the capabilities of the standard tier, but limited messaging allowances. You cannot upgrade from the free tier to either basic or standard.

Partitions

Azure IoT Hubs contain many core components of [Azure Event Hubs](#), including [Partitions](#). Event streams for IoT Hubs are generally populated with incoming telemetry data that is reported by various IoT devices. The partitioning of the event stream is used to reduce contentions that occur when concurrently reading and writing to event streams.

The partition limit is chosen when IoT Hub is created, and cannot be changed. The maximum partition limit for basic tier IoT Hub and standard tier IoT Hub is 32. Most IoT hubs only need 4 partitions. For more information on determining the partitions, see the Event Hubs FAQ [How many partitions do I need?](#)

Tier upgrade

Once you create your IoT hub, you can upgrade from the basic tier to the standard tier without interrupting your existing operations. For more information, see [How to upgrade your IoT hub](#).

The partition configuration remains unchanged when you migrate from basic tier to standard tier.

IoT Hub REST APIs

The difference in supported capabilities between the basic and standard tiers of IoT Hub means that some API calls do not work with basic tier hubs. The following table shows which APIs are available:

API	BASIC TIER	FREE/STANDARD TIER
Delete device	Yes	Yes
Get device	Yes	Yes
Delete module	Yes	Yes
Get module	Yes	Yes
Get registry statistics	Yes	Yes
Get services statistics	Yes	Yes

API	BASIC TIER	FREE/STANDARD TIER
Create Or Update Device	Yes	Yes
Put module	Yes	Yes
Query IoT Hub	Yes	Yes
Query modules	Yes	Yes
Create file upload SAS URI	Yes	Yes
Receive device bound notification	Yes	Yes
Send device event	Yes	Yes
Send module event	Yes	Yes
Update file upload status	Yes	Yes
Bulk device operation	Yes, except for IoT Edge capabilities	Yes
Purge command queue		Yes
Get device twin		Yes
Get module twin		Yes
Invoke device method		Yes
Update device twin		Yes
Update module twin		Yes
Abandon device bound notification		Yes
Complete device bound notification		Yes
Cancel job		Yes
Create job		Yes
Get job		Yes
Query jobs		Yes

Message throughput

The best way to size an IoT Hub solution is to evaluate the traffic on a per-unit basis. In particular, consider the required peak throughput for the following categories of operations:

- Device-to-cloud messages
- Cloud-to-device messages
- Identity registry operations

Traffic is measured on a per-unit basis, not per hub. A level 1 or 2 IoT Hub instance can have as many as 200 units associated with it. A level 3 IoT Hub instance can have up to 10 units. Once you create your IoT hub you can change the number of units or move between the 1, 2, and 3 sizes within a specific tier without interrupting your existing operations. For more information, see [How to upgrade your IoT Hub](#).

As an example of each tier's traffic capabilities, device-to-cloud messages follow these sustained throughput guidelines:

TIER	SUSTAINED THROUGHPUT	SUSTAINED SEND RATE
B1, S1	Up to 1111 KB/minute per unit (1.5 GB/day/unit)	Average of 278 messages/minute per unit (400,000 messages/day per unit)
B2, S2	Up to 16 MB/minute per unit (22.8 GB/day/unit)	Average of 4,167 messages/minute per unit (6 million messages/day per unit)
B3, S3	Up to 814 MB/minute per unit (1144.4 GB/day/unit)	Average of 208,333 messages/minute per unit (300 million messages/day per unit)

In addition to this throughput information, see [IoT Hub quotas and throttles](#) and design your solution accordingly.

Identity registry operation throughput

IoT Hub identity registry operations are not supposed to be run-time operations, as they are mostly related to device provisioning.

For specific burst performance numbers, see [IoT Hub quotas and throttles](#).

Auto-scale

If you are approaching the allowed message limit on your IoT Hub, you can use these [steps to automatically scale](#) to increment an IoT Hub unit in the same IoT Hub tier.

Sharding

While a single IoT hub can scale to millions of devices, sometimes your solution requires specific performance characteristics that a single IoT hub cannot guarantee. In that case, you can partition your devices across multiple IoT hubs. Multiple IoT hubs smooth traffic bursts and obtain the required throughput or operation rates that are required.

Next steps

- For more information about IoT Hub capabilities and performance details, see [IoT Hub pricing](#) or [IoT Hub quotas and throttles](#).
- To change your IoT Hub tier, follow the steps in [Upgrade your IoT hub](#).

IoT Hub high availability and disaster recovery

2/27/2019 • 10 minutes to read

As a first step towards implementing a resilient IoT solution, architects, developers, and business owners must define the uptime goals for the solutions they're building. These goals can be defined primarily based on specific business objectives for each scenario. In this context, the article [Azure Business Continuity Technical Guidance](#) describes a general framework to help you think about business continuity and disaster recovery. The [Disaster recovery and high availability for Azure applications](#) paper provides architecture guidance on strategies for Azure applications to achieve High Availability (HA) and Disaster Recovery (DR).

This article discusses the HA and DR features offered specifically by the IoT Hub service. The broad areas discussed in this article are:

- Intra-region HA
- Cross region DR
- Achieving cross region HA

Depending on the uptime goals you define for your IoT solutions, you should determine which of the options outlined below best suit your business objectives. Incorporating any of these HA/DR alternatives into your IoT solution requires a careful evaluation of the trade-offs between the:

- Level of resiliency you require
- Implementation and maintenance complexity
- COGS impact

Intra-region HA

The IoT Hub service provides intra-region HA by implementing redundancies in almost all layers of the service. The [SLA published by the IoT Hub service](#) is achieved by making use of these redundancies. No additional work is required by the developers of an IoT solution to take advantage of these HA features. Although IoT Hub offers a reasonably high uptime guarantee, transient failures can still be expected as with any distributed computing platform. If you're just getting started with migrating your solutions to the cloud from an on-premise solution, your focus needs to shift from optimizing "mean time between failures" to "mean time to recover". In other words, transient failures are to be considered normal while operating with the cloud in the mix. Appropriate [retry policies](#) must be built in to the components interacting with a cloud application to deal with transient failures.

NOTE

Some Azure services also provide additional layers of availability within a region by integrating with [Availability Zones \(AZs\)](#). AZs are currently not supported by the IoT Hub service.

Cross region DR

There could be some rare situations when a datacenter experiences extended outages due to power failures or other failures involving physical assets. Such events are rare during which the intra region HA capability described above may not always help. IoT Hub provides multiple solutions for recovering from such extended outages.

The recovery options available to customers in such a situation are "Microsoft-initiated failover" and "manual failover". The fundamental difference between the two is that Microsoft initiates the former and the user initiates the latter. Also, manual failover provides a lower recovery time objective (RTO) compared to the Microsoft-initiated

failover option. The specific RTOs offered with each option are discussed in the sections below. When either of these options to perform failover of an IoT hub from its primary region is exercised, the hub becomes fully functional in the corresponding [Azure geo-paired region](#).

Both these failover options offer the following recovery point objectives (RPOs):

DATA TYPE	RECOVERY POINT OBJECTIVES (RPO)
Identity registry	0-5 mins data loss
Device twin data	0-5 mins data loss
Cloud-to-device messages ¹	0-5 mins data loss
Parent ¹ and device jobs	0-5 mins data loss
Device-to-cloud messages	All unread messages are lost
Operations monitoring messages	All unread messages are lost
Cloud-to-device feedback messages	All unread messages are lost

¹Cloud-to-device messages and parent jobs do not get recovered as a part of manual failover in the preview offering of this feature.

Once the failover operation for the IoT hub completes, all operations from the device and back-end applications are expected to continue working without requiring a manual intervention.

Caution

- The Event Hub-compatible name and endpoint of the IoT Hub built-in Events endpoint change after failover. When receiving telemetry messages from the built-in endpoint using either the event hub client or event processor host, you should [use the IoT hub connection string](#) to establish the connection. This ensures that your back-end applications continue to work without requiring manual intervention post failover. If you use the Event Hub-compatible name and endpoint in your back-end application directly, you will need to reconfigure your application by [fetching the new Event Hub-compatible name and endpoint](#) after failover to continue operations.
- After failover, the events emitted via Event Grid can be consumed via the same subscription(s) configured earlier as long as those Event Grid subscriptions continue to be available.
- When routing to blob storage, we recommend enlisting the blobs and then iterating over them, to ensure all containers are read without making any assumptions of partition. The partition range could potentially change during a Microsoft-initiated failover or manual failover. To learn how to enumerate the list of blobs see [routing to blob storage](#).

Microsoft-initiated failover

Microsoft-initiated failover is exercised by Microsoft in rare situations to failover all the IoT hubs from an affected region to the corresponding geo-paired region. This process is a default option (no way for users to opt out) and requires no intervention from the user. Microsoft reserves the right to make a determination of when this option will be exercised. This mechanism doesn't involve a user consent before the user's hub is failed over. Microsoft-initiated failover has a recovery time objective (RTO) of 2-26 hours.

The large RTO is because Microsoft must perform the failover operation on behalf of all the affected customers in that region. If you are running a less critical IoT solution that can sustain a downtime of roughly a day, it is ok for you to take a dependency on this option to satisfy the overall disaster recovery goals for your IoT solution. The total time for runtime operations to become fully operational once this process is triggered, is described in the

"Time to recover" section.

Manual failover (preview)

If your business uptime goals aren't satisfied by the RTO that Microsoft initiated failover provides, you should consider using manual failover to trigger the failover process yourself. The RTO using this option could be anywhere between 10 minutes to a couple of hours. The RTO is currently a function of the number of devices registered against the IoT hub instance being failed over. You can expect the RTO for a hub hosting approximately 100,000 devices to be in the ballpark of 15 minutes. The total time for runtime operations to become fully operational once this process is triggered, is described in the "Time to recover" section.

The manual failover option is always available for use irrespective of whether the primary region is experiencing downtime or not. Therefore, this option could potentially be used to perform planned failovers. One example usage of planned failovers is to perform periodic failover drills. A word of caution though is that a planned failover operation results in a downtime for the hub for the period defined by the RTO for this option, and also results in a data loss as defined by the RPO table above. You could consider setting up a test IoT hub instance to exercise the planned failover option periodically to gain confidence in your ability to get your end-to-end solutions up and running when a real disaster happens.

IMPORTANT

- Test drills should not be performed on IoT hubs that are being used in your production environments.
- Manual failover should not be used as a mechanism to permanently migrate your hub between the Azure geo paired regions. Doing so would cause an increased latency for the operations being performed against the hub from devices homed in the old primary region.
- Manual failover is currently in preview and is not available in the following Azure regions. East US, West US, North Europe, West Europe, Brazil South, South Central US.

Fallback

Failing back to the old primary region can be achieved by triggering the failover action another time. If the original failover operation was performed to recover from an extended outage in the original primary region, we recommended that the hub should be failed back to the original location once that location has recovered from the outage situation.

IMPORTANT

- Users are only allowed to perform 2 successful failover and 2 successful fallback operations per day.
- Back to back failover/fallback operations are not allowed. Users will have to wait for 1 hour between these operations.

Time to recover

While the FQDN (and therefore the connection string) of the IoT hub instance remains the same post failover, the underlying IP address will change. Therefore the overall time for the runtime operations being performed against your IoT hub instance to become fully operational after the failover process is triggered can be expressed using the following function.

Time to recover = RTO [10 min - 2 hours for manual failover | 2 - 26 hours for Microsoft-initiated failover] + DNS propagation delay + Time taken by the client application to refresh any cached IoT Hub IP address.

IMPORTANT

The IoT SDKs do not cache the IP address of the IoT hub. We recommend that user code interfacing with the SDKs should not cache the IP address of the IoT hub.

Achieve cross region HA

If your business uptime goals aren't satisfied by the RTO that either Microsoft-initiated failover or manual failover options provide, you should consider implementing a per-device automatic cross region failover mechanism. A complete treatment of deployment topologies in IoT solutions is outside the scope of this article. The article discusses the *regional failover* deployment model for the purpose of high availability and disaster recovery.

In a regional failover model, the solution back end runs primarily in one datacenter location. A secondary IoT hub and back end are deployed in another datacenter location. If the IoT hub in the primary region suffers an outage or the network connectivity from the device to the primary region is interrupted, devices use a secondary service endpoint. You can improve the solution availability by implementing a cross-region failover model instead of staying within a single region.

At a high level, to implement a regional failover model with IoT Hub, you need to take the following steps:

- **A secondary IoT hub and device routing logic:** If service in your primary region is disrupted, devices must start connecting to your secondary region. Given the state-aware nature of most services involved, it's common for solution administrators to trigger the inter-region failover process. The best way to communicate the new endpoint to devices, while maintaining control of the process, is to have them regularly check a *concierge* service for the current active endpoint. The concierge service can be a web application that is replicated and kept reachable using DNS-redirection techniques (for example, using [Azure Traffic Manager](#)).

NOTE

IoT hub service is not a supported endpoint type in Azure Traffic Manager. The recommendation is to integrate the proposed concierge service with Azure traffic manager by making it implement the endpoint health probe API.

- **Identity registry replication:** To be usable, the secondary IoT hub must contain all device identities that can connect to the solution. The solution should keep geo-replicated backups of device identities, and upload them to the secondary IoT hub before switching the active endpoint for the devices. The device identity export functionality of IoT Hub is useful in this context. For more information, see [IoT Hub developer guide - identity registry](#).
- **Merging logic:** When the primary region becomes available again, all the state and data that have been created in the secondary site must be migrated back to the primary region. This state and data mostly relate to device identities and application metadata, which must be merged with the primary IoT hub and any other application-specific stores in the primary region.

To simplify this step, you should use idempotent operations. Idempotent operations minimize the side-effects from the eventual consistent distribution of events, and from duplicates or out-of-order delivery of events. In addition, the application logic should be designed to tolerate potential inconsistencies or slightly out-of-date state. This situation can occur due to the additional time it takes for the system to heal based on recovery point objectives (RPO).

Choose the right HA/DR option

Here's a summary of the HA/DR options presented in this article that can be used as a frame of reference to choose the right option that works for your solution.

HA/DR OPTION	RTO	RPO	REQUIRES MANUAL INTERVENTION?	IMPLEMENTATION COMPLEXITY	ADDITIONAL COST IMPACT
Microsoft-initiated failover	2 - 26 hours	Refer RPO table above	No	None	None
Manual failover	10 min - 2 hours	Refer RPO table above	Yes	Very low. You only need to trigger this operation from the portal.	None
Cross region HA	< 1 min	Depends on the replication frequency of your custom HA solution	No	High	> 1x the cost of 1 IoT hub

Next steps

Follow these links to learn more about Azure IoT Hub:

- [Get started with IoT Hubs \(Quickstart\)](#)
- [What is Azure IoT Hub?](#)

Support additional protocols for IoT Hub

2/28/2019 • 2 minutes to read

Azure IoT Hub natively supports communication over the MQTT, AMQP, and HTTPS protocols. In some cases, devices or field gateways might not be able to use one of these standard protocols and require protocol adaptation. In such cases, you can use a custom gateway. A custom gateway enables protocol adaptation for IoT Hub endpoints by bridging the traffic to and from IoT Hub. You can use the [Azure IoT protocol gateway](#) as a custom gateway to enable protocol adaptation for IoT Hub.

Azure IoT protocol gateway

The Azure IoT protocol gateway is a framework for protocol adaptation that is designed for high-scale, bidirectional device communication with IoT Hub. The protocol gateway is a pass-through component that accepts device connections over a specific protocol. It bridges the traffic to IoT Hub over AMQP 1.0.

You can deploy the protocol gateway in Azure in a highly scalable way by using Azure Service Fabric, Azure Cloud Services worker roles, or Windows Virtual Machines. In addition, the protocol gateway can be deployed in on-premises environments, such as field gateways.

The Azure IoT protocol gateway includes an MQTT protocol adapter that enables you to customize the MQTT protocol behavior if necessary. Since IoT Hub provides built-in support for the MQTT v3.1.1 protocol, you should only consider using the MQTT protocol adapter if protocol customizations or specific requirements for additional functionality are required.

The MQTT adapter also demonstrates the programming model for building protocol adapters for other protocols. In addition, the Azure IoT protocol gateway programming model allows you to plug in custom components for specialized processing such as custom authentication, message transformations, compression/decompression, or encryption/decryption of traffic between the devices and IoT Hub.

For flexibility, the Azure IoT protocol gateway and MQTT implementation are provided in an open-source software project. You can use the open-source project to add support for various protocols and protocol versions, or customize the implementation for your scenario.

Next steps

To learn more about the Azure IoT protocol gateway and how to use and deploy it as part of your IoT solution, see:

- [Azure IoT protocol gateway repository on GitHub](#)
- [Azure IoT protocol gateway developer guide](#)

To learn more about planning your IoT Hub deployment, see:

- [Compare with Event Hubs](#)
- [Scaling, high availability, and disaster recovery](#)
- [IoT Hub developer guide](#)

Compare message routing and Event Grid for IoT Hub

2/22/2019 • 4 minutes to read

Azure IoT Hub provides the capability to stream data from your connected devices and integrate that data into your business applications. IoT Hub offers two methods for integrating IoT events into other Azure services or business applications. This article discusses the two features that provide this capability, so that you can choose which option is best for your scenario.

NOTE

Some of the features mentioned in this article, like cloud-to-device messaging, device twins, and device management, are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

IoT Hub message routing: This IoT Hub feature enables users to route device-to-cloud messages to service endpoints like Azure Storage containers, Event Hubs, Service Bus queues, and Service Bus topics. Routing also provides a querying capability to filter the data before routing it to the endpoints. In addition to device telemetry data, you can also send [non-telemetry events](#) that can be used to trigger actions.

IoT Hub integration with Event Grid: Azure Event Grid is a fully managed event routing service that uses a publish-subscribe model. IoT Hub and Event Grid work together to [integrate IoT Hub events into Azure and non-Azure services](#), in near-real time.

Differences

While both message routing and Event Grid enable alert configuration, there are some key differences between the two. Refer to the following table for details:

FEATURE	IOT HUB MESSAGE ROUTING	IOT HUB INTEGRATION WITH EVENT GRID
Device messages	Yes, message routing can be used for telemetry data.	No, Event Grid can only be used for non-telemetry IoT Hub events.
Device events	Yes, message routing can report twin changes and device lifecycle events.	Yes, Event Grid can report when devices are created, deleted, connected, and disconnected from IoT Hub
Ordering	Yes, ordering of events is maintained.	No, order of events is not guaranteed.
Maximum message size	256 KB, device-to-cloud	64 KB
Filtering	Rich filtering on message application properties, message system properties, message body, device twin tags, and device twin properties. For examples, see Message Routing Query Syntax .	Filtering based on suffix/prefix of device IDs, which works well for hierarchical services like storage.

FEATURE	IOT HUB MESSAGE ROUTING	IOT HUB INTEGRATION WITH EVENT GRID
Endpoints	<ul style="list-style-type: none"> • Event Hubs • Azure Blob Storage • Service Bus queue • Service Bus topics <p>Paid IoT Hub SKUs (S1, S2, and S3) are limited to 10 custom endpoints. 100 routes can be created per IoT Hub.</p>	<ul style="list-style-type: none"> • Azure Functions • Azure Automation • Event Hubs • Logic Apps • Storage Blob • Custom Topics • Third-party services through WebHooks <p>For the most up-to-date list of endpoints, see Event Grid event handlers.</p>
Cost	<p>There is no separate charge for message routing. Only ingress of telemetry into IoT Hub is charged. For example, if you have a message routed to three different endpoints, you are billed for only one message.</p>	<p>There is no charge from IoT Hub. Event Grid offers the first 100,000 operations per month for free, and then \$0.60 per million operations afterwards.</p>

Similarities

IoT Hub message routing and Event Grid have similarities too, some of which are detailed in the following table:

FEATURE	IOT HUB MESSAGE ROUTING	IOT HUB INTEGRATION WITH EVENT GRID
Reliability	High: Delivers each message to the endpoint at least once for each route. Expires all messages that are not delivered within one hour.	High: Delivers each message to the webhook at least once for each subscription. Expires all events that are not delivered within 24 hours.
Scalability	High: Optimized to support millions of simultaneously connected devices sending billions of messages.	High: Capable of routing 10,000,000 events per second per region.
Latency	Low: Near-real time.	Low: Near-real time.
Send to multiple endpoints	Yes, send a single message to multiple endpoints.	Yes, send a single message to multiple endpoints.
Security	IoT Hub provides per-device identity and revocable access control. For more information, see the IoT Hub access control .	Event Grid provides validation at three points: event subscriptions, event publishing, and webhook event delivery. For more information, see Event Grid security and authentication .

How to choose

IoT Hub message routing and the IoT Hub integration with Event Grid perform different actions to achieve similar results. They both take information from your IoT Hub solution and pass it on so that other services can react. So how do you decide which one to use? Consider the following questions to help guide your decision:

- **What kind of data are you sending to the endpoints?**

Use IoT Hub message routing when you have to send telemetry data to other services. Message routing also enables querying message application and system properties, message body, device twin tags, and device twin properties.

The IoT Hub integration with Event Grid works with events that occur in the IoT Hub service. These IoT Hub events include device created, deleted, connected, and disconnected.

- **What endpoints need to receive this information?**

IoT Hub message routing supports limited endpoints, but you can build connectors to reroute the data and events to additional endpoints. For a complete list of supported endpoints, see the table in the previous section.

The IoT Hub integration with Event Grid supports more endpoints. It also works with webhooks to extend routing outside of the Azure service ecosystem and into third-party business applications.

- **Does it matter if your data arrives in order?**

IoT Hub message routing maintains the order in which messages are sent, so that they arrive in the same way.

Event Grid does not guarantee that endpoints will receive events in the same order that they occurred. However, the event schema does include a timestamp that can be used to identify the order after the events arrive at the endpoint.

Next steps

- Learn more about [IoT Hub Message Routing](#) and the [IoT Hub endpoints](#).
- Learn more about [Azure Event Grid](#).
- To learn how to create Message Routes, see the [Process IoT Hub device-to-cloud messages using routes](#) tutorial.
- Try out the Event Grid integration by [Sending email notifications about Azure IoT Hub events using Logic Apps](#).

Best practices for device configuration within an IoT solution

9/17/2018 • 6 minutes to read

Automatic device management in Azure IoT Hub automates many repetitive and complex tasks of managing large device fleets over the entirety of their lifecycles. This article defines many of the best practices for the various roles involved in developing and operating an IoT solution.

- **IoT hardware manufacturer/integrator:** Manufacturers of IoT hardware, integrators assembling hardware from various manufacturers, or suppliers providing hardware for an IoT deployment manufactured or integrated by other suppliers. Involved in development and integration of firmware, embedded operating systems, and embedded software.
- **IoT solution developer:** The development of an IoT solution is typically done by a solution developer. This developer may be part of an in-house team or a system integrator specializing in this activity. The IoT solution developer can develop various components of the IoT solution from scratch, integrate various standard or open-source components, or customize an [IoT solution accelerator](#).
- **IoT solution operator:** After the IoT solution is deployed, it requires long-term operations, monitoring, upgrades, and maintenance. These tasks can be done by an in-house team that consists of information technology specialists, hardware operations and maintenance teams, and domain specialists who monitor the correct behavior of the overall IoT infrastructure.

Understand automatic device management for configuring IoT devices at scale

Automatic device management includes the many benefits of [device twins](#) and [module twins](#) to synchronize desired and reported states between the cloud and devices. [Automatic device configurations][lnk-auto-device-config] automatically update large sets of twins and summarize progress and compliance. The following high-level steps describe how automatic device management is developed and used:

- The **IoT hardware manufacturer/integrator** implements device management features within an embedded application using [device twins](#). These features could include firmware updates, software installation and update, and settings management.
- The **IoT solution developer** implements the management layer of device management operations using [device twins](#) and [automatic device configurations](#). The solution should include defining an operator interface to perform device management tasks.
- The **IoT solution operator** uses the IoT solution to perform device management tasks, particularly to group devices together, initiate configuration changes like firmware updates, monitor progress, and troubleshoot issues that arise.

IoT hardware manufacturer/integrator

The following are best practices for hardware manufacturers and integrators dealing with embedded software development:

- **Implement device twins:** Device twins enable synchronizing desired configuration from the cloud and for reporting current configuration and device properties. The best way to implement device twins within embedded applications is through the [Azure IoT SDKs](#). Device twins are best suited for configuration

because they:

- Support bi-directional communication.
 - Allow for both connected and disconnected device states.
 - Follow the principle of eventual consistency.
 - Are fully queriable in the cloud.
- **Structure the device twin for device management:** The device twin should be structured such that device management properties are logically grouped together into sections. Doing so will enable configuration changes to be isolated without impacting other sections of the twin. For example, create a section within desired properties for firmware, another section for software, and a third section for network settings.
 - **Report device attributes that are useful for device management:** Attributes like physical device make and model, firmware, operating system, serial number, and other identifiers are useful for reporting and as parameters for targeting configuration changes.
 - **Define the main states for reporting status and progress:** Top-level states should be enumerated so that they can be reported to the operator. For example, a firmware update would report status as Current, Downloading, Applying, In Progress, and Error. Define additional fields for more information on each state.

IoT solution developer

The following are best practices for IoT solution developers who are building systems based in Azure:

- **Implement device twins:** Device twins enable synchronizing desired configuration from the cloud and for reporting current configuration and device properties. The best way to implement device twins within cloud solutions applications is through the [Azure IoT SDKs](#). Device twins are best suited for configuration because they:
 - Support bi-directional communication.
 - Allow for both connected and disconnected device states.
 - Follow the principle of eventual consistency.
 - Are fully queriable in the cloud.
- **Organize devices using device twin tags:** The solution should allow the operator to define quality rings or other sets of devices based on various deployment strategies such as canary. Device organization can be implemented within your solution using device twin tags and [queries](#). Device organization is necessary to allow for configuration roll outs safely and accurately.
- **Implement automatic device configurations:** Automatic device configurations deploy and monitor configuration changes to large sets of IoT devices via device twins. Automatic device configurations target sets of device twins via the **target condition**, which is a query on device twin tags or reported properties. The **target content** is the set of desired properties that will be set within the targeted device twins. The target content should align with the device twin structure defined by the IoT hardware manufacturer/integrator.

The **metrics** are queries on device twin reported properties, and should also align with the device twin structure defined by the IoT hardware manufacturer/integrator. Automatic device configurations also have the benefit of IoT Hub performing device twin operations at a rate that will never exceed the [throttling limits](#) for device twin reads and updates.

- **Use the Device Provisioning Service:** Solution developers should use the Device Provisioning Service to assign device twin tags to new devices, such that they will be automatically configured by **automatic device configurations** that are targeted at twins with that tag.

IoT solution operator

The following are best practices for IoT solution operators who using an IoT solution built on Azure:

- **Organize devices for management:** The IoT solution should define or allow for the creation of quality rings or other sets of devices based on various deployment strategies such as canary. The sets of devices will be used to roll out configuration changes and to perform other at-scale device management operations.
- **Perform configuration changes using a phased roll out:** A phased roll out is an overall process whereby an operator deploys changes to a broadening set of IoT devices. The goal is to make changes gradually to reduce the risk of making wide scale breaking changes. The operator should use the solution's interface to create an [automatic device configuration](#) and the targeting condition should target an initial set of devices (such as a canary group). The operator should then validate the configuration change in the initial set of devices.

Once validation is complete, the operator will update the automatic device configuration to include a larger set of devices. The operator should also set the priority for the configuration to be higher than other configurations currently targeted to those devices. The roll out can be monitored using the metrics reported by the automatic device configuration.

- **Perform rollbacks in case of errors or misconfigurations:** An automatic device configuration that causes errors or misconfigurations can be rolled back by changing the **targeting condition** so that the devices no longer meet the targeting condition. Ensure that another automatic device configuration of lower priority is still targeted for those devices. Verify that the rollback succeeded by viewing the metrics: The rolled-back configuration should no longer show status for untargeted devices, and the second configuration's metrics should now include counts for the devices that are still targeted.

Next steps

- Learn about implementing device twins in [Understand and use device twins in IoT Hub](#).
- Walk through the steps to create, update, or delete an automatic device configuration in [Configure and monitor IoT devices at scale](#).
- Implement a firmware update pattern using device twins and automatic device configurations in [Tutorial: Implement a device firmware update process](#).

Azure IoT SDKs Platform Support

2/22/2019 • 2 minutes to read

The [Azure IoT SDKs](#) are a set of libraries to interact with IoT Hub and the Device Provisioning Service with broad language and platform support. The SDKs run on most common platforms, and developers can port the C SDK to specific platform by following the [Porting Guidance](#).

Microsoft supports a variety of operating systems/platforms/frameworks and can be extended using the Azure IoT C SDK. Some are supported officially by the team, grouped into tiers that represent the level of support users can expect. *Fully supported platforms* means that Microsoft:

- Continuously builds and runs end-to-end tests against master and the LTS supported version(s). To provide test coverage across different versions, we generally test against the latest LTS version and the most popular version. Other versions of the same platform may be supported via platform version compatibility.
- Provides installation guidance or packages if applicable.
- Fully supports the platforms on GitHub.

In addition, a list of partners has ported our C SDK on to more platforms and they are maintaining the platform abstraction layer (PAL). [Azure Certified for IoT Device Catalog](#) also features a list of OS platforms the various SDKs have been tested against. The SDKs also regularly build on these platforms, with limited testing and support:

- MBED2
- Arduino
- Windows CE 2013 (deprecate in October 2018)
- .NET Standard 1.3 with .NET Core 2.1 and .NET Framework 4.7
- Xamarin iOS, Android, UWP

Supported platforms

There are several platforms supported.

C SDK

OS	ARCH	COMPILER	TLS LIBRARY
Ubuntu 16.04 LTS	x64	gcc-5.4.0	openssl - 1.0.2g
Ubuntu 18.04 LTS	x64	gcc-7.3	WolfSSL – 1.13
Ubuntu 18.04 LTS	x64	Clang 6.0.X	Openssl – 1.1.0g
OSX 10.13.4	x64	XCode 9.4.1	Native OSX
Windows Server 2016	x64	Visual Studio 14.0.X	SChannel
Windows Server 2016	x86	Visual Studio 14.0.X	SChannel
Debian 9 Stretch	x64	gcc-7.3	Openssl – 1.1.0f

Python SDK

OS	ARCH	COMPILER	TLS LIBRARY
Windows Server 2016	x86	Python 2.7	openssl
Windows Server 2016	x64	Python 2.7	openssl
Windows Server 2016	x86	Python 3.5	openssl
Windows Server 2016	x64	Python 3.5	openssl
Ubuntu 18.04 LTS	x86	Python 2.7	openssl
Ubuntu 18.04 LTS	x86	Python 3.4	openssl
MacOS High Sierra	x64	Python 2.7	openssl

.NET SDK

OS	ARCH	FRAMEWORK	STANDARD
Ubuntu 16.04 LTS	X64	.NET Core 2.1	.NET standard 2.0
Windows Server 2016	X64	.NET Core 2.1	.NET standard 2.0
Windows Server 2016	X64	.NET Framework 4.7	.NET standard 2.0
Windows Server 2016	X64	.NET Framework 4.5.1	N/A

Node.js SDK

OS	ARCH	NODE VERSION
Ubuntu 16.04 LTS (using node 6 docker image)	X64	Node 6
Windows Server 2016	X64	Node 6

Java SDK

OS	ARCH	JAVA VERSION
Ubuntu 16.04 LTS	X64	Java 8
Windows Server 2016	X64	Java 8
Android API 28	X64	Java 8
Android Things	X64	Java 8

Partner supported platforms

Customers can extend our platform support by porting the Azure IoT C SDK, specifically, creating the platform abstraction layer (PAL) of the SDK. Microsoft works with partners to provide extended support. A list of partners

has ported the C SDK on to more platforms and maintaining the PAL.

PARTNER	DEVICES	LINK	SUPPORT
Espressif	ESP32 ESP8266	Esp-azure	GitHub
Qualcomm	Qualcomm MDM9206 LTE IoT Modem	Qualcomm LTE for IoT SDK	Forum
ST Microelectronics	STM32L4 Series STM32F4 Series STM32F7 Series STM32L4 Discovery Kit for IoT node	X-CUBE-CLOUD X-CUBE-AZURE P-NUCLEO-AZURE FP-CLD-AZURE	Support
Texas Instruments	CC3220SF Launchpad CC3220S Launchpad MSP432E4 Launchpad	Azure IoT Plugin for SimpleLink	TI E2E Forum TI E2E Forum for CC3220 TI E2E Forum for MSP432E4

Next steps

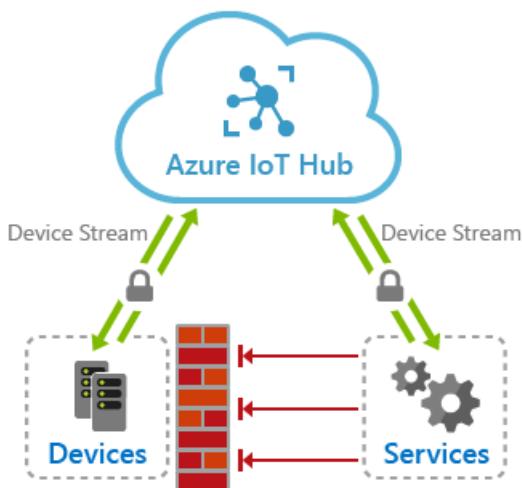
- [Device and service SDKs](#)
- [Porting Guidance](#)

IoT Hub Device Streams (preview)

3/5/2019 • 9 minutes to read

Overview

Azure IoT Hub *device streams* facilitate the creation of secure bi-directional TCP tunnels for a variety of cloud-to-device communication scenarios. A device stream is mediated by an IoT Hub *streaming endpoint* which acts as a proxy between your device and service endpoints. This setup, depicted in the diagram below, is especially useful when devices are behind a network firewall or reside inside of a private network. As such, IoT Hub device streams help address customers' need to reach IoT devices in a firewall-friendly manner and without the need to broadly opening up incoming or outgoing network firewall ports.



Using IoT Hub device streams, devices remain secure and will only need to open up outbound TCP connections to IoT hub's streaming endpoint over port 443. Once a stream is established, the service-side and device-side applications will each have programmatic access to a WebSocket client object to send and receive raw bytes to one another. The reliability and ordering guarantees provided by this tunnel is on par with TCP.

Benefits

IoT Hub device streams provide the following benefits:

- **Firewall-friendly secure connectivity:** IoT devices can be reached from service endpoints without opening of inbound firewall port at the device or network perimeters (only outbound connectivity to IoT Hub is needed over port 443).
- **Authentication:** Both device and service sides of the tunnel need to authenticate with IoT Hub using their corresponding credentials.
- **Encryption:** By default, IoT Hub device streams use TLS-enabled connections. This ensures that the traffic is always encrypted regardless of whether the application uses encryption or not.
- **Simplicity of connectivity:** In many cases, the use of device streams eliminates the need for complex setup of Virtual Private Networks to enable connectivity to IoT devices.
- **Compatibility with TCP/IP stack:** IoT Hub device streams can accommodate TCP/IP application traffic. This means that a wide range of proprietary as well as standards-based protocols can leverage this feature.

- **Ease of use in private network setups:** Service can communicate with a device by referencing its device ID, rather than device's IP address. This is useful in situations where a device is located inside a private network and has a private IP address, or its IP address is assigned dynamically and is unknown to the service side.

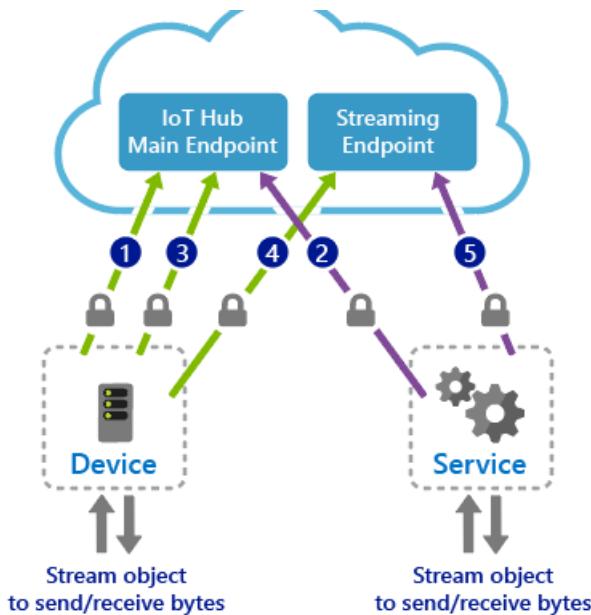
Device Stream Workflows

A device stream is initiated when the service requests to connect to a device by providing its device ID. This workflow particularly fits into a client/server communication model, including SSH and RDP, where a user intends to remotely connect to the SSH or RDP server running on the device using an SSH or RDP client program.

The device stream creation process involves a negotiation between the device, service, IoT hub's main and streaming endpoints. While IoT hub's main endpoint orchestrates the creation of a device stream, the streaming endpoint handles the traffic that flows between the service and device.

Device stream creation flow

Programmatic creation of a device stream using the SDK involves the following steps, which are also depicted in the figure below:



1. The device application registers a callback in advance to be notified of when a new device stream is initiated to the device. This step typically takes place when the device boots up and connects to IoT Hub.
2. The service-side program initiates a device stream when needed by providing the device ID (*not* the IP address).
3. IoT hub notifies the device-side program by invoking the callback registered in step 1. The device may accept or reject the stream initiation request. This logic can be specific to your application scenario. If the stream request is rejected by the device, IoT Hub informs the service accordingly; otherwise, the steps below follow.
4. The device creates a secure outbound TCP connection to the streaming endpoint over port 443 and upgrades the connection to a WebSocket. The URL of the streaming endpoint as well as the credentials to use to authenticate are both provided to the device by IoT Hub as part of the request sent in step 3.
5. The service is notified of the result of device accepting the stream and proceeds to create its own WebSocket client to the streaming endpoint. Similarly, it receives the streaming endpoint URL and authentication information from IoT Hub.

In the handshake process above:

- The handshake process must complete within 60 seconds (step 2 through 5), otherwise the handshake would fail with a timeout and the service will be notified accordingly.
- After the stream creation flow above completes, the streaming endpoint will act as a proxy and will transfer traffic between the service and the device over their respective WebSockets.
- Device and service both need outbound connectivity to IoT Hub's main endpoint as well as the streaming endpoint over port 443. The URL of these endpoints is available on *Overview* tab on the IoT Hub's portal.
- The reliability and ordering guarantees of an established stream is on par with TCP.
- All connections to IoT Hub and streaming endpoint use TLS and are encrypted.

Termination flow

An established stream terminates when either of the TCP connections to the gateway are disconnected (by the service or device). This can take place voluntarily by closing the WebSocket on either the device or service programs, or involuntarily in case of a network connectivity timeout or process failure. Upon termination of either device or service's connection to the streaming endpoint, the other TCP connection will also be (forcefully) terminated and the service and device are responsible to re-create the stream, if needed.

Connectivity Requirements

Both the device and the service sides of a device stream must be capable of establishing TLS-enabled connections to IoT Hub and its streaming endpoint. This requires outbound connectivity over port 443 to these endpoints. The hostname associated with these endpoints can be found on the *Overview* tab of IoT Hub, as shown in the figure below:

Home > IoTHubDeviceStreams
IoTHubDeviceStreams
IoT Hub
Search (Ctrl+F)
Move Delete Refresh
Overview
Activity log
Access control (IAM)
Tags
Events
Settings
Resource group (change) : IoTHubDeviceStreamsRG
Status : Active
Location : centraluseap
Subscription (change) :
Subscription ID :
Tags (change) : Click here to add tags
Hostname : IoTHubDeviceStreams.azure-devices.net
Pricing and scale tier : F1 - Free
Number of IoT Hub units : 1
Device streams (preview) : https://eastus2euap.eastus2euap-001.streams.azure-devices.net:9080
Device streams documentation

Alternatively, the endpoints information can be retrieved using Azure CLI under the hub's properties section, specifically, `property.hostname` and `property.deviceStreams` keys.

```
az iot hub devicestream show --name <YourIoTHubName>
```

The output is a JSON object of all endpoints that your hub's device and service may need to connect to in order to establish a device stream.

```
{  
  "streamingEndpoints": [  
    "https://<YourIoTHubName>.<region-stamp>.streams.azure-devices.net"  
  ]  
}
```

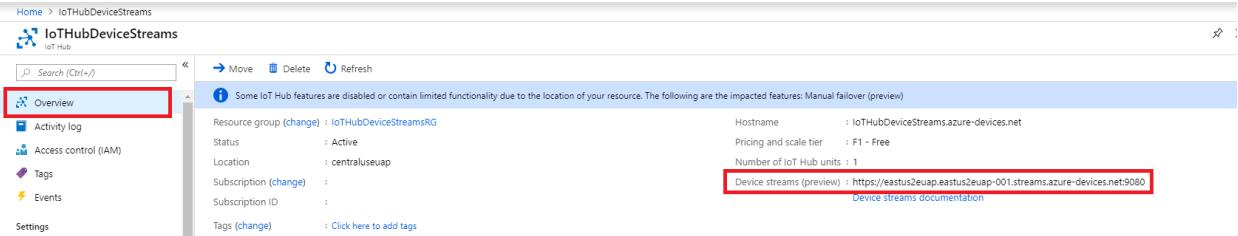
NOTE

Ensure you have installed Azure CLI version 2.0.57 or newer. You can download the latest version [here](#).

Whitelist Device Streaming Endpoints

As mentioned [earlier](#), your device creates an outbound connection to IoT Hub streaming endpoint during device streams initiation process. Your firewalls on the device or its network must allow outbound connectivity to the streaming gateway over port 443 (note that communication takes place over a WebSocket connection that is encrypted using TLS).

The hostname of device streaming endpoint can be found on the Azure IoT Hub portal under the Overview tab.



A screenshot of the Azure IoT Hub Device Streams Overview page. The page shows basic resource details: Resource group (IoTHubDeviceStreamsRG), Status (Active), Location (centraluseuap), Subscription (change), Subscription ID, Tags (change), and Hostname (IoTHubDeviceStreams.azure-devices.net). The 'Device streams (preview)' field contains the URL <https://eastus2euap.eastus2euap-001.streams.azure-devices.net:9080>. A red box highlights this URL.

Alternatively, you can find this information using Azure CLI:

```
az iot hub devicestream show --name <YourIoTHubName>
```

NOTE

Ensure you have installed Azure CLI version 2.0.57 or newer. You can download the latest version [here](#).

Troubleshoot via Device Streams Activity Logs

You can set up Azure Monitor logs to collect the activity log of device streams in your IoT Hub. This can be very helpful in troubleshooting scenarios.

Follow the steps below to configure Azure Monitor logs for your IoT Hub's device stream activities:

1. Navigate to the *Diagnostic settings* tab in your IoT Hub, and click on *Turn on diagnostics* link.

The screenshot shows the 'IoTHubDeviceStreams - Diagnostic settings' page in the Azure portal. The left sidebar contains a navigation menu with various options like Pricing and scale, Operations monitoring, IP Filter, Certificates, Built-in endpoints, Properties, Locks, Automation script, Query explorer, IoT devices, IoT Edge, IoT device configuration, File upload, Message routing, Manual failover (preview), Alerts, Metrics, and Diagnostic settings. The 'Diagnostic settings' option is highlighted with a red box. The main content area shows a 'Subscription' section with a red box around the text 'Turn on diagnostics to collect the following data.' A 'Resource group' dropdown is set to 'IoTHubDeviceStreamsRG'. The breadcrumb navigation at the top indicates the path: OneDeploy Demo and Test Subscription > IoTHubDeviceStreamsRG > IoTHubDeviceStreams.

2. Provide a name for your diagnostics settings, and choose *Send to Log Analytics* option. You will be guided to choose an existing Log Analytics workspace resource or create a new one. Additionally, check the *DeviceStreams* from the list.

Diagnostics settings

 Save Discard Delete

* Name

MyDeviceStreamDiagnostics  Archive to a storage account Stream to an event hub Send to Log Analytics

Log Analytics

iot-hub-2-streaming-analytics >

LOG

 Connections DeviceTelemetry C2DCommands DeviceIdentityOperations FileUploadOperations Routes D2CTwinOperations C2DTwinOperations TwinQueries JobsOperations DirectMethods DistributedTracing Configurations DeviceStreams

3. You can now access your device streams logs under the *Logs* tab in your IoT Hub's portal. Device stream activity logs will appear in the `AzureDiagnostics` table and have `Category=DeviceStreams`.

As shown below the identity of the target device and the result of the operation is also available in the logs.

The screenshot shows the Azure IoT Hub Device Streams - Logs interface. On the left, there's a navigation sidebar with various tabs like Home, Pricing and scale, Operations monitoring, IP Filter, Certificates, Built-in endpoints, Properties, Locks, Automation script, Explorers, Query explorer, IoT devices, Automatic Device Management, IoT Edge, IoT device configuration, Messaging, File upload, Message routing, Resiliency, Manual failover (preview), Monitoring, Alerts, Metrics, Diagnostic settings, and Logs. The Logs tab is selected. In the main area, there's a search bar, a schema dropdown, and a query editor with the text "AzureDiagnostics | take 10". Below the query editor is a table with columns: TenantId, SourceSystem, MG, ManagementGroupName, TimeGenerated [UTC], Computer, ResultSignature, and CallerIpAddress. The table contains 10 rows of data. Some specific fields in the first row are highlighted with red boxes: TenantId ("d21ba1e6-c6c7-48fc-912a-684e7e3bbde7"), SourceSystem ("Azure"), TimeGenerated [UTC] ("2019-01-11T22:33:06.216Z"), ResultSignature ("200"), and CallerIpAddress ("0.0.0.0"). The table also includes a detailed view of the first row with expanded properties like identity_s, DurationMs, ResultType, ResourceId, OperationName, Category, Level, properties_s, SubscriptionId, ResourceGroup, and ResourceProvider.

Regional Availability

During public preview, IoT Hub device streams are available in the Central US and Central US EUAP regions. Please make sure you create your hub in one of these regions.

SDK Availability

Two sides of each stream (on the device and service side) use the IoT Hub SDK to establish the tunnel. During public preview, customers can choose from the following SDK languages:

- The C and C# SDK's support device streams on the device side.
- The NodeJS and C# SDK support device streams on the service side.

IoT Hub Device Stream Samples

We have published two [quickstart samples](#) to demonstrate the use of device streams by applications.

- The *echo* sample demonstrates programmatic use of device streams (by calling the SDK API's directly).
- The *local proxy* sample demonstrates the tunneling of off-the-shelf client/server application traffic (such as SSH, RDP, or web) through device streams.

These samples are covered in greater detail below.

Echo Sample

The echo sample demonstrates programmatic use of device streams to send and receive bytes between service and device applications. Use the links below to access the quickstart guides. Note that you can use service and device programs in different languages, e.g., C device program can work with C# service program.

SDK	SERVICE PROGRAM	DEVICE PROGRAM
C#	Link	Link
Node.js	Link	-

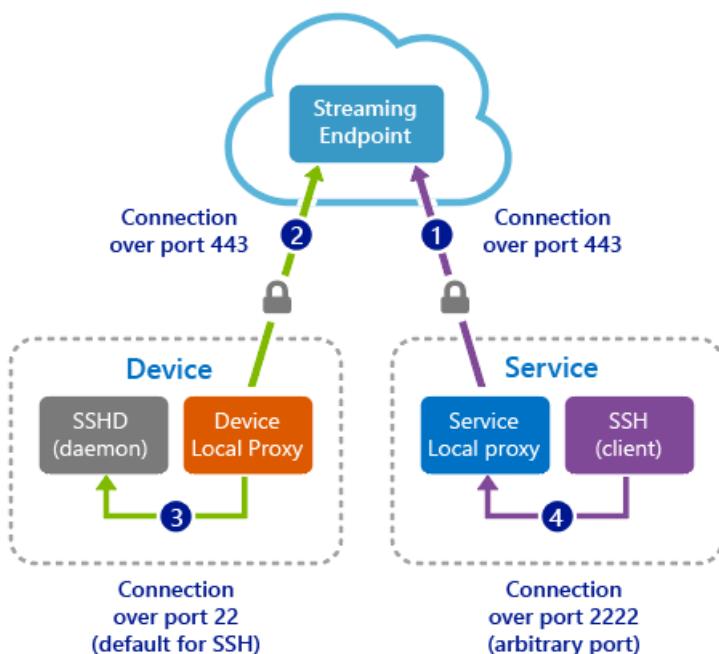
SDK	SERVICE PROGRAM	DEVICE PROGRAM
C	-	Link

Local Proxy Sample (for SSH or RDP)

The local proxy sample demonstrates a way to enable tunneling of an existing application's traffic that involves communication between a client and a server program. This set up works for client/server protocols like SSH and RDP, where the service-side acts as a client (running SSH or RDP client programs), and the device-side acts as the server (running SSH daemon or RDP server programs).

This section describes the use of device streams to enable the user to SSH to a device over device streams (the case for RDP or other client/server application are similar by using the protocol's corresponding port).

The setup leverages two *local proxy* programs shown in the figure below, namely *device-local proxy* and *service-local proxy*. The local proxy programs are responsible for performing the [device stream initiation handshake](#) with IoT Hub, and interacting with SSH client and SSH daemon using regular client/server sockets.



1. The user runs service-local proxy to initiate a device stream to the device.
2. The device-local proxy accepts the stream initiation request and the tunnel is established to IoT Hub's streaming endpoint (as discussed above).
3. The device-local proxy connects to the SSH daemon endpoint listening on port 22 on the device.
4. The service-local proxy listens on a designated port awaiting new SSH connections from the user (port 2222 used in the sample, but this can be configured to any other available port). The user points the SSH client to the service-local proxy port on localhost.

Notes

- The above steps complete an end-to-end tunnel between the SSH client (on the right) to the SSH daemon (on the left). Part of this end-to-end connectivity involves sending traffic over a device stream to IoT Hub.
- The arrows in the figure indicate the direction in which connections are established between endpoints. Specifically, note that there is no inbound connections going to the device (this is often blocked by a firewall).
- The choice of using port 2222 on the service-local proxy is an arbitrary choice. The proxy can be

configured to use any other available port.

- The choice of port 22 is protocol-dependent and specific to SSH in this case. For the case of RDP, the port 3389 must be used. This can be configured in the provided sample programs.

Use the links below for instructions on how to run the local proxy programs in your language of choice. Similar to the [echo sample](#), you can run device- and service-local proxy programs in different languages as they are fully interoperable.

SDK	SERVICE-LOCAL PROXY	DEVICE-LOCAL PROXY
C#	Link	Link
NodeJS	Link	-
C	-	Link

Next steps

Use the links below to learn more about device streams:

[Device streams on IoT show \(Channel 9\)](#) Try a device stream quickstart guide

Azure IoT Hub developer guide

3/5/2019 • 3 minutes to read

Azure IoT Hub is a fully managed service that helps enable reliable and secure bi-directional communications between millions of devices and a solution back end.

NOTE

Some of the features mentioned in this article, like cloud-to-device messaging, device twins, and device management, are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

Azure IoT Hub provides you with:

- Secure communications by using per-device security credentials and access control.
- Multiple device-to-cloud and cloud-to-device hyper-scale communication options.
- Queryable storage of per-device state information and meta-data.
- Easy device connectivity with device libraries for the most popular languages and platforms.

This IoT Hub developer guide includes the following articles:

- [Device-to-cloud communication guidance](#) helps you choose between device-to-cloud messages, device twin's reported properties, and file upload.
- [Cloud-to-device communication guidance](#) helps you choose between direct methods, device twin's desired properties, and cloud-to-device messages.
- [Device-to-cloud and cloud-to-device messaging with IoT Hub](#) describes the messaging features (device-to-cloud and cloud-to-device) that IoT Hub exposes.
 - [Send device-to-cloud messages to IoT Hub](#).
 - [Read device-to-cloud messages from the built-in endpoint](#).
 - [Use custom endpoints and routing rules for device-to-cloud messages](#).
 - [Send cloud-to-device messages from IoT Hub](#).
 - [Create and read IoT Hub messages](#).
- [Upload files from a device](#) describes how you can upload files from a device. The article also includes information about topics such as the notifications the upload process can send.
- [Manage device identities in IoT Hub](#) describes what information each IoT hub's identity registry stores. The article also describes how you can access and modify it.
- [Control access to IoT Hub](#) describes the security model used to grant access to IoT Hub functionality for both devices and cloud components. The article includes information about using tokens and X.509 certificates, and details of the permissions you can grant.
- [Use device twins to synchronize state and configurations](#) describes the *device twin* concept. The article also describes the functionality device twins expose, such as synchronizing a device with its device twin. The article includes information about the data stored in a device twin.

- [Invoke a direct method on a device](#) describes the lifecycle of a direct method. The article describes how to invoke methods on a device from your back-end app and handle the direct method on your device.
- [Schedule jobs on multiple devices](#) describes how you can schedule jobs on multiple devices. The article describes how to submit jobs that perform tasks as executing a direct method, updating a device using a device twin. It also describes how to query the status of a job.
- [Reference - choose a communication protocol](#) describes the communication protocols that IoT Hub supports for device communication and lists the ports that should be open.
- [Reference - IoT Hub endpoints](#) describes the various endpoints that each IoT hub exposes for runtime and management operations. The article also describes how you can create additional endpoints in your IoT hub, and how to use a field gateway to enable connectivity to your IoT Hub endpoints in non-standard scenarios.
- [Reference - IoT Hub query language for device twins, jobs, and message routing](#) describes that IoT Hub query language that enables you to retrieve information from your hub about your device twins and jobs.
- [Reference - quotas and throttling](#) summarizes the quotas set in the IoT Hub service and the throttling that occurs when you exceed a quota.
- [Reference - pricing](#) provides general information on different SKUs and pricing for IoT Hub and details on how the various IoT Hub functionalities are metered as messages by IoT Hub.
- [Reference - Device and service SDKs](#) lists the Azure IoT SDKs for developing device and service apps that interact with your IoT hub. The article includes links to online API documentation.
- [Reference - IoT Hub MQTT support](#) provides detailed information about how IoT Hub supports the MQTT protocol. The article describes the support for the MQTT protocol built-in to the Azure IoT SDKs and provides information about using the MQTT protocol directly.
- [Glossary](#) a list of common IoT Hub-related terms.

Device-to-cloud communications guidance

2/28/2019 • 2 minutes to read

When sending information from the device app to the solution back end, IoT Hub exposes three options:

- [Device-to-cloud messages](#) for time series telemetry and alerts.
- [Device twin's reported properties](#) for reporting device state information such as available capabilities, conditions, or the state of long-running workflows. For example, configuration and software updates.
- [File uploads](#) for media files and large telemetry batches uploaded by intermittently connected devices or compressed to save bandwidth.

NOTE

Some of the features mentioned in this article, like cloud-to-device messaging, device twins, and device management, are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

Here is a detailed comparison of the various device-to-cloud communication options.

	DEVICE-TO-CLOUD MESSAGES	DEVICE TWIN'S REPORTED PROPERTIES	FILE UPLOADS
Scenario	Telemetry time series and alerts. For example, 256-KB sensor data batches sent every 5 minutes.	Available capabilities and conditions. For example, the current device connectivity mode such as cellular or WiFi. Synchronizing long-running workflows, such as configuration and software updates.	Media files. Large (typically compressed) telemetry batches.
Storage and retrieval	Temporarily stored by IoT Hub, up to 7 days. Only sequential reading.	Stored by IoT Hub in the device twin. Retrievable using the IoT Hub query language .	Stored in user-provided Azure Storage account.
Size	Up to 256-KB messages.	Maximum reported properties size is 8 KB.	Maximum file size supported by Azure Blob Storage.
Frequency	High. For more information, see IoT Hub limits .	Medium. For more information, see IoT Hub limits .	Low. For more information, see IoT Hub limits .
Protocol	Available on all protocols.	Available using MQTT or AMQP.	Available when using any protocol, but requires HTTPS on the device.

An application may need to send information both as a telemetry time series or alert and make it available in the device twin. In this scenario, you can choose one of the following options:

- The device app sends a device-to-cloud message and reports a property change.
- The solution back end can store the information in the device twin's tags when it receives the message.

Since device-to-cloud messages enable a much higher throughput than device twin updates, it is sometimes desirable to avoid updating the device twin for every device-to-cloud message.

Cloud-to-device communications guidance

2/28/2019 • 2 minutes to read

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 send interval to 30 minutes.
- [Cloud-to-device messages](#) for one-way notifications to the device app.

NOTE

The features described in this article are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

Here is a detailed comparison of the various cloud-to-device communication options.

	DIRECT METHODS	TWIN'S DESIRED PROPERTIES	CLOUD-TO-DEVICE MESSAGES
Scenario	Commands that require immediate confirmation, such as turning on a fan.	Long-running commands intended to put the device into a certain desired state. For example, set the telemetry send interval to 30 minutes.	One-way notifications to the device app.
Data flow	Two-way. The device app can respond to the method right away. The solution back end receives the outcome contextually to the request.	One-way. The device app receives a notification with the property change.	One-way. The device app receives the message
Durability	Disconnected devices are not contacted. The solution back end is notified that the device is not connected.	Property values are preserved in the device twin. Device will read it at next reconnection. Property values are retrievable with the IoT Hub query language .	Messages can be retained by IoT Hub for up to 48 hours.
Targets	Single device using deviceId , or multiple devices using jobs .	Single device using deviceId , or multiple devices using jobs .	Single device by deviceId .
Size	Maximum direct method payload size is 128 KB.	Maximum desired properties size is 8 KB.	Up to 64 KB messages.
Frequency	High. For more information, see IoT Hub limits .	Medium. For more information, see IoT Hub limits .	Low. For more information, see IoT Hub limits .

	DIRECT METHODS	TWIN'S DESIRED PROPERTIES	CLOUD-TO-DEVICE MESSAGES
Protocol	Available using MQTT or AMQP.	Available using MQTT or AMQP.	Available on all protocols. Device must poll when using HTTPS.

Learn how to use direct methods, desired properties, and cloud-to-device messages in the following tutorials:

- [Use direct methods](#)
- [Use desired properties to configure devices](#)
- [Send cloud-to-device messages](#)

Send device-to-cloud and cloud-to-device messages with IoT Hub

2/28/2019 • 2 minutes to read

IoT Hub allows for bi-directional communication with your devices. Use IoT Hub messaging to communicate with your devices by sending messages from your devices to your solutions back end and sending commands from your IoT solutions back end to your devices. Learn more about the [IoT Hub message format](#).

Sending device-to-cloud messages to IoT Hub

IoT Hub has a built-in service endpoint that can be used by back-end services to read telemetry messages from your devices. This endpoint is compatible with [Event Hubs](#) and you can use standard IoT Hub SDKs to [read from this built-in endpoint](#).

IoT Hub also supports [custom endpoints](#) that can be defined by users to send device telemetry data and events to Azure services using [message routing](#).

Sending cloud-to-device messages from IoT Hub

You can send [cloud-to-device](#) messages from the solution back end to your devices.

NOTE

Some of the features mentioned in this article, like cloud-to-device messaging, device twins, and device management, are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

Core properties of IoT Hub messaging functionality are the reliability and durability of messages. These properties enable resilience to intermittent connectivity on the device side, and to load spikes in event processing on the cloud side. IoT Hub implements *at least once* delivery guarantees for both device-to-cloud and cloud-to-device messaging.

Choosing the right type of IoT Hub messaging

Use device-to-cloud messages for sending time series telemetry and alerts from your device app, and cloud-to-device messages for one-way notifications to your device app.

- Refer to [Device-to-cloud communication guidance](#) to choose between device-to-cloud messages, reported properties, or file upload.
- Refer to [Cloud-to-device communication guidance](#) to choose between cloud-to-device messages, desired properties, or direct methods.

Next steps

- Learn about IoT Hub [message routing](#).
- Learn about IoT Hub [cloud-to-device messaging](#).

Use IoT Hub message routing to send device-to-cloud messages to different endpoints

3/7/2019 • 7 minutes to read

NOTE

Some of the features mentioned in this article, like cloud-to-device messaging, device twins, and device management, are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

Message routing enables you to send messages from your devices to cloud services in an automated, scalable, and reliable manner. Message routing can be used for:

- **Sending device telemetry messages as well as events** namely, device lifecycle events, and device twin change events to the built-in-endpoint and custom endpoints. Learn about [routing endpoints](#).
- **Filtering data before routing it to various endpoints** by applying rich queries. Message routing allows you to query on the message properties and message body as well as device twin tags and device twin properties. Learn more about using [queries in message routing](#).

IoT Hub needs write access to these service endpoints for message routing to work. If you configure your endpoints through the Azure portal, the necessary permissions are added for you. Make sure you configure your services to support the expected throughput. When you first configure your IoT solution, you may need to monitor your additional endpoints and make any necessary adjustments for the actual load.

The IoT Hub defines a [common format](#) for all device-to-cloud messaging for interoperability across protocols. If a message matches multiple routes that point to the same endpoint, IoT Hub delivers message to that endpoint only once. Therefore, you don't need to configure deduplication on your Service Bus queue or topic. In partitioned queues, partition affinity guarantees message ordering. Use this tutorial to learn how to [configure message routing](#).

Routing endpoints

An IoT hub has a default built-in-endpoint (**messages/events**) that is compatible with Event Hubs. You can create [custom endpoints](#) to route messages to by linking other services in your subscription to the IoT Hub. IoT Hub currently supports the following services as custom endpoints:

Built-in endpoint

You can use standard [Event Hubs integration and SDKs](#) to receive device-to-cloud messages from the built-in endpoint (**messages/events**). Once a Route is created, data stops flowing to the built-in-endpoint unless a Route is created to that endpoint.

Azure Blob Storage

IoT Hub supports writing data to Azure Blob Storage in the [Apache Avro](#) as well as JSON format. The capability to encode JSON format is in preview in all regions IoT Hub is available in, except East US, West US and West Europe. The default is AVRO. The encoding format can be only set when the blob storage endpoint is configured. The format cannot be edited for an existing endpoint. When using JSON encoding, you must set the `contentType` to JSON and `contentEncoding` to UTF-8 in the message [system properties](#). You can select the encoding format using the IoT Hub Create or Update REST API, specifically the

[RoutingStorageContainerProperties](#), the Azure Portal, [Azure CLI](#) or the [Azure Powershell](#). The following diagram shows how to select the encoding format in the Azure Portal.

Add a storage endpoint

Route your telemetry and device messages to Azure Storage as blobs.

* Endpoint name [?](#)

Azure Storage account and container

Create a new container, or choose an existing one that shares a subscription with this IoT hub.

Azure Storage container
[Pick a container](#)

Batch frequency [?](#)

Chunk size window [?](#)

Encoding [?](#)
 AVRO JSON

* Blob file name format [?](#)

The format must contain {iothub}, {partition}, {YYYY}, {MM}, {DD}, {HH} and {mm} in any order.

If multiple files are created within the same minute, the filename format would be ashitaHub/0/2019/01/29/09/55-01.

IoT Hub batches messages and writes data to a blob whenever the batch reaches a certain size or a certain amount of time has elapsed. IoT Hub defaults to the following file naming convention:

```
{iothub}/{partition}/{YYYY}/{MM}/{DD}/{HH}/{mm}
```

You may use any file naming convention, however you must use all listed tokens. IoT Hub will write to an empty blob if there is no data to write.

When routing to blob storage, we recommend enlisting the blobs and then iterating over them, to ensure all containers are read without making any assumptions of partition. The partition range could potentially change during a [Microsoft-initiated failover](#) or IoT Hub [manual failover](#). You can use the [List Blobs API](#) to enumerate the list of blobs. Please see the following sample as guidance.

```

public void ListBlobsInContainer(string containerName, string iothub)
{
    var storageAccount = CloudStorageAccount.Parse(this.blobConnectionString);
    var cloudBlobContainer =
storageAccount.CreateCloudBlobClient().GetContainerReference(containerName);
    if (cloudBlobContainer.Exists())
    {
        var results = cloudBlobContainer.ListBlobs(prefix: $"{iothub}/");
        foreach (IListBlobItem item in results)
        {
            Console.WriteLine(item.Uri);
        }
    }
}

```

Service Bus Queues and Service Bus Topics

Service Bus queues and topics used as IoT Hub endpoints must not have **Sessions** or **Duplicate Detection** enabled. If either of those options are enabled, the endpoint appears as **Unreachable** in the Azure portal.

Event Hubs

Apart from the built-in-Event Hubs compatible endpoint, you can also route data to custom endpoints of type Event Hubs.

Reading data that has been routed

You can configure a route by following this [tutorial](#).

Use the following tutorials to learn how to read message from an endpoint.

- Reading from [Built-in-endpoint](#)
- Reading from [Blob storage](#)
- Reading from [Event Hubs](#)
- Reading from [Service Bus Queues](#)
- Read from [Service Bus Topics](#)

Fallback route

The fallback route sends all the messages that don't satisfy query conditions on any of the existing routes to the built-in-Event Hubs (**messages/events**), that is compatible with [Event Hubs](#). If message routing is turned on, you can enable the fallback route capability. Once a route is created, data stops flowing to the built-in-endpoint, unless a route is created to that endpoint. If there are no routes to the built-in-endpoint and a fallback route is enabled, only messages that don't match any query conditions on routes will be sent to the built-in-endpoint. Also, if all existing routes are deleted, fallback route must be enabled to receive all data at the built-in-endpoint.

You can enable/disable the fallback route in the Azure Portal-> Message Routing blade. You can also use Azure Resource Manager for [FallbackRouteProperties](#) to use a custom endpoint for fallback route.

Non-telemetry events

In addition to device telemetry, message routing also enables sending device twin change events and device lifecycle events. For example, if a route is created with data source set to **device twin change events**, IoT Hub sends messages to the endpoint that contain the change in the device twin. Similarly, if a route is created with data source set to **device lifecycle events**, IoT Hub will send a message indicating whether the device was deleted or created.

IoT Hub also integrates with Azure Event Grid to publish device events to support real time integrations and automation of workflows based on these events. See key [differences between message routing and Event Grid](#) to learn which works best for your scenario.

Testing routes

When you create a new route or edit an existing route, you should test the route query with a sample message. You can test individual routes or test all routes at once and no messages are routed to the endpoints during the test. Azure Portal, Azure Resource Manager, Azure PowerShell, and Azure CLI can be used for testing. Outcomes help identify whether the sample message matched the query, message did not match the query, or test couldn't run because the sample message or query syntax are incorrect. To learn more, see [Test Route](#) and [Test all routes](#).

Latency

When you route device-to-cloud telemetry messages using built-in endpoints, there is a slight increase in the end-to-end latency after the creation of the first route.

In most cases, the average increase in latency is less than 500 ms. You can monitor the latency using **Routing: message latency for messages/events** or **d2c.endpoints.latency.builtIn.events** IoT Hub metric. Creating or deleting any route after the first one does not impact the end-to-end latency.

Monitoring and troubleshooting

IoT Hub provides several routing and endpoint related metrics to give you an overview of the health of your hub and messages sent. You can combine information from multiple metrics to identify root cause for issues. For example, use metric **Routing: telemetry messages dropped** or **d2c.telemetry.egress.dropped** to identify the number of messages that were dropped when they didn't match queries on any of the routes and fallback route was disabled. [IoT Hub metrics](#) lists all metrics that are enabled by default for your IoT Hub.

You can use the REST API [Get Endpoint Health](#) to get [health status](#) of the endpoints. We recommend using the [IoT Hub metrics](#) related to routing message latency to identify and debug errors when endpoint health is dead or unhealthy. For example, for endpoint type Event Hubs, you can monitor **d2c.endpoints.latency.eventHubs**. The status of an unhealthy endpoint will be updated to healthy when IoT Hub has established an eventually consistent state of health.

Using the **routes** diagnostic logs in Azure Monitor [diagnostic settings](#), you can tracks errors that occur during evaluation of a routing query and endpoint health as perceived by IoT Hub, for example when an endpoint is dead. These diagnostic logs can be sent to Azure Monitor logs, Event Hubs, or Azure Storage for custom processing.

Next steps

- To learn how to create Message Routes, see [Process IoT Hub device-to-cloud messages using routes](#).
- [How to send device-to-cloud messages](#)
- For information about the SDKs you can use to send device-to-cloud messages, see [Azure IoT SDKs](#).

Create and read IoT Hub messages

3/13/2019 • 4 minutes to read

To support seamless interoperability across protocols, IoT Hub defines a common message format for all device-facing protocols. This message format is used for both [device-to-cloud routing](#) and [cloud-to-device](#) messages.

NOTE

Some of the features mentioned in this article, like cloud-to-device messaging, device twins, and device management, are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

IoT Hub implements device-to-cloud messaging using a streaming messaging pattern. IoT Hub's device-to-cloud messages are more like [Event Hubs events](#) than [Service Bus messages](#) in that there is a high volume of events passing through the service that can be read by multiple readers.

An IoT Hub message consists of:

- A predetermined set of *system properties* as listed below.
- A set of *application properties*. A dictionary of string properties that the application can define and access, without needing to deserialize the message body. IoT Hub never modifies these properties.
- An opaque binary body.

Property names and values can only contain ASCII alphanumeric characters, plus

{'!', '#', '\$', '%', '&', '^', '*', '+', '-', '.', '^', '_', ',', '|', '~'} when you send device-to-cloud messages using the HTTPS protocol or send cloud-to-device messages.

Device-to-cloud messaging with IoT Hub has the following characteristics:

- Device-to-cloud messages are durable and retained in an IoT hub's default **messages/events** endpoint for up to seven days.
- Device-to-cloud messages can be at most 256 KB, and can be grouped in batches to optimize sends. Batches can be at most 256 KB.
- IoT Hub does not allow arbitrary partitioning. Device-to-cloud messages are partitioned based on their originating **deviceId**.
- As explained in [Control access to IoT Hub](#), IoT Hub enables per-device authentication and access control.

For more information about how to encode and decode messages sent using different protocols, see [Azure IoT SDKs](#).

The following table lists the set of system properties in IoT Hub messages.

PROPERTY	DESCRIPTION	IS USER SETTABLE?
----------	-------------	-------------------

PROPERTY	DESCRIPTION	IS USER SETTABLE?
message-id	A user-settable identifier for the message used for request-reply patterns. Format: A case-sensitive string (up to 128 characters long) of ASCII 7-bit alphanumeric characters + <div style="border: 1px solid black; padding: 2px; display: inline-block;">{'-', ':', '.', '+', '%', '_', '#', '*', '?', '!', '(', ')', ',', '='}, '@', ';', '\$', '''}</div> .	Yes
sequence-number	A number (unique per device-queue) assigned by IoT Hub to each cloud-to-device message.	No for C2D messages; yes otherwise.
to	A destination specified in Cloud-to-Device messages.	No for C2D messages; yes otherwise.
absolute-expiry-time	Date and time of message expiration.	Yes
iothub-queuedtime	Date and time the Cloud-to-Device message was received by IoT Hub.	No for C2D messages; yes otherwise.
correlation-id	A string property in a response message that typically contains the MessageId of the request, in request-reply patterns.	Yes
user-id	An ID used to specify the origin of messages. When messages are generated by IoT Hub, it is set to <div style="border: 1px solid black; padding: 2px; display: inline-block;">{iot hub name}</div> .	No
iothub-ack	A feedback message generator. This property is used in cloud-to-device messages to request IoT Hub to generate feedback messages as a result of the consumption of the message by the device. Possible values: none (default): no feedback message is generated, positive : receive a feedback message if the message was completed, negative : receive a feedback message if the message expired (or maximum delivery count was reached) without being completed by the device, or full : both positive and negative.	Yes
iothub-connection-device-id	An ID set by IoT Hub on device-to-cloud messages. It contains the deviceId of the device that sent the message.	No for D2C messages; yes otherwise.
iothub-connection-auth-generation-id	An ID set by IoT Hub on device-to-cloud messages. It contains the generationId (as per Device identity properties) of the device that sent the message.	No for D2C messages; yes otherwise.

PROPERTY	DESCRIPTION	IS USER SETTABLE?
iothub-connection-auth-method	An authentication method set by IoT Hub on device-to-cloud messages. This property contains information about the authentication method used to authenticate the device sending the message.	No for D2C messages; yes otherwise.
iothub-creation-time-utc	Date and time the message was created on a device. A device must set this value explicitly.	Yes

Message size

IoT Hub measures message size in a protocol-agnostic way, considering only the actual payload. The size in bytes is calculated as the sum of the following:

- The body size in bytes.
- The size in bytes of all the values of the message system properties.
- The size in bytes of all user property names and values.

Property names and values are limited to ASCII characters, so the length of the strings equals the size in bytes.

Anti-spoofing properties

To avoid device spoofing in device-to-cloud messages, IoT Hub stamps all messages with the following properties:

- **iothub-connection-device-id**
- **iothub-connection-auth-generation-id**
- **iothub-connection-auth-method**

The first two contain the **deviceId** and **generationId** of the originating device, as per [Device identity properties](#).

The **iothub-connection-auth-method** property contains a JSON serialized object, with the following properties:

```
{
  "scope": "{ hub | device }",
  "type": "{ symkey | sas | x509 }",
  "issuer": "iothub"
}
```

Next steps

- For information about message size limits in IoT Hub, see [IoT Hub quotas and throttling](#).
- To learn how to create and read IoT Hub messages in various programming languages, see the [Quickstarts](#).

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

3/6/2019 • 3 minutes to read

By default, messages are routed to the built-in service-facing endpoint (**messages/events**) that is compatible with [Event Hubs](#). This endpoint is currently only exposed using the [AMQP](#) protocol on port 5671. An IoT hub exposes the following properties to enable you to control the built-in Event Hub-compatible messaging endpoint **messages/events**.

PROPERTY	DESCRIPTION
Partition count	Set this property at creation to define the number of partitions for device-to-cloud event ingestion.
Retention time	This property specifies how long in days messages are retained by IoT Hub. The default is one day, but it can be increased to seven days.

IoT Hub allows data retention in the built-in Event Hubs for a maximum of 7 days. You can set the retention time during creation of your IoT Hub. Data retention size in IoT Hub depends on your IoT hub tier and unit type. In terms of size, the built-in Event Hubs can retain messages of the maximum message size up to at least 24 hours of quota. For example, for 1 S1 unit IoT Hub provides enough storage to retain at least 400K messages of 4k size each. If your devices are sending smaller messages, they may be retained for longer (up to 7 days) depending on how much storage is consumed. We guarantee retaining the data for the specified retention time as a minimum.

IoT Hub also enables you to manage consumer groups on the built-in device-to-cloud receive endpoint.

If you're using [message routing](#) and the [fallback route](#) is enabled, all messages that don't match a query on any route go to the built-in endpoint. If you disable this fallback route, messages that don't match any query are dropped.

You can modify the retention time, either programmatically using the [IoT Hub resource provider REST APIs](#), or with the [Azure portal](#).

IoT Hub exposes the **messages/events** built-in endpoint for your back-end services to read the device-to-cloud messages received by your hub. This endpoint is Event Hub-compatible, which enables you to use any of the mechanisms the Event Hubs service supports for reading messages.

Read from the built-in endpoint

Some product integrations and Event Hubs SDKs are aware of IoT Hub and let you use your IoT hub service connection string to connect to the built-in endpoint.

When you use Event Hubs SDKs or product integrations that are unaware of IoT Hub, you need an Event Hub-compatible endpoint and Event Hub-compatible name. You can retrieve these values from the portal as follows:

1. Sign in to the [Azure portal](#) and navigate to your IoT hub.
2. Click **Built-in endpoints**.
3. The **Events** section contains the following values: **Partitions**, **Event Hub-compatible name**, **Event Hub-compatible endpoint**, **Retention time**, and **Consumer groups**.

In the portal, the Event Hub-compatible endpoint field contains a complete Event Hubs connection string that looks like:

Endpoint=sb://abcd1234namespace.servicebus.windows.net/;SharedAccessKeyName=iothubowner;SharedAccessKey=keykeykeykeykeykeykey=;EntityPath=iothub-ehub-abcd-1234-123456. If the SDK you're using requires other values, then they would be:

NAME	VALUE
Endpoint	sb://abcd1234namespace.servicebus.windows.net/
Hostname	abcd1234namespace.servicebus.windows.net
Namespace	abcd1234namespace

You can then use any shared access policy that has the **ServiceConnect** permissions to connect to the specified Event Hub.

The SDKs you can use to connect to the built-in Event Hub-compatible endpoint that IoT Hub exposes include:

LANGUAGE	SDK	EXAMPLE	NOTES
.NET	https://github.com/Azure/azure-event-hubs-dotnet	Quickstart	Uses Event Hubs-compatible information
Java	https://github.com/Azure/azure-event-hubs-java	Quickstart	Uses Event Hubs-compatible information
Node.js	https://github.com/Azure/azure-event-hubs-node	Quickstart	Uses IoT Hub connection string
Python	https://github.com/Azure/azure-event-hubs-python	https://github.com/Azure/azure-event-hubs-python/blob/master/examples/iothub_recv.py	Uses IoT Hub connection string

The product integrations you can use with the built-in Event Hub-compatible endpoint that IoT Hub exposes include:

- [Azure Functions](#). See [Processing data from IoT Hub with Azure Functions](#).
- [Azure Stream Analytics](#). See [Stream data as input into Stream Analytics](#).

- [Time Series Insights](#). See [Add an IoT hub event source to your Time Series Insights environment](#).
- [Apache Storm spout](#). You can view the [spout source](#) on GitHub.
- [Apache Spark integration](#).
- [Azure Databricks](#).

Next steps

- For more information about IoT Hub endpoints, see [IoT Hub endpoints](#).
- The [Quickstarts](#) show you how to send device-to-cloud messages from simulated devices and read the messages from the built-in endpoint.

For more detail, see the [Process IoT Hub device-to-cloud messages using routes](#) tutorial.

- If you want to route your device-to-cloud messages to custom endpoints, see [Use message routes and custom endpoints for device-to-cloud messages](#).

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

2/28/2019 • 2 minutes to read

NOTE

Some of the features mentioned in this article, like cloud-to-device messaging, device twins, and device management, are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

IoT Hub [Message Routing](#) enables users to route device-to-cloud messages to service-facing endpoints. Routing also provides a querying capability to filter the data before routing it to the endpoints. Each routing query you configure has the following properties:

PROPERTY	DESCRIPTION
Name	The unique name that identifies the query.
Source	The origin of the data stream to be acted upon. For example, device telemetry.
Condition	The query expression for the routing query that is run against the message application properties, system properties, message body, device twin tags, and device twin properties to determine if it is a match for the endpoint. For more information about constructing a query, see the see message routing query syntax
Endpoint	The name of the endpoint where IoT Hub sends messages that match the query. We recommend that you choose an endpoint in the same region as your IoT hub.

A single message may match the condition on multiple routing queries, in which case IoT Hub delivers the message to the endpoint associated with each matched query. IoT Hub also automatically deduplicates message delivery, so if a message matches multiple queries that have the same destination, it is only written once to that destination.

Endpoints and routing

An IoT hub has a default [built-in endpoint](#). You can create custom endpoints to route messages to by linking other services in your subscription to the hub. IoT Hub currently supports Azure Storage containers, Event Hubs, Service Bus queues, and Service Bus topics as custom endpoints.

When you use routing and custom endpoints, messages are only delivered to the built-in endpoint if they don't match any query. To deliver messages to the built-in endpoint as well as to a custom endpoint, add a route that sends messages to the built-in **events** endpoint.

NOTE

- IoT Hub only supports writing data to Azure Storage containers as blobs.
- Service Bus queues and topics with **Sessions** or **Duplicate Detection** enabled are not supported as custom endpoints.

For more information about creating custom endpoints in IoT Hub, see [IoT Hub endpoints](#).

For more information about reading from custom endpoints, see:

- Reading from [Azure Storage containers](#).
- Reading from [Event Hubs](#).
- Reading from [Service Bus queues](#).
- Reading from [Service Bus topics](#).

Next steps

- For more information about IoT Hub endpoints, see [IoT Hub endpoints](#).
- For more information about the query language you use to define routing queries, see [Message Routing query syntax](#).
- The [Process IoT Hub device-to-cloud messages using routes](#) tutorial shows you how to use routing queries and custom endpoints.

IoT Hub message routing query syntax

2/5/2019 • 5 minutes to read

Message routing enables users to route different data types namely, device telemetry messages, device lifecycle events, and device twin change events to various endpoints. You can also apply rich queries to this data before routing it to receive the data that matters to you. This article describes the IoT Hub message routing query language, and provides some common query patterns.

NOTE

Some of the features mentioned in this article, like cloud-to-device messaging, device twins, and device management, are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

Message routing allows you to query on the message properties and message body as well as device twin tags and device twin properties. If the message body is not JSON, message routing can still route the message, but queries cannot be applied to the message body. Queries are described as Boolean expressions where a Boolean true makes the query succeed which routes all the incoming data, and Boolean false fails the query and no data is routed. If the expression evaluates to null or undefined, it is treated as false and an error will be generated in diagnostic logs in case of a failure. The query syntax must be correct for the route to be saved and evaluated.

Message routing query based on message properties

The IoT Hub defines a [common format](#) for all device-to-cloud messaging for interoperability across protocols. IoT Hub message assumes the following JSON representation of the message. System properties are added for all users and identify content of the message. Users can selectively add application properties to the message. We recommend using unique property names as IoT Hub device-to-cloud messaging is not case-sensitive. For example, if you have multiple properties with the same name, IoT Hub will only send one of the properties.

```
{  
  "message": {  
    "systemProperties": {  
      "contentType": "application/json",  
      "contentEncoding": "UTF-8",  
      "iothub-message-source": "deviceMessages",  
      "iothub-enqueuedtime": "2017-05-08T18:55:31.851465Z"  
    },  
    "appProperties": {  
      "processingPath": "{cold | warm | hot}",  
      "verbose": "{true, false}",  
      "severity": 1-5,  
      "testDevice": "{true | false}"  
    },  
    "body": "{\"Weather\":{\"Temperature\":50}}"  
  }  
}
```

System properties

System properties help identify contents and source of the messages.

PROPERTY	TYPE	DESCRIPTION
contentType	string	The user specifies the content type of the message. To allow query on the message body, this value should be set application/JSON.
contentEncoding	string	The user specifies the encoding type of the message. Allowed values are UTF-8, UTF-16, UTF-32 if the contentType is set to application/JSON.
iohub-connection-device-id	string	This value is set by IoT Hub and identifies the ID of the device. To query, use <code>\$connectionDeviceId</code> .
iohub-enqueuedtime	string	This value is set by IoT Hub and represents the actual time of enqueueing the message in UTC. To query, use <code>enqueuedTime</code> .

As described in the [IoT Hub Messages](#), there are additional system properties in a message. In addition to **contentType**, **contentEncoding**, and **enqueuedTime**, the **connectionDeviceId** and **connectionModuleId** can also be queried.

Application properties

Application properties are user-defined strings that can be added to the message. These fields are optional.

Query expressions

A query on message system properties needs to be prefixed with the `$` symbol. Queries on application properties are accessed with their name and should not be prefixed with the `$` symbol. If an application property name begins with `$`, then IoT Hub will search for it in the system properties, and it is not found, then it will look in the application properties. For example:

To query on system property contentEncoding

```
$contentEncoding = 'UTF-8'
```

To query on application property processingPath:

```
processingPath = 'hot'
```

To combine these queries, you can use Boolean expressions and functions:

```
$contentEncoding = 'UTF-8' AND processingPath = 'hot'
```

A full list of supported operators and functions are listed in [Expression and conditions](#)

Message routing query based on message body

To enable querying on message body, the message should be in a JSON encoded in either UTF-8, UTF-16 or UTF-32. The `contentType` must be set to `application/JSON` and `contentEncoding` to one of the supported UTF encodings in the system property. If these properties are not specified, IoT Hub will not evaluate the query expression on the message body.

The following example shows how to create a message with a properly formed and encoded JSON body:

```
var messageBody = JSON.stringify(Object.assign({}, {
    "Weather": {
        "Temperature": 50,
        "Time": "2017-03-09T00:00:00.000Z",
        "PrevTemperatures": [
            20,
            30,
            40
        ],
        "IsEnabled": true,
        "Location": {
            "Street": "One Microsoft Way",
            "City": "Redmond",
            "State": "WA"
        },
        "HistoricalData": [
            {
                "Month": "Feb",
                "Temperature": 40
            },
            {
                "Month": "Jan",
                "Temperature": 30
            }
        ]
    }
}));
// Encode message body using UTF-8
var messageBytes = Buffer.from(messageBody, "utf8");

var message = new Message(messageBytes);

// Set message body type and content encoding
message.contentEncoding = "utf-8";
message.contentType = "application/json";

// Add other custom application properties
message.properties.add("Status", "Active");

deviceClient.sendEvent(message, (err, res) => {
    if (err) console.log('error: ' + err.toString());
    if (res) console.log('status: ' + res.constructor.name);
});
```

Query expressions

A query on message body needs to be prefixed with the `$body`. You can use a body reference, body array reference, or multiple body references in the query expression. Your query expression can also combine a body reference with message system properties, and message application properties reference. For example, the following are all valid query expressions:

```
$body.Weather.HistoricalData[0].Month = 'Feb'
```

```
$body.Weather.Temperature = 50 AND $body.Weather.IsEnabled
```

```
length($body.Weather.Location.State) = 2
```

```
$body.Weather.Temperature = 50 AND processingPath = 'hot'
```

Message routing query based on device twin

Message routing enables you to query on [Device Twin](#) tags and properties, which are JSON objects. Note that querying on module twin is not supported. A sample of Device Twin tags and properties is shown below.

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

Query expressions

A query on message body needs to be prefixed with the `$twin`. Your query expression can also combine a twin tag or property reference with a body reference, message system properties, and message application properties reference. We recommend using unique names in tags and properties as the query is not case-sensitive. Also refrain from using `twin`, `$twin`, `body`, or `$body`, as a property names. For example, the following are all valid query expressions:

```
$twin.properties.desired.telemetryConfig.sendFrequency = '5m'
```

```
$body.Weather.Temperature = 50 AND $twin.properties.desired.telemetryConfig.sendFrequency = '5m'
```

```
$twin.tags.deploymentLocation.floor = 1
```

Next steps

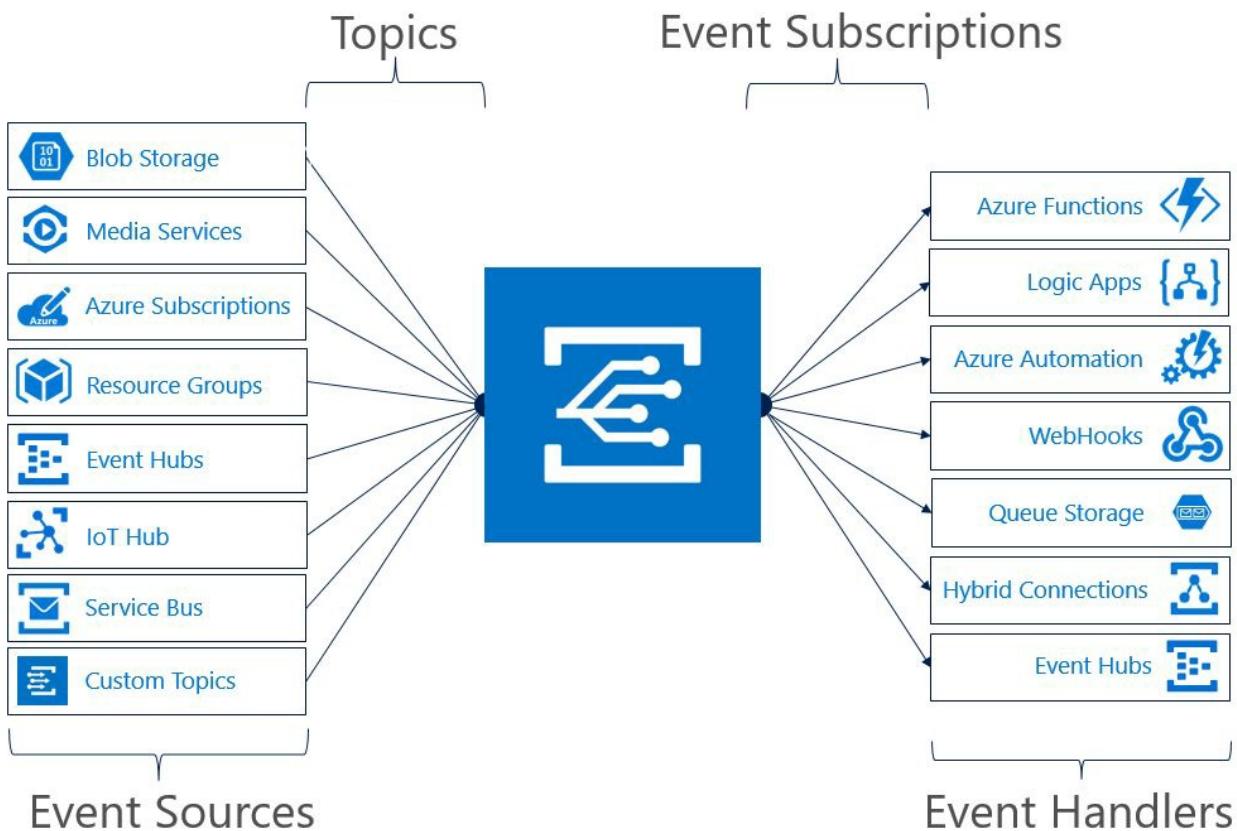
- Learn about [message routing](#).
- Try the [message routing tutorial](#).

React to IoT Hub events by using Event Grid to trigger actions

2/22/2019 • 4 minutes to read

Azure IoT Hub integrates with Azure Event Grid so that you can send event notifications to other services and trigger downstream processes. Configure your business applications to listen for IoT Hub events so that you can react to critical events in a reliable, scalable, and secure manner. For example, build an application that updates a database, creates a work ticket, and delivers an email notification every time a new IoT device is registered to your IoT hub.

[Azure Event Grid](#) is a fully managed event routing service that uses a publish-subscribe model. Event Grid has built-in support for Azure services like [Azure Functions](#) and [Azure Logic Apps](#), and can deliver event alerts to non-Azure services using webhooks. For a complete list of the event handlers that Event Grid supports, see [An introduction to Azure Event Grid](#).



Regional availability

The Event Grid integration is available for IoT hubs located in the regions where Event Grid is supported. For the latest list of regions, see [An introduction to Azure Event Grid](#).

Event types

IoT Hub publishes the following event types:

Event Type	Description
Microsoft.Devices.DeviceCreated	Published when a device is registered to an IoT hub.
Microsoft.Devices.DeviceDeleted	Published when a device is deleted from an IoT hub.
Microsoft.Devices.DeviceConnected	Published when a device is connected to an IoT hub.
Microsoft.Devices.DeviceDisconnected	Published when a device is disconnected from an IoT hub.

Use either the Azure portal or Azure CLI to configure which events to publish from each IoT hub. For an example, try the tutorial [Send email notifications about Azure IoT Hub events using Logic Apps](#).

Event schema

IoT Hub events contain all the information you need to respond to changes in your device lifecycle. You can identify an IoT Hub event by checking that the `eventType` property starts with **Microsoft.Devices**. For more information about how to use Event Grid event properties, see the [Event Grid event schema](#).

Device connected schema

The following example shows the schema of a device connected event:

Device created schema

The following example shows the schema of a device created event:

```
[{
  "id": "56afc886-767b-d359-d59e-0da7877166b2",
  "topic": "/SUBSCRIPTIONS/<subscription ID>/RESOURCEGROUPS/<resource group name>/PROVIDERS/MICROSOFT.DEVICES/IOTHUBS/<hub name>",
  "subject": "devices/LogicAppTestDevice",
  "eventType": "Microsoft.Devices.DeviceCreated",
  "eventTime": "2018-01-02T19:17:44.4383997Z",
  "data": {
    "twin": {
      "deviceId": "LogicAppTestDevice",
      "etag": "AAAAAAAAAAE=",
      "deviceEtag": "null",
      "status": "enabled",
      "statusUpdateTime": "2001-01-01T00:00:00",
      "connectionState": "Disconnected",
      "lastActivityTime": "2001-01-01T00:00:00",
      "cloudToDeviceMessageCount": 0,
      "authenticationType": "sas",
      "x509Thumbprint": {
        "primaryThumbprint": null,
        "secondaryThumbprint": null
      },
      "version": 2,
      "properties": {
        "desired": {
          "$metadata": {
            "$lastUpdated": "2018-01-02T19:17:44.4383997Z"
          },
          "$version": 1
        },
        "reported": {
          "$metadata": {
            "$lastUpdated": "2018-01-02T19:17:44.4383997Z"
          },
          "$version": 1
        }
      }
    },
    "hubName": "egtesthub1",
    "deviceId": "LogicAppTestDevice"
  },
  "dataVersion": "1",
  "metadataVersion": "1"
}]
}
```

For a detailed description of each property, see [Azure Event Grid event schema for IoT Hub](#).

Filter events

IoT Hub event subscriptions can filter events based on event type and device name. Subject filters in Event Grid work based on **Begins With** (prefix) and **Ends With** (suffix) matches. The filter uses an **AND** operator, so events with a subject that match both the prefix and suffix are delivered to the subscriber.

The subject of IoT Events uses the format:

```
devices/{deviceId}
```

Limitations for device connected and device disconnected events

To receive device connected and device disconnected events, you must open the D2C link or C2D link for your device. If your device is using MQTT protocol, IoT Hub will keep the C2D link open. For AMQP, you can open the

C2D link by calling the [Receive Async API](#).

The D2C link is open if you are sending telemetry. If the device connection is flickering, which means the device connects and disconnects frequently, we will not send every single connection state, but will publish the connection state that is snapshotted every minute. In case of an IoT Hub outage, we will publish the device connection state as soon as the outage is over. If the device disconnects during that outage, the device disconnected event will be published within 10 minutes.

Tips for consuming events

Applications that handle IoT Hub events should follow these suggested practices:

- Multiple subscriptions can be configured to route events to the same event handler, so don't assume that events are from a particular source. Always check the message topic to ensure that it comes from the IoT hub that you expect.
- Don't assume that all events you receive are the types that you expect. Always check the eventType before processing the message.
- Messages can arrive out of order or after a delay. Use the etag field to understand if your information about objects is up-to-date.

Next steps

- [Try the IoT Hub events tutorial](#)
- [Learn how to order device connected and disconnected events](#)
- [Learn more about Event Grid](#)
- [Compare the differences between routing IoT Hub events and messages](#)

Send cloud-to-device messages from IoT Hub

2/28/2019 • 5 minutes to read

To send one-way notifications to the device app from your solution back end, send cloud-to-devices messages from your IoT hub to your device. For a discussion of other cloud-to-devices options supported by IoT Hub, see [Cloud-to-device communications guidance](#).

NOTE

The features described in this article are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

You send cloud-to-device messages through a service-facing endpoint ([/messages/devicebound](#)). A device then receives the messages through a device-specific endpoint ([/devices/{deviceId}/messages/devicebound](#)).

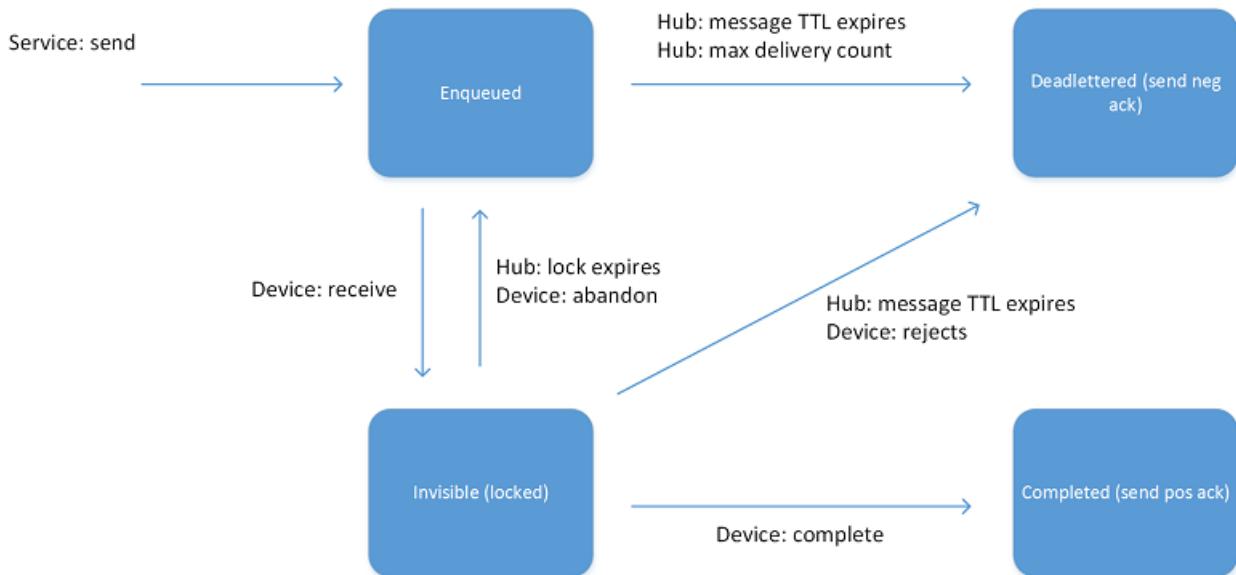
To target each cloud-to-device message at a single device, IoT Hub sets the **to** property to [/devices/{deviceId}/messages/devicebound](#).

Each device queue holds at most 50 cloud-to-device messages. Trying to send more messages to the same device results in an error.

The cloud-to-device message lifecycle

To guarantee at-least-once message delivery, IoT Hub persists cloud-to-device messages in per-device queues. Devices must explicitly acknowledge *completion* for IoT Hub to remove them from the queue. This approach guarantees resiliency against connectivity and device failures.

The following diagram shows the lifecycle state graph for a cloud-to-device message in IoT Hub.



When the IoT Hub service sends a message to a device, the service sets the message state to **Enqueued**. When a device wants to *receive* a message, IoT Hub *locks* the message (by setting the state to **Invisible**), which allows other threads on the device to start receiving other messages. When a device thread completes the processing of a message, it notifies IoT Hub by *completing* the message. IoT Hub then sets the state to **Completed**.

A device can also choose to:

- *Reject* the message, which causes IoT Hub to set it to the **Dead lettered** state. Devices that connect over the MQTT protocol cannot reject cloud-to-device messages.
- *Abandon* the message, which causes IoT Hub to put the message back in the queue, with the state set to **Enqueued**. Devices that connect over the MQTT protocol cannot abandon cloud-to-device messages.

A thread could fail to process a message without notifying IoT Hub. In this case, messages automatically transition from the **Invisible** state back to the **Enqueued** state after a *visibility (or lock) timeout*. The default value of this timeout is one minute.

The **max delivery count** property on IoT Hub determines the maximum number of times a message can transition between the **Enqueued** and **Invisible** states. After that number of transitions, IoT Hub sets the state of the message to **Dead lettered**. Similarly, IoT Hub sets the state of a message to **Dead lettered** after its expiration time (see [Time to live](#)).

The [How to send cloud-to-device messages with IoT Hub](#) shows you how to send cloud-to-device messages from the cloud and receive them on a device.

Typically, a device completes a cloud-to-device message when the loss of the message does not affect the application logic. For example, when the device has persisted the message content locally or has successfully executed an operation. The message could also carry transient information, whose loss would not impact the functionality of the application. Sometimes, for long-running tasks, you can:

- Complete the cloud-to-device message after persisting the task description in local storage.
- Notify the solution back end with one or more device-to-cloud messages at various stages of progress of the task.

Message expiration (time to live)

Every cloud-to-device message has an expiration time. This time is set by one of:

- The **ExpiryTimeUtc** property in the service.
- IoT Hub using the default *time to live* specified as an IoT Hub property.

See [Cloud-to-device configuration options](#).

A common way to take advantage of message expiration and avoid sending messages to disconnected devices, is to set short time to live values. This approach achieves the same result as maintaining the device connection state, while being more efficient. When you request message acknowledgements, IoT Hub notifies you which devices are:

- Able to receive messages.
- Are not online or have failed.

Message feedback

When you send a cloud-to-device message, the service can request the delivery of per-message feedback regarding the final state of that message.

ACK PROPERTY	BEHAVIOR
positive	If the cloud-to-device message reaches the Completed state, IoT Hub generates a feedback message.
negative	If the cloud-to-device message reaches the Dead lettered state, IoT Hub generates a feedback message.

ACK PROPERTY	BEHAVIOR
full	IoT Hub generates a feedback message in either case.

If **Ack** is **full**, and you don't receive a feedback message, it means that the feedback message expired. The service can't know what happened to the original message. In practice, a service should ensure that it can process the feedback before it expires. The maximum expiry time is two days, which leaves time to get the service running again if a failure occurs.

As explained in [Endpoints](#), IoT Hub delivers feedback through a service-facing endpoint (**/messages/servicebound/feedback**) as messages. The semantics for receiving feedback are the same as for cloud-to-device messages. Whenever possible, message feedback is batched in a single message, with the following format:

PROPERTY	DESCRIPTION
EnqueuedTime	Timestamp indicating when the feedback message was received by the hub.
UserId	{iot hub name}
ContentType	application/vnd.microsoft.iothub.feedback.json

The body is a JSON-serialized array of records, each with the following properties:

PROPERTY	DESCRIPTION
EnqueuedTimeUtc	Timestamp indicating when the outcome of the message happened. For example, the hub received the feedback message or the original message expired.
OriginalMessageId	MessageId of the cloud-to-device message to which this feedback information relates.
StatusCode	Required string. Used in feedback messages generated by IoT Hub. 'Success' 'Expired' 'DeliveryCountExceeded' 'Rejected' 'Purged'
Description	String values for StatusCode .
DeviceId	DeviceId of the target device of the cloud-to-device message to which this piece of feedback relates.
DeviceGenerationId	DeviceGenerationId of the target device of the cloud-to-device message to which this piece of feedback relates.

The service must specify a **MessageId** for the cloud-to-device message to be able to correlate its feedback with the original message.

The following example shows the body of a feedback message.

```
[
  {
    "OriginalMessageId": "0987654321",
    "EnqueuedTimeUtc": "2015-07-28T16:24:48.789Z",
    "StatusCode": 0,
    "Description": "Success",
    "DeviceId": "123",
    "DeviceGenerationId": "abcdefghijklmnopqrstuvwxyz"
  },
  {
    ...
  },
  ...
]
```

Cloud-to-device configuration options

Each IoT hub exposes the following configuration options for cloud-to-device messaging:

PROPERTY	DESCRIPTION	RANGE AND DEFAULT
defaultTtlAsIso8601	Default TTL for cloud-to-device messages.	ISO_8601 interval up to 2D (minimum 1 minute). Default: 1 hour.
maxDeliveryCount	Maximum delivery count for cloud-to-device per-device queues.	1 to 100. Default: 10.
feedback.ttlAsIso8601	Retention for service-bound feedback messages.	ISO_8601 interval up to 2D (minimum 1 minute). Default: 1 hour.
feedback.maxDeliveryCount	Maximum delivery count for feedback queue.	1 to 100. Default: 100.

For more information about how to set these configuration options, see [Create IoT hubs](#).

Next steps

For information about the SDKs you can use to receive cloud-to-device messages, see [Azure IoT SDKs](#).

To try out receiving cloud-to-device messages, see the [Send cloud-to-device](#) tutorial.

Reference - choose a communication protocol

3/6/2019 • 2 minutes to read

IoT Hub allows devices to use the following protocols for device-side communications:

- [MQTT](#)
- MQTT over WebSockets
- [AMQP](#)
- AMQP over WebSockets
- HTTPS

For information about how these protocols support specific IoT Hub features, see [Device-to-cloud communications guidance](#) and [Cloud-to-device communications guidance](#).

The following table provides the high-level recommendations for your choice of protocol:

PROTOCOL	WHEN YOU SHOULD CHOOSE THIS PROTOCOL
MQTT MQTT over WebSocket	Use on all devices that do not require to connect multiple devices (each with its own per-device credentials) over the same TLS connection.
AMQP AMQP over WebSocket	Use on field and cloud gateways to take advantage of connection multiplexing across devices.
HTTPS	Use for devices that cannot support other protocols.

Consider the following points when you choose your protocol for device-side communications:

- **Cloud-to-device pattern.** HTTPS does not have an efficient way to implement server push. As such, when you are using HTTPS, devices poll IoT Hub for cloud-to-device messages. This approach is inefficient for both the device and IoT Hub. Under current HTTPS guidelines, each device should poll for messages every 25 minutes or more. MQTT and AMQP support server push when receiving cloud-to-device messages. They enable immediate pushes of messages from IoT Hub to the device. If delivery latency is a concern, MQTT or AMQP are the best protocols to use. For rarely connected devices, HTTPS works as well.
- **Field gateways.** When using MQTT and HTTPS, you cannot connect multiple devices (each with its own per-device credentials) using the same TLS connection. For [Field gateway scenarios](#) that require one TLS connection between the field gateway and IoT Hub for each connected device, these protocols are suboptimal.
- **Low resource devices.** The MQTT and HTTPS libraries have a smaller footprint than the AMQP libraries. As such, if the device has limited resources (for example, less than 1-MB RAM), these protocols might be the only protocol implementation available.
- **Network traversal.** The standard AMQP protocol uses port 5671, and MQTT listens on port 8883. Use of these ports could cause problems in networks that are closed to non-HTTPS protocols. Use MQTT over WebSockets, AMQP over WebSockets, or HTTPS in this scenario.
- **Payload size.** MQTT and AMQP are binary protocols, which result in more compact payloads than HTTPS.

WARNING

When using HTTPS, each device should poll for cloud-to-device messages every 25 minutes or more. However, during development, it is acceptable to poll more frequently than every 25 minutes.

Port numbers

Devices can communicate with IoT Hub in Azure using various protocols. Typically, the choice of protocol is driven by the specific requirements of the solution. The following table lists the outbound ports that must be open for a device to be able to use a specific protocol:

PROTOCOL	PORT
MQTT	8883
MQTT over WebSockets	443
AMQP	5671
AMQP over WebSockets	443
HTTPS	443

Once you have created an IoT hub in an Azure region, the IoT hub keeps the same IP address for the lifetime of that IoT hub. However, if Microsoft moves the IoT hub to a different scale unit to maintain quality of service, then it is assigned a new IP address.

Next steps

To learn more about how IoT Hub implements the MQTT protocol, see [Communicate with your IoT hub using the MQTT protocol](#).

Upload files with IoT Hub

2/28/2019 • 5 minutes to read

As detailed in the [IoT Hub endpoints](#) article, a device can start a file upload by sending a notification through a device-facing endpoint ([/devices/{deviceId}/files](#)). When a device notifies IoT Hub that an upload is complete, IoT Hub sends a file upload notification message through the [/messages/servicebound/filenotifications](#) service-facing endpoint.

Instead of brokering messages through IoT Hub itself, IoT Hub instead acts as a dispatcher to an associated Azure Storage account. A device requests a storage token from IoT Hub that is specific to the file the device wishes to upload. The device uses the SAS URI to upload the file to storage, and when the upload is complete the device sends a notification of completion to IoT Hub. IoT Hub checks the file upload is complete and then adds a file upload notification message to the service-facing file notification endpoint.

Before you upload a file to IoT Hub from a device, you must configure your hub by [associating an Azure Storage account to it](#).

Your device can then [initialize an upload](#) and then [notify IoT hub](#) when the upload completes. Optionally, when a device notifies IoT Hub that the upload is complete, the service can generate a [notification message](#).

When to use

Use file upload to send media files and large telemetry batches uploaded by intermittently connected devices or compressed to save bandwidth.

Refer to [Device-to-cloud communication guidance](#) if in doubt between using reported properties, device-to-cloud messages, or file upload.

Associate an Azure Storage account with IoT Hub

To use the file upload functionality, you must first link an Azure Storage account to the IoT Hub. You can complete this task either through the Azure portal, or programmatically through the [IoT Hub resource provider REST APIs](#). Once you've associated an Azure Storage account with your IoT Hub, the service returns a SAS URI to a device when the device starts a file upload request.

The [Upload files from your device to the cloud with IoT Hub](#) how-to guides provide a complete walkthrough of the file upload process. These how-to guides show you how to use the Azure portal to associate a storage account with an IoT hub.

NOTE

The [Azure IoT SDKs](#) automatically handle retrieving the SAS URI, uploading the file, and notifying IoT Hub of a completed upload.

Initialize a file upload

IoT Hub has an endpoint specifically for devices to request a SAS URI for storage to upload a file. To start the file upload process, the device sends a POST request to `{iot_hub}.azure-devices.net/devices/{deviceId}/files` with the following JSON body:

```
{  
    "blobName": "{name of the file for which a SAS URI will be generated}"  
}
```

IoT Hub returns the following data, which the device uses to upload the file:

```
{  
    "correlationId": "somecorrelationid",  
    "hostName": "yourstorageaccount.blob.core.windows.net",  
    "containerName": "testcontainer",  
    "blobName": "test-device1/image.jpg",  
    "sasToken": "1234asdfSASToken"  
}
```

Deprecated: initialize a file upload with a GET

NOTE

This section describes deprecated functionality for how to receive a SAS URI from IoT Hub. Use the POST method described previously.

IoT Hub has two REST endpoints to support file upload, one to get the SAS URI for storage and the other to notify the IoT hub of a completed upload. The device starts the file upload process by sending a GET to the IoT hub at `{iot hub}.azure-devices.net/devices/{deviceId}/files/{filename}`. The IoT hub returns:

- A SAS URI specific to the file to be uploaded.
- A correlation ID to be used once the upload is completed.

Notify IoT Hub of a completed file upload

The device uploads the file to storage using the Azure Storage SDKs. When the upload is complete, the device sends a POST request to `{iot hub}.azure-devices.net/devices/{deviceId}/files/notifications` with the following JSON body:

```
{  
    "correlationId": "{correlation ID received from the initial request}",  
    "isSuccess": bool,  
    "statusCode": XXX,  
    "statusDescription": "Description of status"  
}
```

The value of `isSuccess` is a Boolean that indicates whether the file was uploaded successfully. The status code for `statusCode` is the status for the upload of the file to storage, and the `statusDescription` corresponds to the `statusCode`.

Reference topics:

The following reference topics provide you with more information about uploading files from a device.

File upload notifications

Optionally, when a device notifies IoT Hub that an upload is complete, IoT Hub generates a notification message. This message contains the name and storage location of the file.

As explained in [Endpoints](#), IoT Hub delivers file upload notifications through a service-facing endpoint (**/messages/servicebound/fileuploadnotifications**) as messages. The receive semantics for file upload notifications are the same as for cloud-to-device messages and have the same [message lifecycle](#). Each message retrieved from the file upload notification endpoint is a JSON record with the following properties:

PROPERTY	DESCRIPTION
EnqueuedTimeUtc	Timestamp indicating when the notification was created.
DeviceId	DeviceId of the device which uploaded the file.
BlobUri	URI of the uploaded file.
BlobName	Name of the uploaded file.
LastUpdatedTime	Timestamp indicating when the file was last updated.
BlobSizeInBytes	Size of the uploaded file.

Example. This example shows the body of a file upload notification message.

```
{
  "deviceId": "mydevice",
  "blobUri": "https://{{storage account}}.blob.core.windows.net/{{container name}}/mydevice/myfile.jpg",
  "blobName": "mydevice/myfile.jpg",
  "lastUpdatedTime": "2016-06-01T21:22:41+00:00",
  "blobSizeInBytes": 1234,
  "enqueuedTimeUtc": "2016-06-01T21:22:43.7996883Z"
}
```

File upload notification configuration options

Each IoT hub has the following configuration options for file upload notifications:

PROPERTY	DESCRIPTION	RANGE AND DEFAULT
enableFileUploadNotifications	Controls whether file upload notifications are written to the file notifications endpoint.	Bool. Default: True.
fileNotifications.ttlAsIso8601	Default TTL for file upload notifications.	ISO_8601 interval up to 48H (minimum 1 minute). Default: 1 hour.
fileNotifications.lockDuration	Lock duration for the file upload notifications queue.	5 to 300 seconds (minimum 5 seconds). Default: 60 seconds.
fileNotifications.maxDeliveryCount	Maximum delivery count for the file upload notification queue.	1 to 100. Default: 100.

Additional reference material

Other reference topics in the IoT Hub developer guide include:

- [IoT Hub endpoints](#) describes the various IoT hub endpoints for run-time and management operations.

- [Throttling and quotas](#) describes the quotas and throttling behaviors that apply to the IoT Hub service.
- [Azure IoT device and service SDKs](#) lists the various language SDKs you can use when you develop both device and service apps that interact with IoT Hub.
- [IoT Hub query language](#) describes the query language you can use to retrieve information from IoT Hub about your device twins and jobs.
- [IoT Hub MQTT support](#) provides more information about IoT Hub support for the MQTT protocol.

Next steps

Now you've learned how to upload files from devices using IoT Hub, you may be interested in the following IoT Hub developer guide topics:

- [Manage device identities in IoT Hub](#)
- [Control access to IoT Hub](#)
- [Use device twins to synchronize state and configurations](#)
- [Invoke a direct method on a device](#)
- [Schedule jobs on multiple devices](#)

To try out some of the concepts described in this article, see the following IoT Hub tutorial:

- [How to upload files from devices to the cloud with IoT Hub](#)

Understand the identity registry in your IoT hub

2/28/2019 • 11 minutes to read

Every IoT hub has an identity registry that stores information about the devices and modules permitted to connect to the IoT hub. Before a device or module can connect to an IoT hub, there must be an entry for that device or module in the IoT hub's identity registry. A device or module must also authenticate with the IoT hub based on credentials stored in the identity registry.

The device or module ID stored in the identity registry is case-sensitive.

At a high level, the identity registry is a REST-capable collection of device or module identity resources. When you add an entry in the identity registry, IoT Hub creates a set of per-device resources such as the queue that contains in-flight cloud-to-device messages.

Use the identity registry when you need to:

- Provision devices or modules that connect to your IoT hub.
- Control per-device/per-module access to your hub's device or module-facing endpoints.

NOTE

- The identity registry does not contain any application-specific metadata.
- Module identity and module twin is in public preview. Below feature will be supported on module identity when it's general available.

Identity registry operations

The IoT Hub identity registry exposes the following operations:

- Create device or module identity
- Update device or module identity
- Retrieve device or module identity by ID
- Delete device or module identity
- List up to 1000 identities
- Export device identities to Azure blob storage
- Import device identities from Azure blob storage

All these operations can use optimistic concurrency, as specified in [RFC7232](#).

IMPORTANT

The only way to retrieve all identities in an IoT hub's identity registry is to use the [Export](#) functionality.

An IoT Hub identity registry:

- Does not contain any application metadata.
- Can be accessed like a dictionary, by using the **deviceId** or **moduleId** as the key.
- Does not support expressive queries.

An IoT solution typically has a separate solution-specific store that contains application-specific metadata.

For example, the solution-specific store in a smart building solution would record the room in which a temperature sensor is deployed.

IMPORTANT

Only use the identity registry for device management and provisioning operations. High throughput operations at run time should not depend on performing operations in the identity registry. For example, checking the connection state of a device before sending a command is not a supported pattern. Make sure to check the [throttling rates](#) for the identity registry, and the [device heartbeat](#) pattern.

Disable devices

You can disable devices by updating the **status** property of an identity in the identity registry. Typically, you use this property in two scenarios:

- During a provisioning orchestration process. For more information, see [Device Provisioning](#).
- If, for any reason, you think a device is compromised or has become unauthorized.

This feature is not available for modules.

Import and export device identities

Use asynchronous operations on the [IoT Hub resource provider endpoint](#) to export device identities in bulk from an IoT hub's identity registry. Exports are long-running jobs that use a customer-supplied blob container to save device identity data read from the identity registry.

Use asynchronous operations on the [IoT Hub resource provider endpoint](#) to import device identities in bulk to an IoT hub's identity registry. Imports are long-running jobs that use data in a customer-supplied blob container to write device identity data into the identity registry.

For more information about the import and export APIs, see [IoT Hub resource provider REST APIs](#). To learn more about running import and export jobs, see [Bulk management of IoT Hub device identities](#).

Device provisioning

The device data that a given IoT solution stores depends on the specific requirements of that solution. But, as a minimum, a solution must store device identities and authentication keys. Azure IoT Hub includes an identity registry that can store values for each device such as IDs, authentication keys, and status codes. A solution can use other Azure services such as table storage, blob storage, or Cosmos DB to store any additional device data.

Device provisioning is the process of adding the initial device data to the stores in your solution. To enable a new device to connect to your hub, you must add a device ID and keys to the IoT Hub identity registry. As part of the provisioning process, you might need to initialize device-specific data in other solution stores. You can also use the Azure IoT Hub Device Provisioning Service to enable zero-touch, just-in-time provisioning to one or more IoT hubs without requiring human intervention. To learn more, see the [provisioning service documentation](#).

Device heartbeat

The IoT Hub identity registry contains a field called **connectionState**. Only use the **connectionState** field during development and debugging. IoT solutions should not query the field at run time. For example, do not query the **connectionState** field to check if a device is connected before you send a cloud-to-device message or an SMS. We recommend subscribing to the [device disconnected event](#) on Event Grid to get

alerts and monitor the device connection state. Use this [tutorial](#) to learn how to integrate Device Connected and Device Disconnected events from IoT Hub in your IoT solution.

If your IoT solution needs to know if a device is connected, you can implement the *heartbeat pattern*. In the heartbeat pattern, the device sends device-to-cloud messages at least once every fixed amount of time (for example, at least once every hour). Therefore, even if a device does not have any data to send, it still sends an empty device-to-cloud message (usually with a property that identifies it as a heartbeat). On the service side, the solution maintains a map with the last heartbeat received for each device. If the solution does not receive a heartbeat message within the expected time from the device, it assumes that there is a problem with the device.

A more complex implementation could include the information from [Azure Monitor](#) and [Azure Resource Health](#) to identify devices that are trying to connect or communicate but failing, check [Monitor with diagnostics](#) guide. When you implement the heartbeat pattern, make sure to check [IoT Hub Quotas and Throttles](#).

NOTE

If an IoT solution uses the connection state solely to determine whether to send cloud-to-device messages, and messages are not broadcast to large sets of devices, consider using the simpler *short expiry time* pattern. This pattern achieves the same result as maintaining a device connection state registry using the heartbeat pattern, while being more efficient. If you request message acknowledgements, IoT Hub can notify you about which devices are able to receive messages and which are not.

Device and module lifecycle notifications

IoT Hub can notify your IoT solution when an identity is created or deleted by sending lifecycle notifications. To do so, your IoT solution needs to create a route and to set the Data Source equal to *DeviceLifecycleEvents* or *ModuleLifecycleEvents*. By default, no lifecycle notifications are sent, that is, no such routes pre-exist. The notification message includes properties, and body.

Properties: Message system properties are prefixed with the `$` symbol.

Notification message for device:

NAME	VALUE
<code>\$content-type</code>	application/json
<code>\$iothub-enqueuedtime</code>	Time when the notification was sent
<code>\$iothub-message-source</code>	deviceLifecycleEvents
<code>\$content-encoding</code>	utf-8
<code>opType</code>	createDeviceIdentity or deleteDeviceIdentity
<code>hubName</code>	Name of IoT Hub
<code>deviceId</code>	ID of the device
<code>operationTimestamp</code>	ISO8601 timestamp of operation
<code>iothub-message-schema</code>	deviceLifecycleNotification

Body: This section is in JSON format and represents the twin of the created device identity. For example,

```
{  
    "deviceId": "11576-ailn-test-0-67333793211",  
    "etag": "AAAAAAAEEAE=",  
    "properties": {  
        "desired": {  
            "$metadata": {  
                "$lastUpdated": "2016-02-30T16:24:48.789Z"  
            },  
            "$version": 1  
        },  
        "reported": {  
            "$metadata": {  
                "$lastUpdated": "2016-02-30T16:24:48.789Z"  
            },  
            "$version": 1  
        }  
    }  
}
```

Notification message for module:

NAME	VALUE
\$content-type	application/json
\$iothub-enqueuedtime	Time when the notification was sent
\$iothub-message-source	moduleLifecycleEvents
\$content-encoding	utf-8
opType	createModuleIdentity or deleteModuleIdentity
hubName	Name of IoT Hub
moduleId	ID of the module
operationTimestamp	ISO8601 timestamp of operation
iothub-message-schema	moduleLifecycleNotification

Body: This section is in JSON format and represents the twin of the created module identity. For example,

```
{
    "deviceId": "11576-ailn-test-0-67333793211",
    "moduleId": "tempSensor",
    "etag": "AAAAAAAAAAE=",
    "properties": {
        "desired": {
            "$metadata": {
                "$lastUpdated": "2016-02-30T16:24:48.789Z"
            },
            "$version": 1
        },
        "reported": {
            "$metadata": {
                "$lastUpdated": "2016-02-30T16:24:48.789Z"
            },
            "$version": 1
        }
    }
}
```

Device identity properties

Device identities are represented as JSON documents with the following properties:

PROPERTY	OPTIONS	DESCRIPTION
deviceId	required, read-only on updates	A case-sensitive string (up to 128 characters long) of ASCII 7-bit alphanumeric characters plus certain special characters: - . + % _ # * ? ! () , = @ \$ ' '
generationId	required, read-only	An IoT hub-generated, case-sensitive string up to 128 characters long. This value is used to distinguish devices with the same deviceId , when they have been deleted and re-created.
etag	required, read-only	A string representing a weak ETag for the device identity, as per RFC7232 .
auth	optional	A composite object containing authentication information and security materials.
auth.symkey	optional	A composite object containing a primary and a secondary key, stored in base64 format.
status	required	An access indicator. Can be Enabled or Disabled . If Enabled , the device is allowed to connect. If Disabled , this device cannot access any device-facing endpoint.

PROPERTY	OPTIONS	DESCRIPTION
statusReason	optional	A 128 character-long string that stores the reason for the device identity status. All UTF-8 characters are allowed.
statusUpdateTime	read-only	A temporal indicator, showing the date and time of the last status update.
connectionState	read-only	A field indicating connection status: either Connected or Disconnected . This field represents the IoT Hub view of the device connection status. Important: This field should be used only for development/debugging purposes. The connection state is updated only for devices using MQTT or AMQP. Also, it is based on protocol-level pings (MQTT pings, or AMQP pings), and it can have a maximum delay of only 5 minutes. For these reasons, there can be false positives, such as devices reported as connected but that are disconnected.
connectionStateUpdatedTime	read-only	A temporal indicator, showing the date and last time the connection state was updated.
lastActivityTime	read-only	A temporal indicator, showing the date and last time the device connected, received, or sent a message.

NOTE

Connection state can only represent the IoT Hub view of the status of the connection. Updates to this state may be delayed, depending on network conditions and configurations.

NOTE

Currently the device SDKs do not support using the `+` and `#` characters in the **deviceId**.

Module identity properties

Module identities are represented as JSON documents with the following properties:

PROPERTY	OPTIONS	DESCRIPTION
----------	---------	-------------

PROPERTY	OPTIONS	DESCRIPTION
deviceId	required, read-only on updates	A case-sensitive string (up to 128 characters long) of ASCII 7-bit alphanumeric characters plus certain special characters: - . + % _ # * ? ! () , = @ \$ ' '
moduleId	required, read-only on updates	A case-sensitive string (up to 128 characters long) of ASCII 7-bit alphanumeric characters plus certain special characters: - . + % _ # * ? ! () , = @ \$ ' '
generationId	required, read-only	An IoT hub-generated, case-sensitive string up to 128 characters long. This value is used to distinguish devices with the same deviceId , when they have been deleted and re-created.
etag	required, read-only	A string representing a weak ETag for the device identity, as per RFC7232 .
auth	optional	A composite object containing authentication information and security materials.
auth.symkey	optional	A composite object containing a primary and a secondary key, stored in base64 format.
status	required	An access indicator. Can be Enabled or Disabled . If Enabled , the device is allowed to connect. If Disabled , this device cannot access any device-facing endpoint.
statusReason	optional	A 128 character-long string that stores the reason for the device identity status. All UTF-8 characters are allowed.
statusUpdateTime	read-only	A temporal indicator, showing the date and time of the last status update.

PROPERTY	OPTIONS	DESCRIPTION
connectionState	read-only	A field indicating connection status: either Connected or Disconnected . This field represents the IoT Hub view of the device connection status. Important: This field should be used only for development/debugging purposes. The connection state is updated only for devices using MQTT or AMQP. Also, it is based on protocol-level pings (MQTT pings, or AMQP pings), and it can have a maximum delay of only 5 minutes. For these reasons, there can be false positives, such as devices reported as connected but that are disconnected.
connectionStateUpdatedTime	read-only	A temporal indicator, showing the date and last time the connection state was updated.
lastActivityTime	read-only	A temporal indicator, showing the date and last time the device connected, received, or sent a message.

NOTE

Currently the device SDKs do not support using the `[+]` and `[#]` characters in the **deviceId** and **moduleId**.

Additional reference material

Other reference topics in the IoT Hub developer guide include:

- [IoT Hub endpoints](#) describes the various endpoints that each IoT hub exposes for run-time and management operations.
- [Throttling and quotas](#) describes the quotas and throttling behaviors that apply to the IoT Hub service.
- [Azure IoT device and service SDKs](#) lists the various language SDKs you can use when you develop both device and service apps that interact with IoT Hub.
- [IoT Hub query language](#) describes the query language you can use to retrieve information from IoT Hub about your device twins and jobs.
- [IoT Hub MQTT support](#) provides more information about IoT Hub support for the MQTT protocol.

Next steps

Now that you have learned how to use the IoT Hub identity registry, you may be interested in the following IoT Hub developer guide topics:

- [Control access to IoT Hub](#)
- [Use device twins to synchronize state and configurations](#)
- [Invoke a direct method on a device](#)

- [Schedule jobs on multiple devices](#)

To try out some of the concepts described in this article, see the following IoT Hub tutorial:

- [Get started with Azure IoT Hub](#)

To explore using the IoT Hub Device Provisioning Service to enable zero-touch, just-in-time provisioning, see:

- [Azure IoT Hub Device Provisioning Service](#)

Control access to IoT Hub

3/6/2019 • 17 minutes to read

This article describes the options for securing your IoT hub. IoT Hub uses *permissions* to grant access to each IoT hub endpoint. Permissions limit the access to an IoT hub based on functionality.

This article introduces:

- The different permissions that you can grant to a device or back-end app to access your IoT hub.
- The authentication process and the tokens it uses to verify permissions.
- How to scope credentials to limit access to specific resources.
- IoT Hub support for X.509 certificates.
- Custom device authentication mechanisms that use existing device identity registries or authentication schemes.

NOTE

Some of the features mentioned in this article, like cloud-to-device messaging, device twins, and device management, are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

You must have appropriate permissions to access any of the IoT Hub endpoints. For example, a device must include a token containing security credentials along with every message it sends to IoT Hub.

Access control and permissions

You can grant [permissions](#) in the following ways:

- **IoT hub-level shared access policies.** Shared access policies can grant any combination of [permissions](#). You can define policies in the [Azure portal](#), programmatically by using the [IoT Hub Resource REST APIs](#), or using the `az iot hub policy` CLI. A newly created IoT hub has the following default policies:

SHARED ACCESS POLICY	PERMISSIONS
iothubowner	All permission
service	ServiceConnect permissions
device	DeviceConnect permissions
registryRead	RegistryRead permissions
registryReadWrite	RegistryRead and RegistryWrite permissions

- **Per-Device Security Credentials.** Each IoT Hub contains an [identity registry](#). For each device in this identity registry, you can configure security credentials that grant **DeviceConnect** permissions scoped to the corresponding device endpoints.

For example, in a typical IoT solution:

- The device management component uses the *registryReadWrite* policy.
- The event processor component uses the *service* policy.
- The run-time device business logic component uses the *service* policy.
- Individual devices connect using credentials stored in the IoT hub's identity registry.

NOTE

See [permissions](#) for detailed information.

Authentication

Azure IoT Hub grants access to endpoints by verifying a token against the shared access policies and identity registry security credentials.

Security credentials, such as symmetric keys, are never sent over the wire.

NOTE

The Azure IoT Hub resource provider is secured through your Azure subscription, as are all providers in the [Azure Resource Manager](#).

For more information about how to construct and use security tokens, see [IoT Hub security tokens](#).

Protocol specifics

Each supported protocol, such as MQTT, AMQP, and HTTPS, transports tokens in different ways.

When using MQTT, the CONNECT packet has the deviceld as the ClientId, `{iothubhostname}/{deviceId}` in the Username field, and a SAS token in the Password field. `{iothubhostname}` should be the full CName of the IoT hub (for example, contoso.azure-devices.net).

When using AMQP, IoT Hub supports [SAS PLAIN](#) and [AMQP Claims-Based-Security](#).

If you use AMQP claims-based-security, the standard specifies how to transmit these tokens.

For SASL PLAIN, the **username** can be:

- `{policyName}@sas.root.{iothubName}` if using IoT hub-level tokens.
- `{deviceId}@sas.{iothubname}` if using device-scoped tokens.

In both cases, the password field contains the token, as described in [IoT Hub security tokens](#).

HTTPS implements authentication by including a valid token in the **Authorization** request header.

Example

Username (DeviceId is case-sensitive): `iothubname.azure-devices.net/DeviceId`

Password (You can generate a SAS token with the [device explorer](#) tool, the CLI extension command `az iot hub generate-sas-token`, or the [Azure IoT Tools for Visual Studio Code](#)):

```
SharedAccessSignature sr=iothubname.azure-devices.net%2fdevices%2fDeviceId&sig=kPszxZZZZZZZZZZZZAhLT%2bV7o%3d&se=1487709501
```

NOTE

The [Azure IoT SDKs](#) automatically generate tokens when connecting to the service. In some cases, the Azure IoT SDKs do not support all the protocols or all the authentication methods.

Special considerations for SASL PLAIN

When using SASL PLAIN with AMQP, a client connecting to an IoT hub can use a single token for each TCP connection. When the token expires, the TCP connection disconnects from the service and triggers a reconnection. This behavior, while not problematic for a back-end app, is damaging for a device app for the following reasons:

- Gateways usually connect on behalf of many devices. When using SASL PLAIN, they have to create a distinct TCP connection for each device connecting to an IoT hub. This scenario considerably increases the consumption of power and networking resources, and increases the latency of each device connection.
- Resource-constrained devices are adversely affected by the increased use of resources to reconnect after each token expiration.

Scope IoT hub-level credentials

You can scope IoT hub-level security policies by creating tokens with a restricted resource URI. For example, the endpoint to send device-to-cloud messages from a device is **/devices/{deviceId}/messages/events**. You can also use an IoT hub-level shared access policy with **DeviceConnect** permissions to sign a token whose **resourceURI** is **/devices/{deviceId}**. This approach creates a token that is only usable to send messages on behalf of device **deviceId**.

This mechanism is similar to the [Event Hubs publisher policy](#), and enables you to implement custom authentication methods.

Security tokens

IoT Hub uses security tokens to authenticate devices and services to avoid sending keys on the wire. Additionally, security tokens are limited in time validity and scope. [Azure IoT SDKs](#) automatically generate tokens without requiring any special configuration. Some scenarios do require you to generate and use security tokens directly. Such scenarios include:

- The direct use of the MQTT, AMQP, or HTTPS surfaces.
- The implementation of the token service pattern, as explained in [Custom device authentication](#).

IoT Hub also allows devices to authenticate with IoT Hub using [X.509 certificates](#).

Security token structure

You use security tokens to grant time-bounded access to devices and services to specific functionality in IoT Hub. To get authorization to connect to IoT Hub, devices and services must send security tokens signed with either a shared access or symmetric key. These keys are stored with a device identity in the identity registry.

A token signed with a shared access key grants access to all the functionality associated with the shared access policy permissions. A token signed with a device identity's symmetric key only grants the **DeviceConnect** permission for the associated device identity.

The security token has the following format:

```
SharedAccessSignature sig={signature-string}&se={expiry}&skn={policyName}&sr={URL-encoded-resourceURI}
```

Here are the expected values:

VALUE	DESCRIPTION
-------	-------------

VALUE	DESCRIPTION
{signature}	An HMAC-SHA256 signature string of the form: <code>{URL-encoded-resourceURI} + "\n" + expiry</code> . Important: The key is decoded from base64 and used as key to perform the HMAC-SHA256 computation.
{resourceURI}	URI prefix (by segment) of the endpoints that can be accessed with this token, starting with host name of the IoT hub (no protocol). For example, <code>myHub.azure-devices.net/devices/device1</code>
{expiry}	UTF8 strings for number of seconds since the epoch 00:00:00 UTC on 1 January 1970.
{URL-encoded-resourceURI}	Lower case URL-encoding of the lower case resource URI
{policyName}	The name of the shared access policy to which this token refers. Absent if the token refers to device-registry credentials.

Note on prefix: The URI prefix is computed by segment and not by character. For example `/a/b` is a prefix for `/a/b/c` but not for `/a/bc`.

The following Node.js snippet shows a function called **generateSasToken** that computes the token from the inputs `resourceUri, signingKey, policyName, expiresInMins`. The next sections detail how to initialize the different inputs for the different token use cases.

```
var generateSasToken = function(resourceUri, signingKey, policyName, expiresInMins) {
    resourceUri = encodeURIComponent(resourceUri);

    // Set expiration in seconds
    var expires = (Date.now() / 1000) + expiresInMins * 60;
    expires = Math.ceil(expires);
    var toSign = resourceUri + '\n' + expires;

    // Use crypto
    var hmac = crypto.createHmac('sha256', new Buffer(signingKey, 'base64'));
    hmac.update(toSign);
    var base64UriEncoded = encodeURIComponent(hmac.digest('base64'));

    // Construct authorization string
    var token = "SharedAccessSignature sr=" + resourceUri + "&sig=";
    + base64UriEncoded + "&se=" + expires;
    if (policyName) token += "&skn=" + policyName;
    return token;
};
```

As a comparison, the equivalent Python code to generate a security token is:

```

from base64 import b64encode, b64decode
from hashlib import sha256
from time import time
from urllib import quote_plus, urlencode
from hmac import HMAC

def generate_sas_token(uri, key, policy_name, expiry=3600):
    ttl = time() + expiry
    sign_key = "%s\n%d" % ((quote_plus(uri)), int(ttl))
    print sign_key
    signature = b64encode(HMAC(b64decode(key), sign_key, sha256).digest())

    rawtoken = {
        'sr' : uri,
        'sig': signature,
        'se' : str(int(ttl))
    }

    if policy_name is not None:
        rawtoken['skn'] = policy_name

    return 'SharedAccessSignature ' + urlencode(rawtoken)

```

The functionality in C# to generate a security token is:

```

using System;
using System.Globalization;
using System.Net;
using System.Net.Http;
using System.Security.Cryptography;
using System.Text;

public static string generateSasToken(string resourceUri, string key, string policyName, int
expiryInSeconds = 3600)
{
    TimeSpan fromEpochStart = DateTime.UtcNow - new DateTime(1970, 1, 1);
    string expiry = Convert.ToString((int)fromEpochStart.TotalSeconds + expiryInSeconds);

    string stringToSign = WebUtility.UrlEncode(resourceUri) + "\n" + expiry;

    HMACSHA256 hmac = new HMACSHA256(Convert.FromBase64String(key));
    string signature = Convert.ToBase64String(hmac.ComputeHash(Encoding.UTF8.GetBytes(stringToSign)));

    string token = String.Format(CultureInfo.InvariantCulture, "SharedAccessSignature sr={0}&sig={1}&se=
{2}", WebUtility.UrlEncode(resourceUri), WebUtility.UrlEncode(signature), expiry);

    if (!String.IsNullOrEmpty(policyName))
    {
        token += "&skn=" + policyName;
    }

    return token;
}

```

NOTE

Since the time validity of the token is validated on IoT Hub machines, the drift on the clock of the machine that generates the token must be minimal.

Use SAS tokens in a device app

There are two ways to obtain **DeviceConnect** permissions with IoT Hub with security tokens: use a [symmetric device key from the identity registry](#), or use a [shared access key](#).

Remember that all functionality accessible from devices is exposed by design on endpoints with prefix

`/devices/{deviceId}`.

IMPORTANT

The only way that IoT Hub authenticates a specific device is using the device identity symmetric key. In cases when a shared access policy is used to access device functionality, the solution must consider the component issuing the security token as a trusted subcomponent.

The device-facing endpoints are (irrespective of the protocol):

ENDPOINT	FUNCTIONALITY
<code>{iot hub host name}/devices/{deviceId}/messages/events</code>	Send device-to-cloud messages.
<code>{iot hub host name}/devices/{deviceId}/messages/devicebound</code>	Receive cloud-to-device messages.

Use a symmetric key in the identity registry

When using a device identity's symmetric key to generate a token, the `policyName` (`skn`) element of the token is omitted.

For example, a token created to access all device functionality should have the following parameters:

- resource URI: `{IoT hub name}.azure-devices.net/devices/{device id}`,
- signing key: any symmetric key for the `{device id}` identity,
- no policy name,
- any expiration time.

An example using the preceding Node.js function would be:

```
var endpoint ="myhub.azure-devices.net/devices/device1";
var deviceKey ="...";

var token = generateSasToken(endpoint, deviceKey, null, 60);
```

The result, which grants access to all functionality for device1, would be:

```
SharedAccessSignature sr=myhub.azure-
devices.net%2fdevices%2fdevice1&sig=13y8ejUk2z7PLmvtwR5RqlGBOVwiq7rQR3WZ5xZX3N4%3D&se=1456971697
```

NOTE

It's possible to generate a SAS token with the [device explorer](#) tool, the CLI extension command [az iot hub generate-sas-token](#), or the [Azure IoT Tools for Visual Studio Code](#).

Use a shared access policy

When you create a token from a shared access policy, set the `skn` field to the name of the policy. This policy must grant the **DeviceConnect** permission.

The two main scenarios for using shared access policies to access device functionality are:

- [cloud protocol gateways](#),

- **token services** used to implement custom authentication schemes.

Since the shared access policy can potentially grant access to connect as any device, it is important to use the correct resource URI when creating security tokens. This setting is especially important for token services, which have to scope the token to a specific device using the resource URI. This point is less relevant for protocol gateways as they are already mediating traffic for all devices.

As an example, a token service using the pre-created shared access policy called **device** would create a token with the following parameters:

- resource URI: `{IoT hub name}.azure-devices.net/devices/{device id}`,
- signing key: one of the keys of the `device` policy,
- policy name: `device`,
- any expiration time.

An example using the preceding Node.js function would be:

```
var endpoint = "myhub.azure-devices.net/devices/device1";
var policyName = 'device';
var policyKey = '...';

var token = generateSasToken(endpoint, policyKey, policyName, 60);
```

The result, which grants access to all functionality for device1, would be:

```
SharedAccessSignature sr=myhub.azure-
devices.net%2fdevices%2fdevice1&sig=13y8ejUk2z7PLmvtwR5RqlGB0Vwiq7rQR3WZ5xZX3N4%3D&se=1456971697&skn=device
```

A protocol gateway could use the same token for all devices simply setting the resource URI to `myhub.azure-devices.net/devices`.

Use security tokens from service components

Service components can only generate security tokens using shared access policies granting the appropriate permissions as explained previously.

Here are the service functions exposed on the endpoints:

ENDPOINT	FUNCTIONALITY
<code>{iot hub host name}/devices</code>	Create, update, retrieve, and delete device identities.
<code>{iot hub host name}/messages/events</code>	Receive device-to-cloud messages.
<code>{iot hub host name}/servicebound/feedback</code>	Receive feedback for cloud-to-device messages.
<code>{iot hub host name}/devicebound</code>	Send cloud-to-device messages.

As an example, a service generating using the pre-created shared access policy called **registryRead** would create a token with the following parameters:

- resource URI: `{IoT hub name}.azure-devices.net/devices`,
- signing key: one of the keys of the `registryRead` policy,
- policy name: `registryRead`,
- any expiration time.

```
var endpoint = "myhub.azure-devices.net/devices";
var policyName = 'registryRead';
var policyKey = '...';

var token = generateSasToken(endpoint, policyKey, policyName, 60);
```

The result, which would grant access to read all device identities, would be:

```
SharedAccessSignature sr=myhub.azure-
devices.net%2fdevices&sig=JdyscqTpXdEJs49e1IUCCohw2D1FDR3zfH5KqGJo4r4%3D&se=1456973447&skn=registryRead
```

Supported X.509 certificates

You can use any X.509 certificate to authenticate a device with IoT Hub by uploading either a certificate thumbprint or a certificate authority (CA) to Azure IoT Hub. Authentication using certificate thumbprints only verifies that the presented thumbprint matches the configured thumbprint. Authentication using certificate authority validates the certificate chain.

Supported certificates include:

- **An existing X.509 certificate.** A device may already have an X.509 certificate associated with it. The device can use this certificate to authenticate with IoT Hub. Works with either thumbprint or CA authentication.
- **CA-signed X.509 certificate.** To identify a device and authenticate it with IoT Hub, you can use an X.509 certificate generated and signed by a Certification Authority (CA). Works with either thumbprint or CA authentication.
- **A self-generated and self-signed X.509 certificate.** A device manufacturer or in-house deployer can generate these certificates and store the corresponding private key (and certificate) on the device. You can use tools such as [OpenSSL](#) and [Windows SelfSignedCertificate](#) utility for this purpose. Only works with thumbprint authentication.

A device may either use an X.509 certificate or a security token for authentication, but not both.

For more information about authentication using certificate authority, see [Device Authentication using X.509 CA Certificates](#).

Register an X.509 certificate for a device

The [Azure IoT Service SDK for C#](#) (version 1.0.8+) supports registering a device that uses an X.509 certificate for authentication. Other APIs such as import/export of devices also support X.509 certificates.

You can also use the CLI extension command [az iot hub device-identity](#) to configure X.509 certificates for devices.

C# Support

The **RegistryManager** class provides a programmatic way to register a device. In particular, the **AddDeviceAsync** and **UpdateDeviceAsync** methods enable you to register and update a device in the IoT Hub identity registry. These two methods take a **Device** instance as input. The **Device** class includes an **Authentication** property that allows you to specify primary and secondary X.509 certificate thumbprints. The thumbprint represents a SHA256 hash of the X.509 certificate (stored using binary DER encoding). You have the option of specifying a primary thumbprint or a secondary thumbprint or both. Primary and secondary thumbprints are supported to handle certificate rollover scenarios.

Here is a sample C# code snippet to register a device using an X.509 certificate thumbprint:

```
var device = new Device(deviceId)
{
    Authentication = new AuthenticationMechanism()
    {
        X509Thumbprint = new X509Thumbprint()
        {
            PrimaryThumbprint = "B4172AB44C28F3B9E117648C6F7294978A00CDCBA34A46A1B8588B3F7D82C4F1"
        }
    }
};

RegistryManager registryManager =
RegistryManager.CreateFromConnectionString(deviceGatewayConnectionString);
await registryManager.AddDeviceAsync(device);
```

Use an X.509 certificate during run-time operations

The [Azure IoT device SDK for .NET](#) (version 1.0.11+) supports the use of X.509 certificates.

C# Support

The class **DeviceAuthenticationWithX509Certificate** supports the creation of **DeviceClient** instances using an X.509 certificate. The X.509 certificate must be in the PFX (also called PKCS #12) format that includes the private key.

Here is a sample code snippet:

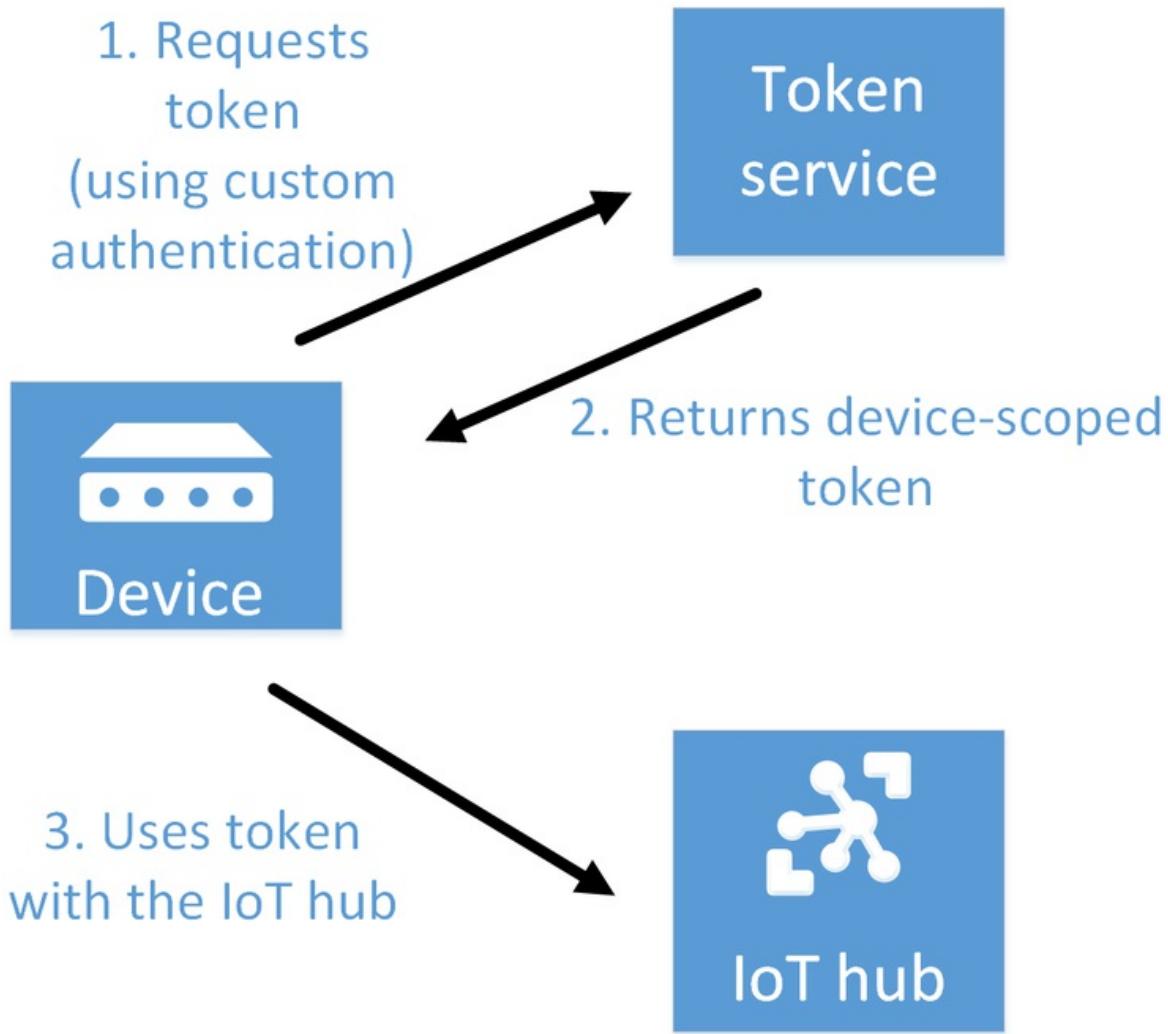
```
var authMethod = new DeviceAuthenticationWithX509Certificate("<device id>", x509Certificate);

var deviceClient = DeviceClient.Create("<IoTHub DNS HostName>", authMethod);
```

Custom device and module authentication

You can use the IoT Hub [identity registry](#) to configure per-device/module security credentials and access control using [tokens](#). If an IoT solution already has a custom identity registry and/or authentication scheme, consider creating a *token service* to integrate this infrastructure with IoT Hub. In this way, you can use other IoT features in your solution.

A token service is a custom cloud service. It uses an IoT Hub *shared access policy* with **DeviceConnect** or **ModuleConnect** permissions to create *device-scoped* or *module-scoped* tokens. These tokens enable a device and module to connect to your IoT hub.



Here are the main steps of the token service pattern:

1. Create an IoT Hub shared access policy with **DeviceConnect** or **ModuleConnect** permissions for your IoT hub. You can create this policy in the [Azure portal](#) or programmatically. The token service uses this policy to sign the tokens it creates.
2. When a device/module needs to access your IoT hub, it requests a signed token from your token service. The device can authenticate with your custom identity registry/authentication scheme to determine the device/module identity that the token service uses to create the token.
3. The token service returns a token. The token is created by using `/devices/{deviceId}` or `/devices/{deviceId}/module/{moduleId}` as `resourceURI`, with `deviceId` as the device being authenticated or `moduleId` as the module being authenticated. The token service uses the shared access policy to construct the token.
4. The device/module uses the token directly with the IoT hub.

NOTE

You can use the .NET class [SharedAccessSignatureBuilder](#) or the Java class [IoTHubServiceSasToken](#) to create a token in your token service.

The token service can set the token expiration as desired. When the token expires, the IoT hub severs the device/module connection. Then, the device/module must request a new token from the token service. A short expiry time increases the load on both the device/module and the token service.

For a device/module to connect to your hub, you must still add it to the IoT Hub identity registry — even though it is using a token and not a key to connect. Therefore, you can continue to use per-device/per-module access control by enabling or disabling device/module identities in the [identity registry](#). This approach mitigates the risks of using tokens with long expiry times.

Comparison with a custom gateway

The token service pattern is the recommended way to implement a custom identity registry/authentication scheme with IoT Hub. This pattern is recommended because IoT Hub continues to handle most of the solution traffic. However, if the custom authentication scheme is so intertwined with the protocol, you may require a *custom gateway* to process all the traffic. An example of such a scenario is using [Transport Layer Security \(TLS\)](#) and [pre-shared keys \(PSKs\)](#). For more information, see the [protocol gateway](#) article.

Reference topics:

The following reference topics provide you with more information about controlling access to your IoT hub.

IoT Hub permissions

The following table lists the permissions you can use to control access to your IoT hub.

PERMISSION	NOTES
RegistryRead	Grants read access to the identity registry. For more information, see Identity registry . This permission is used by back-end cloud services.
RegistryReadWrite	Grants read and write access to the identity registry. For more information, see Identity registry . This permission is used by back-end cloud services.
ServiceConnect	Grants access to cloud service-facing communication and monitoring endpoints. Grants permission to receive device-to-cloud messages, send cloud-to-device messages, and retrieve the corresponding delivery acknowledgments. Grants permission to retrieve delivery acknowledgements for file uploads. Grants permission to access twins to update tags and desired properties, retrieve reported properties, and run queries. This permission is used by back-end cloud services.
DeviceConnect	Grants access to device-facing endpoints. Grants permission to send device-to-cloud messages and receive cloud-to-device messages. Grants permission to perform file upload from a device. Grants permission to receive device twin desired property notifications and update device twin reported properties. Grants permission to perform file uploads. This permission is used by devices.

Additional reference material

Other reference topics in the IoT Hub developer guide include:

- [IoT Hub endpoints](#) describes the various endpoints that each IoT hub exposes for run-time and management operations.

- [Throttling and quotas](#) describes the quotas and throttling behaviors that apply to the IoT Hub service.
- [Azure IoT device and service SDKs](#) lists the various language SDKs you can use when you develop both device and service apps that interact with IoT Hub.
- [IoT Hub query language](#) describes the query language you can use to retrieve information from IoT Hub about your device twins and jobs.
- [IoT Hub MQTT support](#) provides more information about IoT Hub support for the MQTT protocol.

Next steps

Now that you have learned how to control access IoT Hub, you may be interested in the following IoT Hub developer guide topics:

- [Use device twins to synchronize state and configurations](#)
- [Invoke a direct method on a device](#)
- [Schedule jobs on multiple devices](#)

If you would like to try out some of the concepts described in this article, see the following IoT Hub tutorials:

- [Get started with Azure IoT Hub](#)
- [How to send cloud-to-device messages with IoT Hub](#)
- [How to process IoT Hub device-to-cloud messages](#)

Understand and use device twins in IoT Hub

2/28/2019 • 11 minutes to read

Device twins are JSON documents that store device state information including metadata, configurations, and conditions. Azure IoT Hub maintains a device twin for each device that you connect to IoT Hub.

NOTE

The features described in this article are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

This article describes:

- The structure of the device twin: *tags*, *desired* and *reported properties*.
- The operations that device apps and back ends can perform on device twins.

Use device twins to:

- Store device-specific metadata in the cloud. For example, the deployment location of a vending machine.
- Report current state information such as available capabilities and conditions from your device app. For example, a device is connected to your IoT hub over cellular or WiFi.
- Synchronize the state of long-running workflows between device app and back-end app. For example, when the solution back end specifies the new firmware version to install, and the device app reports the various stages of the update process.
- Query your device metadata, configuration, or state.

Refer to [Device-to-cloud communication guidance](#) for guidance on using reported properties, device-to-cloud messages, or file upload.

Refer to [Cloud-to-device communication guidance](#) for guidance on using desired properties, direct methods, or cloud-to-device messages.

Device twins

Device twins store device-related information that:

- Device and back ends can use to synchronize device conditions and configuration.
- The solution back end can use to query and target long-running operations.

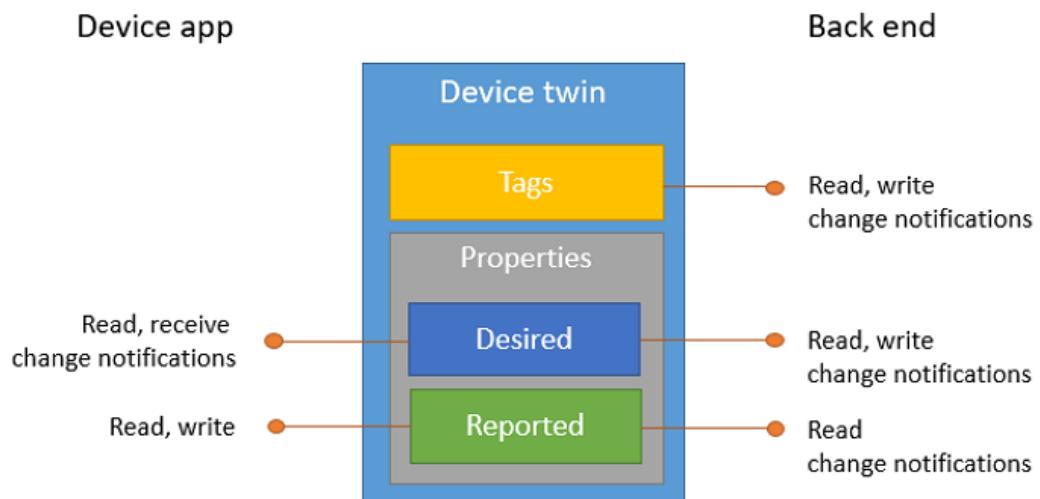
The lifecycle of a device twin is linked to the corresponding [device identity](#). Device twins are implicitly created and deleted when a device identity is created or deleted in IoT Hub.

A device twin is a JSON document that includes:

- **Tags.** A section of the JSON document that the solution back end can read from and write to. Tags are not visible to device apps.
- **Desired properties.** Used along with reported properties to synchronize device configuration or conditions. The solution back end can set desired properties, and the device app can read them. The

device app can also receive notifications of changes in the desired properties.

- **Reported properties.** Used along with desired properties to synchronize device configuration or conditions. The device app can set reported properties, and the solution back end can read and query them.
- **Device identity properties.** The root of the device twin JSON document contains the read-only properties from the corresponding device identity stored in the [identity registry](#).



The following example shows a device twin JSON document:

```
{
    "deviceId": "devA",
    "etag": "AAAAAAAAAAc=",
    "status": "enabled",
    "statusReason": "provisioned",
    "statusUpdateTime": "2001-01-01T00:00:00",
    "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
        }
    }
}
```

In the root object are the device identity properties, and container objects for `tags` and both `reported` and `desired` properties. The `properties` container contains some read-only elements (`$metadata`, `$etag`, and `$version`) described in the [Device twin metadata](#) and [Optimistic concurrency](#) sections.

Reported property example

In the previous example, the device twin contains a `batteryLevel` property that is reported by the device app. This property makes it possible to query and operate on devices based on the last reported battery level. Other examples include the device app reporting device capabilities or connectivity options.

NOTE

Reported properties simplify scenarios where the solution back end is interested in the last known value of a property. Use [device-to-cloud messages](#) if the solution back end needs to process device telemetry in the form of sequences of timestamped events, such as time series.

Desired property example

In the previous example, the `telemetryConfig` device twin desired and reported properties are used by the solution back end and the device app to synchronize the telemetry configuration for this device. For example:

1. The solution back end sets the desired property with the desired configuration value. Here is the portion of the document with the desired property set:

```
"desired": {  
    "telemetryConfig": {  
        "sendFrequency": "5m"  
    },  
    ...  
},
```

2. The device app is notified of the change immediately if connected, or at the first reconnect. The device app then reports the updated configuration (or an error condition using the `status` property). Here is the portion of the reported properties:

```
"reported": {  
    "telemetryConfig": {  
        "sendFrequency": "5m",  
        "status": "success"  
    },  
    ...  
}
```

3. The solution back end can track the results of the configuration operation across many devices by [querying](#) device twins.

NOTE

The preceding snippets are examples, optimized for readability, of one way to encode a device configuration and its status. IoT Hub does not impose a specific schema for the device twin desired and reported properties in the device twins.

You can use twins to synchronize long-running operations such as firmware updates. For more information on how to use properties to synchronize and track a long running operation across devices, see [Use desired properties to configure devices](#).

Back-end operations

The solution back end operates on the device twin using the following atomic operations, exposed through HTTPS:

- **Retrieve device twin by ID.** This operation returns the device twin document, including tags and desired and reported system properties.
- **Partially update device twin.** This operation enables the solution back end to partially update the tags or desired properties in a device twin. The partial update is expressed as a JSON document that adds or updates any property. Properties set to `null` are removed. The following example creates a new desired property with value `{"newProperty": "newValue"}`, overwrites the existing value of `existingProperty` with `"othernewValue"`, and removes `otherOldProperty`. No other changes are made to existing desired properties or tags:

```
{
  "properties": {
    "desired": {
      "newProperty": {
        "nestedProperty": "newValue"
      },
      "existingProperty": "otherNewValue",
      "otherOldProperty": null
    }
  }
}
```

- **Replace desired properties.** This operation enables the solution back end to completely overwrite all existing desired properties and substitute a new JSON document for `properties/desired`.
- **Replace tags.** This operation enables the solution back end to completely overwrite all existing tags and substitute a new JSON document for `tags`.
- **Receive twin notifications.** This operation allows the solution back end to be notified when the twin is modified. To do so, your IoT solution needs to create a route and to set the Data Source equal to `twinChangeEvents`. By default, no such routes pre-exist, so no twin notifications are sent. If the rate of change is too high, or for other reasons such as internal failures, the IoT Hub might send only one notification that contains all changes. Therefore, if your application needs reliable auditing and logging of all intermediate states, you should use device-to-cloud messages. The twin notification message includes properties and body.

- Properties

NAME	VALUE
\$content-type	application/json
\$iothub-enqueuedtime	Time when the notification was sent
\$iothub-message-source	twinChangeEvents
\$content-encoding	utf-8
deviceId	ID of the device
hubName	Name of IoT Hub
operationTimestamp	ISO8601 timestamp of operation
iothub-message-schema	deviceLifecycleNotification
opType	"replaceTwin" or "updateTwin"

Message system properties are prefixed with the `$` symbol.

- Body

This section includes all the twin changes in a JSON format. It uses the same format as a patch, with the difference that it can contain all twin sections: tags, properties.reported, properties.desired, and that it contains the "\$metadata" elements. For example,

```
{
  "properties": {
    "desired": {
      "$metadata": {
        "$lastUpdated": "2016-02-30T16:24:48.789Z"
      },
      "$version": 1
    },
    "reported": {
      "$metadata": {
        "$lastUpdated": "2016-02-30T16:24:48.789Z"
      },
      "$version": 1
    }
  }
}
```

All the preceding operations support [Optimistic concurrency](#) and require the **ServiceConnect** permission, as defined in [Control access to IoT Hub](#).

In addition to these operations, the solution back end can:

- Query the device twins using the SQL-like [IoT Hub query language](#).
- Perform operations on large sets of device twins using [jobs](#).

Device operations

The device app operates on the device twin using the following atomic operations:

- **Retrieve device twin.** This operation returns the device twin document (including tags and desired and reported system properties) for the currently connected device.
- **Partially update reported properties.** This operation enables the partial update of the reported properties of the currently connected device. This operation uses the same JSON update format that the solution back end uses for a partial update of desired properties.
- **Observe desired properties.** The currently connected device can choose to be notified of updates to the desired properties when they happen. The device receives the same form of update (partial or full replacement) executed by the solution back end.

All the preceding operations require the **DeviceConnect** permission, as defined in [Control Access to IoT Hub](#).

The [Azure IoT device SDKs](#) make it easy to use the preceding operations from many languages and platforms. For more information on the details of IoT Hub primitives for desired properties synchronization, see [Device reconnection flow](#).

Tags and properties format

Tags, desired properties, and reported properties are JSON objects with the following restrictions:

- All keys in JSON objects are case-sensitive 64 bytes UTF-8 UNICODE strings. Allowed characters exclude UNICODE control characters (segments C0 and C1), and `,`, `$`, and SP.
- All values in JSON objects can be of the following JSON types: boolean, number, string, object. Arrays are not allowed. The maximum value for integers is 4503599627370495 and the minimum value for integers is -4503599627370496.
- All JSON objects in tags, desired, and reported properties can have a maximum depth of 5. For

instance, the following object is valid:

```
{  
  ...  
  "tags": {  
    "one": {  
      "two": {  
        "three": {  
          "four": {  
            "five": {  
              "property": "value"  
            }  
          }  
        }  
      }  
    },  
    ...  
  }  
}
```

- All string values can be at most 512 bytes in length.

Device twin size

IoT Hub enforces an 8KB size limitation on each of the respective total values of `tags`, `properties/desired`, and `properties/reported`, excluding read-only elements.

The size is computed by counting all characters, excluding UNICODE control characters (segments C0 and C1) and spaces that are outside of string constants.

IoT Hub rejects with an error all operations that would increase the size of those documents above the limit.

Device twin metadata

IoT Hub maintains the timestamp of the last update for each JSON object in device twin desired and reported properties. The timestamps are in UTC and encoded in the [ISO8601](#) format

`YYYY-MM-DDTHH:MM:SS.mmmZ`.

For example:

```
{
  ...
  "properties": {
    "desired": {
      "telemetryConfig": {
        "sendFrequency": "5m"
      },
      "$metadata": {
        "telemetryConfig": {
          "sendFrequency": {
            "$lastUpdated": "2016-03-30T16:24:48.789Z"
          },
          "$lastUpdated": "2016-03-30T16:24:48.789Z"
        },
        "$lastUpdated": "2016-03-30T16:24:48.789Z"
      },
      "$version": 23
    },
    "reported": {
      "telemetryConfig": {
        "sendFrequency": "5m",
        "status": "success"
      }
      "batteryLevel": "55%",
      "$metadata": {
        "telemetryConfig": {
          "sendFrequency": "5m",
          "status": {
            "$lastUpdated": "2016-03-31T16:35:48.789Z"
          },
          "$lastUpdated": "2016-03-31T16:35:48.789Z"
        }
        "batteryLevel": {
          "$lastUpdated": "2016-04-01T16:35:48.789Z"
        },
        "$lastUpdated": "2016-04-01T16:24:48.789Z"
      },
      "$version": 123
    }
  }
  ...
}
```

This information is kept at every level (not just the leaves of the JSON structure) to preserve updates that remove object keys.

Optimistic concurrency

Tags, desired, and reported properties all support optimistic concurrency. Tags have an ETag, as per [RFC7232](#), that represents the tag's JSON representation. You can use ETags in conditional update operations from the solution back end to ensure consistency.

Device twin desired and reported properties do not have ETags, but have a `$version` value that is guaranteed to be incremental. Similarly to an ETag, the version can be used by the updating party to enforce consistency of updates. For example, a device app for a reported property or the solution back end for a desired property.

Versions are also useful when an observing agent (such as the device app observing the desired properties) must reconcile races between the result of a retrieve operation and an update notification. The [Device reconnection flow section](#) provides more information.

Device reconnection flow

IoT Hub does not preserve desired properties update notifications for disconnected devices. It follows that a device that is connecting must retrieve the full desired properties document, in addition to subscribing for update notifications. Given the possibility of races between update notifications and full retrieval, the following flow must be ensured:

1. Device app connects to an IoT hub.
2. Device app subscribes for desired properties update notifications.
3. Device app retrieves the full document for desired properties.

The device app can ignore all notifications with `$version` less or equal than the version of the full retrieved document. This approach is possible because IoT Hub guarantees that versions always increment.

NOTE

This logic is already implemented in the [Azure IoT device SDKs](#). This description is useful only if the device app cannot use any of Azure IoT device SDKs and must program the MQTT interface directly.

Additional reference material

Other reference topics in the IoT Hub developer guide include:

- The [IoT Hub endpoints](#) article describes the various endpoints that each IoT hub exposes for run-time and management operations.
- The [Throttling and quotas](#) article describes the quotas that apply to the IoT Hub service and the throttling behavior to expect when you use the service.
- The [Azure IoT device and service SDKs](#) article lists the various language SDKs you can use when you develop both device and service apps that interact with IoT Hub.
- The [IoT Hub query language for device twins, jobs, and message routing](#) article describes the IoT Hub query language you can use to retrieve information from IoT Hub about your device twins and jobs.
- The [IoT Hub MQTT support](#) article provides more information about IoT Hub support for the MQTT protocol.

Next steps

Now you have learned about device twins, you may be interested in the following IoT Hub developer guide topics:

- [Understand and use module twins in IoT Hub](#)
- [Invoke a direct method on a device](#)
- [Schedule jobs on multiple devices](#)

To try out some of the concepts described in this article, see the following IoT Hub tutorials:

- [How to use the device twin](#)
- [How to use device twin properties](#)
- [Device management with Azure IoT Tools for VS Code](#)

Understand and use module twins in IoT Hub

11/21/2018 • 10 minutes to read

This article assumes you've read [Understand and use device twins in IoT Hub](#) first. In IoT Hub, under each device identity, you can create up to 20 module identities. Each module identity implicitly generates a module twin. Similar to device twins, module twins are JSON documents that store module state information including metadata, configurations, and conditions. Azure IoT Hub maintains a module twin for each module that you connect to IoT Hub.

On the device side, the IoT Hub device SDKs enable you to create modules where each one opens an independent connection to IoT Hub. This functionality enables you to use separate namespaces for different components on your device. For example, you have a vending machine that has three different sensors. Each sensor is controlled by different departments in your company. You can create a module for each sensor. This way, each department is only able to send jobs or direct methods to the sensor that they control, avoiding conflicts and user errors.

Module identity and module twin provide the same capabilities as device identity and device twin but at a finer granularity. This finer granularity enables capable devices, such as operating system-based devices or firmware devices managing multiple components, to isolate configuration and conditions for each of those components. Module identity and module twins provide a management separation of concerns when working with IoT devices that have modular software components. We aim at supporting all the device twin functionality at module twin level by module twin general availability.

NOTE

The features described in this article are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

This article describes:

- The structure of the module twin: *tags*, *desired* and *reported properties*.
- The operations that the modules and back ends can perform on module twins.

Refer to [Device-to-cloud communication guidance](#) for guidance on using reported properties, device-to-cloud messages, or file upload.

Refer to [Cloud-to-device communication guidance](#) for guidance on using desired properties, direct methods, or cloud-to-device messages.

Module twins

Module twins store module-related information that:

- Modules on the device and IoT Hub can use to synchronize module conditions and configuration.
- The solution back end can use to query and target long-running operations.

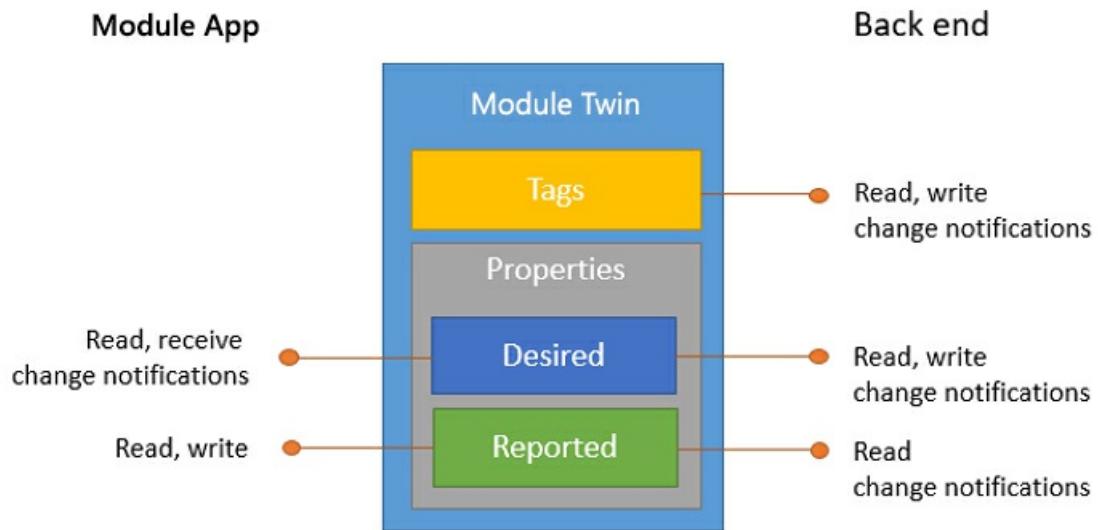
The lifecycle of a module twin is linked to the corresponding [module identity](#). Modules twins are implicitly created and deleted when a module identity is created or deleted in IoT Hub.

A module twin is a JSON document that includes:

- **Tags.** A section of the JSON document that the solution back end can read from and write to. Tags are not

visible to modules on the device. Tags are set for querying purpose.

- **Desired properties.** Used along with reported properties to synchronize module configuration or conditions. The solution back end can set desired properties, and the module app can read them. The module app can also receive notifications of changes in the desired properties.
- **Reported properties.** Used along with desired properties to synchronize module configuration or conditions. The module app can set reported properties, and the solution back end can read and query them.
- **Module identity properties.** The root of the module twin JSON document contains the read-only properties from the corresponding module identity stored in the [identity registry](#).



The following example shows a module twin JSON document:

```
{
    "deviceId": "devA",
    "moduleId": "moduleA",
    "etag": "AAAAAAAAC=",
    "status": "enabled",
    "statusReason": "provisioned",
    "statusUpdateTime": "2001-01-01T00:00:00",
    "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
        }
    }
}
```

In the root object are the module identity properties, and container objects for `tags` and both `reported` and `desired` properties. The `properties` container contains some read-only elements (`$metadata`, `$etag`, and `$version`) described in the [Module twin metadata](#) and [Optimistic concurrency](#) sections.

Reported property example

In the previous example, the module twin contains a `batteryLevel` property that is reported by the module app. This property makes it possible to query and operate on modules based on the last reported battery level. Other examples include the module app reporting module capabilities or connectivity options.

NOTE

Reported properties simplify scenarios where the solution back end is interested in the last known value of a property. Use [device-to-cloud messages](#) if the solution back end needs to process module telemetry in the form of sequences of timestamped events, such as time series.

Desired property example

In the previous example, the `telemetryConfig` module twin desired and reported properties are used by the solution back end and the module app to synchronize the telemetry configuration for this module. For example:

1. The solution back end sets the desired property with the desired configuration value. Here is the portion of the document with the desired property set:

```
...  
"desired": {  
    "telemetryConfig": {  
        "sendFrequency": "5m"  
    },  
    ...  
},  
...  
}
```

2. The module app is notified of the change immediately if connected, or at the first reconnect. The module app then reports the updated configuration (or an error condition using the `status` property). Here is the portion of the reported properties:

```
"reported": {  
    "telemetryConfig": {  
        "sendFrequency": "5m",  
        "status": "success"  
    }  
    ...  
}
```

3. The solution back end can track the results of the configuration operation across many modules, by [querying](#) module twins.

NOTE

The preceding snippets are examples, optimized for readability, of one way to encode a module configuration and its status. IoT Hub does not impose a specific schema for the module twin desired and reported properties in the module twins.

Back-end operations

The solution back end operates on the module twin using the following atomic operations, exposed through HTTPS:

- **Retrieve module twin by ID.** This operation returns the module twin document, including tags and desired and reported system properties.
- **Partially update module twin.** This operation enables the solution back end to partially update the tags or desired properties in a module twin. The partial update is expressed as a JSON document that adds or updates any property. Properties set to `null` are removed. The following example creates a new desired property with value `{"newProperty": "newValue"}`, overwrites the existing value of `existingProperty` with `"otherNewValue"`, and removes `otherOldProperty`. No other changes are made to existing desired properties or tags:

```
{
  "properties": {
    "desired": {
      "newProperty": {
        "nestedProperty": "newValue"
      },
      "existingProperty": "othernewValue",
      "otherOldProperty": null
    }
  }
}
```

- **Replace desired properties.** This operation enables the solution back end to completely overwrite all existing desired properties and substitute a new JSON document for `properties/desired`.
- **Replace tags.** This operation enables the solution back end to completely overwrite all existing tags and substitute a new JSON document for `tags`.
- **Receive twin notifications.** This operation allows the solution back end to be notified when the twin is modified. To do so, your IoT solution needs to create a route and to set the Data Source equal to `twinChangeEvents`. By default, no twin notifications are sent, that is, no such routes pre-exist. If the rate of change is too high, or for other reasons such as internal failures, the IoT Hub might send only one notification that contains all changes. Therefore, if your application needs reliable auditing and logging of all intermediate states, you should use device-to-cloud messages. The twin notification message includes properties and body.
 - Properties

NAME	VALUE
\$content-type	application/json
\$iothub-enqueuedtime	Time when the notification was sent
\$iothub-message-source	twinChangeEvents
\$content-encoding	utf-8
deviceId	ID of the device
moduleId	ID of the module
hubName	Name of IoT Hub
operationTimestamp	ISO8601 timestamp of operation
iothub-message-schema	deviceLifecycleNotification
opType	"replaceTwin" or "updateTwin"

Message system properties are prefixed with the `$` symbol.

- Body

This section includes all the twin changes in a JSON format. It uses the same format as a patch, with the difference that it can contain all twin sections: tags, properties.reported, properties.desired, and

that it contains the "\$metadata" elements. For example,

```
{  
  "properties": {  
    "desired": {  
      "$metadata": {  
        "$lastUpdated": "2016-02-30T16:24:48.789Z"  
      },  
      "$version": 1  
    },  
    "reported": {  
      "$metadata": {  
        "$lastUpdated": "2016-02-30T16:24:48.789Z"  
      },  
      "$version": 1  
    }  
  }  
}
```

All the preceding operations support [Optimistic concurrency](#) and require the **ServiceConnect** permission, as defined in the [Control Access to IoT Hub](#) article.

In addition to these operations, the solution back end can query the module twins using the SQL-like [IoT Hub query language](#).

Module operations

The module app operates on the module twin using the following atomic operations:

- **Retrieve module twin.** This operation returns the module twin document (including tags and desired and reported system properties) for the currently connected module.
- **Partially update reported properties.** This operation enables the partial update of the reported properties of the currently connected module. This operation uses the same JSON update format that the solution back end uses for a partial update of desired properties.
- **Observe desired properties.** The currently connected module can choose to be notified of updates to the desired properties when they happen. The module receives the same form of update (partial or full replacement) executed by the solution back end.

All the preceding operations require the **ModuleConnect** permission, as defined in the [Control Access to IoT Hub](#) article.

The [Azure IoT device SDKs](#) make it easy to use the preceding operations from many languages and platforms.

Tags and properties format

Tags, desired properties, and reported properties are JSON objects with the following restrictions:

- All keys in JSON objects are case-sensitive 64 bytes UTF-8 UNICODE strings. Allowed characters exclude UNICODE control characters (segments C0 and C1), and `[. , SP, and $]`.
- All values in JSON objects can be of the following JSON types: boolean, number, string, object. Arrays are not allowed. The maximum value for integers is 4503599627370495 and the minimum value for integers is -4503599627370496.
- All JSON objects in tags, desired, and reported properties can have a maximum depth of 5. For instance, the following object is valid:

```
{  
  ...  
  "tags": {  
    "one": {  
      "two": {  
        "three": {  
          "four": {  
            "five": {  
              "property": "value"  
            }  
          }  
        }  
      }  
    }  
  },  
  ...  
}
```

- All string values can be at most 512 bytes in length.

Module twin size

IoT Hub enforces an 8KB size limitation on each of the respective total values of `tags`, `properties/desired`, and `properties/reported`, excluding read-only elements.

The size is computed by counting all characters, excluding UNICODE control characters (segments C0 and C1) and spaces that are outside of string constants.

IoT Hub rejects with an error all operations that would increase the size of those documents above the limit.

Module twin metadata

IoT Hub maintains the timestamp of the last update for each JSON object in module twin desired and reported properties. The timestamps are in UTC and encoded in the [ISO8601](#) format `YYYY-MM-DDTHH:MM:SS.mmmZ`. For example:

```
{
  ...
  "properties": {
    "desired": {
      "telemetryConfig": {
        "sendFrequency": "5m"
      },
      "$metadata": {
        "telemetryConfig": {
          "sendFrequency": {
            "$lastUpdated": "2016-03-30T16:24:48.789Z"
          },
          "$lastUpdated": "2016-03-30T16:24:48.789Z"
        },
        "$lastUpdated": "2016-03-30T16:24:48.789Z"
      },
      "$version": 23
    },
    "reported": {
      "telemetryConfig": {
        "sendFrequency": "5m",
        "status": "success"
      }
      "batteryLevel": "55%",
      "$metadata": {
        "telemetryConfig": {
          "sendFrequency": "5m",
          "status": {
            "$lastUpdated": "2016-03-31T16:35:48.789Z"
          },
          "$lastUpdated": "2016-03-31T16:35:48.789Z"
        }
        "batteryLevel": {
          "$lastUpdated": "2016-04-01T16:35:48.789Z"
        },
        "$lastUpdated": "2016-04-01T16:24:48.789Z"
      },
      "$version": 123
    }
  }
  ...
}
```

This information is kept at every level (not just the leaves of the JSON structure) to preserve updates that remove object keys.

Optimistic concurrency

Tags, desired, and reported properties all support optimistic concurrency. Tags have an ETag, as per [RFC7232](#), that represents the tag's JSON representation. You can use ETags in conditional update operations from the solution back end to ensure consistency.

Module twin desired and reported properties do not have ETags, but have a `$version` value that is guaranteed to be incremental. Similarly to an ETag, the version can be used by the updating party to enforce consistency of updates. For example, a module app for a reported property or the solution back end for a desired property.

Versions are also useful when an observing agent (such as the module app observing the desired properties) must reconcile races between the result of a retrieve operation and an update notification. The section [Device reconnection flow](#) provides more information.

Next steps

To try out some of the concepts described in this article, see the following IoT Hub tutorials:

- [Get started with IoT Hub module identity and module twin using .NET back end and .NET device](#)

Understand and invoke direct methods from IoT Hub

1/28/2019 • 5 minutes to read

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.

NOTE

The features described in this article are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

Each device method targets a single device. [Schedule jobs on multiple devices](#) shows how to provide a way to invoke direct methods on multiple devices, and schedule method invocation for disconnected devices.

Anyone with **service connect** permissions on IoT Hub may invoke a method on a device.

Direct methods follow a request-response pattern and are meant for communications that require immediate confirmation of their result. For example, interactive control of the device, such as turning on a fan.

Refer to [Cloud-to-device communication guidance](#) if in doubt between using desired properties, direct methods, or cloud-to-device messages.

Method lifecycle

Direct methods are implemented on the device and may require zero or more inputs in the method payload to correctly instantiate. You invoke a direct method through a service-facing URI (`{iot hub}/twins/{device id}/methods/`). A device receives direct methods through a device-specific MQTT topic (`($iothub/methods/POST/{method name}/`) or through AMQP links (the `IoHub-methodname` and `IoHub-status` application properties).

NOTE

When you invoke a direct method on a device, property names and values can only contain US-ASCII printable alphanumeric, except any in the following set:

```
{'$', '(', ')', '<', '>', '@', ',', ';', ':', '\', "'", '/', '[', ']', '?', '=', '{', '}', SP, HT}
```

Direct methods are synchronous and either succeed or fail after the timeout period (default: 30 seconds, settable up to 300 seconds). Direct methods are useful in interactive scenarios where you want a device to act if and only if the device is online and receiving commands. For example, turning on a light from a phone. In these scenarios, you want to see an immediate success or failure so the cloud service can act on the result as soon as possible. The device may return some message body as a result of the method, but it isn't required for the method to do so. There is no guarantee on ordering or any concurrency semantics on method calls.

Direct methods are HTTPS-only from the cloud side, and MQTT or AMQP from the device side.

The payload for method requests and responses is a JSON document up to 128 KB.

Invoke a direct method from a back-end app

Now, invoke a direct method from a back-end app.

Method invocation

Direct method invocations on a device are HTTPS calls that are made up of the following items:

- The *request URI* specific to the device along with the [API version](#):

```
https://fully-qualified-iothubname.azure-devices.net/twins/{deviceId}/methods?api-version=2018-06-30
```

- The POST *method*
- *Headers* that contain the authorization, request ID, content type, and content encoding.
- A transparent JSON *body* in the following format:

```
{
  "methodName": "reboot",
  "responseTimeoutInSeconds": 200,
  "payload": {
    "input1": "someInput",
    "input2": "anotherInput"
  }
}
```

Timeout is in seconds. If timeout is not set, it defaults to 30 seconds.

Example

See below for a barebone example using `curl`.

```
curl -X POST \
  https://iothubname.azure-devices.net/twins/myfirstdevice/methods?api-version=2018-06-30 \
  -H 'Authorization: SharedAccessSignature sr=iothubname.azure-devices.net&sig=x&se=x&skn=iothubowner' \
  -H 'Content-Type: application/json' \
  -d '{
    "methodName": "reboot",
    "responseTimeoutInSeconds": 200,
    "payload": {
      "input1": "someInput",
      "input2": "anotherInput"
    }
}'
```

Response

The back-end app receives a response that is made up of the following items:

- *HTTP status code*, which is used for errors coming from the IoT Hub, including a 404 error for devices not currently connected.
- *Headers* that contain the ETag, request ID, content type, and content encoding.
- A JSON *body* in the following format:

```
{
  "status" : 201,
  "payload" : {...}
}
```

Both `status` and `body` are provided by the device and used to respond with the device's own status code and/or description.

Method invocation for IoT Edge modules

Invoking direct methods using a module ID is supported in the [IoT Service Client C# SDK](#).

For this purpose, use the `ServiceClient.InvokeDeviceMethodAsync()` method and pass in the `deviceId` and `moduleId` as parameters.

Handle a direct method on a device

Let's look at how to handle a direct method on an IoT device.

MQTT

The following section is for the MQTT protocol.

Method invocation

Devices receive direct method requests on the MQTT topic:

`$iothub/methods/POST/{method name}/?$rid={request id}`. The number of subscriptions per device is limited to 5. It is therefore recommended not to subscribe to each direct method individually. Instead consider subscribing to `$iothub/methods/POST/#` and then filter the delivered messages based on your desired method names.

The body that the device receives is in the following format:

```
{  
    "input1": "someInput",  
    "input2": "anotherInput"  
}
```

Method requests are QoS 0.

Response

The device sends responses to `$iothub/methods/res/{status}/?$rid={request id}`, where:

- The `status` property is the device-supplied status of method execution.
- The `$rid` property is the request ID from the method invocation received from IoT Hub.

The body is set by the device and can be any status.

AMQP

The following section is for the AMQP protocol.

Method invocation

The device receives direct method requests by creating a receive link on address

`amqps://{{hostname}}:5671/devices/{{deviceId}}/methods/deviceBound`.

The AMQP message arrives on the receive link that represents the method request. It contains the following sections:

- The correlation ID property, which contains a request ID that should be passed back with the corresponding method response.
- An application property named `IoTHub-methodname`, which contains the name of the method being invoked.
- The AMQP message body containing the method payload as JSON.

Response

The device creates a sending link to return the method response on address

```
amqps://{{hostname}}:5671/devices/{{deviceId}}/methods/deviceBound
```

The method's response is returned on the sending link and is structured as follows:

- The correlation ID property, which contains the request ID passed in the method's request message.
- An application property named `IoTHub-status`, which contains the user supplied method status.
- The AMQP message body containing the method response as JSON.

Additional reference material

Other reference topics in the IoT Hub developer guide include:

- [IoT Hub endpoints](#) describes the various endpoints that each IoT hub exposes for run-time and management operations.
- [Throttling and quotas](#) describes the quotas that apply and the throttling behavior to expect when you use IoT Hub.
- [Azure IoT device and service SDKs](#) lists the various language SDKs you can use when you develop both device and service apps that interact with IoT Hub.
- [IoT Hub query language for device twins, jobs, and message routing](#) describes the IoT Hub query language you can use to retrieve information from IoT Hub about your device twins and jobs.
- [IoT Hub MQTT support](#) provides more information about IoT Hub support for the MQTT protocol.

Next steps

Now you have learned how to use direct methods, you may be interested in the following IoT Hub developer guide article:

- [Schedule jobs on multiple devices](#)

If you would like to try out some of the concepts described in this article, you may be interested in the following IoT Hub tutorial:

- [Use direct methods](#)
- [Device management with Azure IoT Tools for VS Code](#)

Schedule jobs on multiple devices

2/28/2019 • 4 minutes to read

Azure IoT Hub enables a number of building blocks like [device twin properties and tags](#) and [direct methods](#). Typically, back-end apps enable device administrators and operators to update and interact with IoT devices in bulk and at a scheduled time. Jobs execute device twin updates and direct methods against a set of devices at a scheduled time. For example, an operator would use a back-end app that initiates and tracks a job to reboot a set of devices in building 43 and floor 3 at a time that would not be disruptive to the operations of the building.

NOTE

The features described in this article are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

Consider using jobs when you need to schedule and track progress any of the following activities on a set of devices:

- Update desired properties
- Update tags
- Invoke direct methods

Job lifecycle

Jobs are initiated by the solution back end and maintained by IoT Hub. You can initiate a job through a service-facing URI (`PUT https://<iot hub>/jobs/v2/<jobID>?api-version=2018-06-30`) and query for progress on an executing job through a service-facing URI (`GET https://<iot hub>/jobs/v2/<jobID>?api-version=2018-06-30`). To refresh the status of running jobs once a job is initiated, run a job query.

NOTE

When you initiate a job, property names and values can only contain US-ASCII printable alphanumeric, except any in the following set: `$ () < > @ , ; : \ " / [] ? = { }` SP HT

Jobs to execute direct methods

The following snippet shows the HTTPS 1.1 request details for executing a [direct method](#) on a set of devices using a job:

```

PUT /jobs/v2/<jobId>?api-version=2018-06-30

Authorization: <config.sharedAccessSignature>
Content-Type: application/json; charset=utf-8
Request-Id: <guid>
User-Agent: <sdk-name>/<sdk-version>

{
    "jobId": "<jobId>",
    "type": "scheduleDirectMethod",
    "cloudToDeviceMethod": {
        "methodName": "<methodName>",
        "payload": <payload>,
        "responseTimeoutInSeconds": methodTimeoutInSeconds
    },
    "queryCondition": "<queryOrDevices>", // query condition
    "startTime": <jobStartTime>,           // as an ISO-8601 date string
    "maxExecutionTimeInSeconds": <maxExecutionTimeInSeconds>
}

```

The query condition can also be on a single device ID or on a list of device IDs as shown in the following examples:

```

"queryCondition" = "deviceId = 'MyDevice1'"
"queryCondition" = "deviceId IN ['MyDevice1','MyDevice2']"
"queryCondition" = "deviceId IN ['MyDevice1']"

```

[IoT Hub Query Language](#) covers IoT Hub query language in additional detail.

Jobs to update device twin properties

The following snippet shows the HTTPS 1.1 request details for updating device twin properties using a job:

```

PUT /jobs/v2/<jobId>?api-version=2018-06-30

Authorization: <config.sharedAccessSignature>
Content-Type: application/json; charset=utf-8
Request-Id: <guid>
User-Agent: <sdk-name>/<sdk-version>

{
    "jobId": "<jobId>",
    "type": "scheduleTwinUpdate",
    "updateTwin": <patch>           // Valid JSON object
    "queryCondition": "<queryOrDevices>", // query condition
    "startTime": <jobStartTime>,       // as an ISO-8601 date string
    "maxExecutionTimeInSeconds": <maxExecutionTimeInSeconds>
}

```

Querying for progress on jobs

The following snippet shows the HTTPS 1.1 request details for querying for jobs:

```

GET /jobs/v2/query?api-version=2018-06-30[&jobType=<jobType>][&jobStatus=<jobStatus>][&pageSize=<pageSize>]
[&continuationToken=<continuationToken>]

Authorization: <config.sharedAccessSignature>
Content-Type: application/json; charset=utf-8
Request-Id: <guid>
User-Agent: <sdk-name>/<sdk-version>

```

The continuationToken is provided from the response.

You can query for the job execution status on each device using the [IoT Hub query language for device twins, jobs, and message routing](#).

Jobs Properties

The following list shows the properties and corresponding descriptions, which can be used when querying for jobs or job results.

PROPERTY	DESCRIPTION
jobId	Application provided ID for the job.
startTime	Application provided start time (ISO-8601) for the job.
endTime	IoT Hub provided date (ISO-8601) for when the job completed. Valid only after the job reaches the 'completed' state.
type	Types of jobs:
	scheduledUpdateTwin : A job used to update a set of desired properties or tags.
	scheduledDeviceMethod : A job used to invoke a device method on a set of device twins.
status	Current state of the job. Possible values for status:
	pending : Scheduled and waiting to be picked up by the job service.
	scheduled : Scheduled for a time in the future.
	running : Currently active job.
	canceled : Job has been canceled.
	failed : Job failed.
	completed : Job has completed.
deviceJobStatistics	Statistics about the job's execution.
	deviceJobStatistics properties:

PROPERTY	DESCRIPTION
	deviceJobStatistics.deviceCount : Number of devices in the job.
	deviceJobStatistics.failedCount : Number of devices where the job failed.
	deviceJobStatistics.succeededCount : Number of devices where the job succeeded.
	deviceJobStatistics.runningCount : Number of devices that are currently running the job.
	deviceJobStatistics.pendingCount : Number of devices that are pending to run the job.

Additional reference material

Other reference topics in the IoT Hub developer guide include:

- [IoT Hub endpoints](#) describes the various endpoints that each IoT hub exposes for run-time and management operations.
- [Throttling and quotas](#) describes the quotas that apply to the IoT Hub service and the throttling behavior to expect when you use the service.
- [Azure IoT device and service SDKs](#) lists the various language SDKs you can use when you develop both device and service apps that interact with IoT Hub.
- [IoT Hub query language for device twins, jobs, and message routing](#) describes the IoT Hub query language. Use this query language to retrieve information from IoT Hub about your device twins and jobs.
- [IoT Hub MQTT support](#) provides more information about IoT Hub support for the MQTT protocol.

Next steps

To try out some of the concepts described in this article, see the following IoT Hub tutorial:

- [Schedule and broadcast jobs](#)

Reference - IoT Hub endpoints

3/6/2019 • 4 minutes to read

NOTE

Some of the features mentioned in this article, like cloud-to-device messaging, device twins, and device management, are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

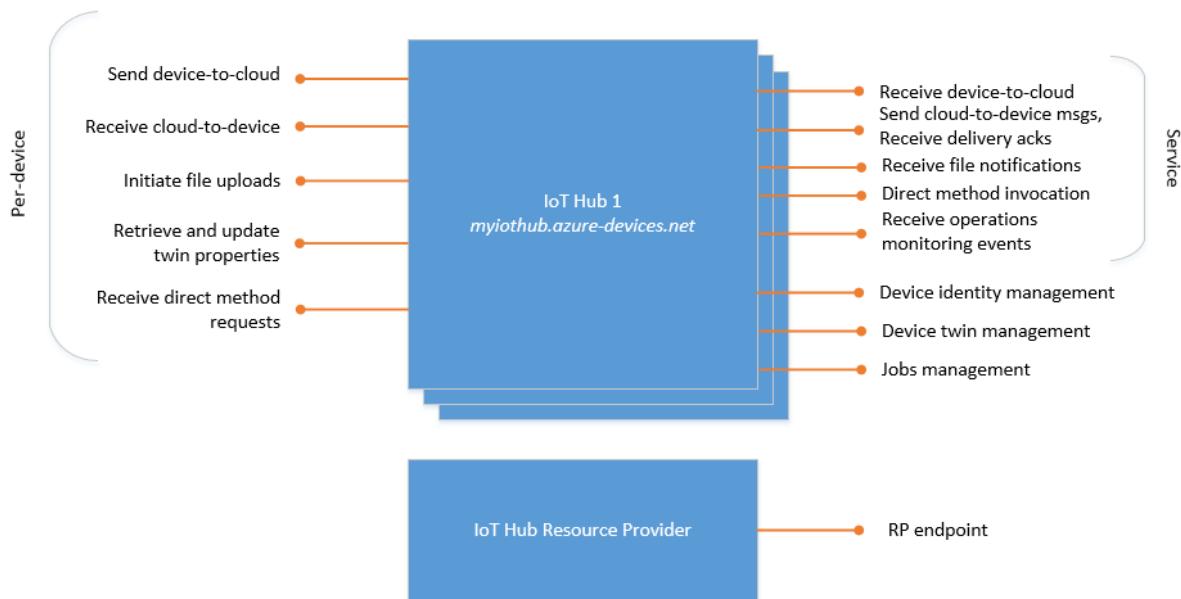
IoT Hub names

You can find the hostname of the IoT hub that hosts your endpoints in the portal on your hub's **Overview** page. By default, the DNS name of an IoT hub looks like: `{your iot hub name}.azure-devices.net`.

You can use Azure DNS to create a custom DNS name for your IoT hub. For more information, see [Use Azure DNS to provide custom domain settings for an Azure service](#).

List of built-in IoT Hub endpoints

Azure IoT Hub is a multi-tenant service that exposes its functionality to various actors. The following diagram shows the various endpoints that IoT Hub exposes.



The following list describes the endpoints:

- **Resource provider.** The IoT Hub resource provider exposes an [Azure Resource Manager](#) interface. This interface enables Azure subscription owners to create and delete IoT hubs, and to update IoT hub properties. IoT Hub properties govern [hub-level security policies](#), as opposed to device-level access control, and functional options for cloud-to-device and device-to-cloud messaging. The IoT Hub resource provider also enables you to [export device identities](#).
- **Device identity management.** Each IoT hub exposes a set of HTTPS REST endpoints to manage device identities (create, retrieve, update, and delete). [Device identities](#) are used for device

authentication and access control.

- **Device twin management.** Each IoT hub exposes a set of service-facing HTTPS REST endpoint to query and update [device twins](#) (update tags and properties).
- **Jobs management.** Each IoT hub exposes a set of service-facing HTTPS REST endpoint to query and manage [jobs](#).
- **Device endpoints.** For each device in the identity registry, IoT Hub exposes a set of endpoints:
 - *Send device-to-cloud messages.* A device uses this endpoint to [send device-to-cloud messages](#).
 - *Receive cloud-to-device messages.* A device uses this endpoint to receive targeted [cloud-to-device messages](#).
 - *Initiate file uploads.* A device uses this endpoint to receive an Azure Storage SAS URI from IoT Hub to [upload a file](#).
 - *Retrieve and update device twin properties.* A device uses this endpoint to access its [device twin](#)'s properties.
 - *Receive direct method requests.* A device uses this endpoint to listen for [direct method](#)'s requests.

These endpoints are exposed using [MQTT v3.1.1](#), HTTPS 1.1, and [AMQP 1.0](#) protocols. AMQP is also available over [WebSockets](#) on port 443.

- **Service endpoints.** Each IoT hub exposes a set of endpoints for your solution back end to communicate with your devices. With one exception, these endpoints are only exposed using the [AMQP](#) protocol. The method invocation endpoint is exposed over the HTTPS protocol.
 - *Receive device-to-cloud messages.* This endpoint is compatible with [Azure Event Hubs](#). A back-end service can use it to read the [device-to-cloud messages](#) sent by your devices. You can create custom endpoints on your IoT hub in addition to this built-in endpoint.
 - *Send cloud-to-device messages and receive delivery acknowledgments.* These endpoints enable your solution back end to send reliable [cloud-to-device messages](#), and to receive the corresponding delivery or expiration acknowledgments.
 - *Receive file notifications.* This messaging endpoint allows you to receive notifications of when your devices successfully upload a file.
 - *Direct method invocation.* This endpoint allows a back-end service to invoke a [direct method](#) on a device.
 - *Receive operations monitoring events.* This endpoint allows you to receive operations monitoring events if your IoT hub has been configured to emit them. For more information, see [IoT Hub operations monitoring](#).

The [Azure IoT SDKs](#) article describes the various ways to access these endpoints.

All IoT Hub endpoints use the [TLS](#) protocol, and no endpoint is ever exposed on unencrypted/unsecured channels.

Custom endpoints

You can link existing Azure services in your subscription to your IoT hub to act as endpoints for message routing. These endpoints act as service endpoints and are used as sinks for message routes. Devices cannot write directly to the additional endpoints. Learn more about [message routing](#).

IoT Hub currently supports the following Azure services as additional endpoints:

- Azure Storage containers
- Event Hubs
- Service Bus Queues
- Service Bus Topics

For the limits on the number of endpoints you can add, see [Quotas and throttling](#).

You can use the REST API [Get Endpoint Health](#) to get health status of the endpoints. We recommend using the [IoT Hub metrics](#) related to routing message latency to identify and debug errors when endpoint health is dead or unhealthy.

HEALTH STATUS	DESCRIPTION
healthy	The endpoint is accepting messages as expected.
unhealthy	The endpoint is not accepting messages as expected and IoT Hub is retrying to send data to this endpoint. The status of an unhealthy endpoint will be updated to healthy when IoT Hub has established an eventually consistent state of health.
unknown	IoT Hub has not established a connection with the endpoint. No messages have been delivered to or rejected from this endpoint.
dead	The endpoint is not accepting messages, after IoT Hub retried sending messages for the retrial period.

Field gateways

In an IoT solution, a *field gateway* sits between your devices and your IoT Hub endpoints. It is typically located close to your devices. Your devices communicate directly with the field gateway by using a protocol supported by the devices. The field gateway connects to an IoT Hub endpoint using a protocol that is supported by IoT Hub. A field gateway might be a dedicated hardware device or a low-power computer running custom gateway software.

You can use [Azure IoT Edge](#) to implement a field gateway. IoT Edge offers functionality such as multiplexing communications from multiple devices onto the same IoT Hub connection.

Next steps

Other reference topics in this IoT Hub developer guide include:

- [IoT Hub query language for device twins, jobs, and message routing](#)
- [Quotas and throttling](#)
- [IoT Hub MQTT support](#)

IoT Hub query language for device and module twins, jobs, and message routing

3/11/2019 • 12 minutes to read

IoT Hub provides a powerful SQL-like language to retrieve information regarding [device twins](#) and [jobs](#), and [message routing](#). This article presents:

- An introduction to the major features of the IoT Hub query language, and
- The detailed description of the language. For details on query language for message routing, see [queries in message routing](#).

NOTE

Some of the features mentioned in this article, like cloud-to-device messaging, device twins, and device management, are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

Device and module twin queries

[Device twins](#) and module twins can contain arbitrary JSON objects as both tags and properties. IoT Hub enables you to query device twins and module twins as a single JSON document containing all twin information.

Assume, for instance, that your IoT hub device twins have the following structure (module twin would be similar just with an additional moduleId):

```
{
    "deviceId": "myDeviceId",
    "etag": "AAAAAAAAAAc=",
    "status": "enabled",
    "statusUpdateTime": "2001-01-01T00:00:00",
    "connectionState": "Disconnected",
    "lastActivityTime": "2001-01-01T00:00:00",
    "cloudToDeviceMessageCount": 0,
    "authenticationType": "sas",
    "x509Thumbprint": {
        "primaryThumbprint": null,
        "secondaryThumbprint": null
    },
    "version": 2,
    "tags": {
        "location": {
            "region": "US",
            "plant": "Redmond43"
        }
    },
    "properties": {
        "desired": {
            "telemetryConfig": {
                "configId": "db00ebf5-ebeb-42be-86a1-458cccb69e57",
                "sendFrequencyInSecs": 300
            },
            "$metadata": {
                ...
            },
            "$version": 4
        },
        "reported": {
            "connectivity": {
                "type": "cellular"
            },
            "telemetryConfig": {
                "configId": "db00ebf5-ebeb-42be-86a1-458cccb69e57",
                "sendFrequencyInSecs": 300,
                "status": "Success"
            },
            "$metadata": {
                ...
            },
            "$version": 7
        }
    }
}
```

Device twin queries

IoT Hub exposes the device twins as a document collection called **devices**. For example, the following query retrieves the whole set of device twins:

```
SELECT * FROM devices
```

NOTE

Azure IoT SDKs support paging of large results.

IoT Hub allows you to retrieve device twins filtering with arbitrary conditions. For instance, to receive device twins where the **location.region** tag is set to **US** use the following query:

```
SELECT * FROM devices
WHERE tags.location.region = 'US'
```

Boolean operators and arithmetic comparisons are supported as well. For example, to retrieve device twins located in the US and configured to send telemetry less than every minute, use the following query:

```
SELECT * FROM devices
WHERE tags.location.region = 'US'
AND properties.reported.telemetryConfig.sendFrequencyInSecs >= 60
```

As a convenience, it is also possible to use array constants with the **IN** and **NIN** (not in) operators. For instance, to retrieve device twins that report WiFi or wired connectivity use the following query:

```
SELECT * FROM devices
WHERE properties.reported.connectivity IN ['wired', 'wifi']
```

It is often necessary to identify all device twins that contain a specific property. IoT Hub supports the function `is_defined()` for this purpose. For instance, to retrieve device twins that define the `connectivity` property use the following query:

```
SELECT * FROM devices
WHERE is_defined(properties.reported.connectivity)
```

Refer to the [WHERE clause](#) section for the full reference of the filtering capabilities.

Grouping and aggregations are also supported. For instance, to find the count of devices in each telemetry configuration status, use the following query:

```
SELECT properties.reported.telemetryConfig.status AS status,
       COUNT() AS numberofDevices
  FROM devices
 GROUP BY properties.reported.telemetryConfig.status
```

This grouping query would return a result similar to the following example:

```
[
  {
    "numberOfDevices": 3,
    "status": "Success"
  },
  {
    "numberOfDevices": 2,
    "status": "Pending"
  },
  {
    "numberOfDevices": 1,
    "status": "Error"
  }
]
```

In this example, three devices reported successful configuration, two are still applying the configuration, and one reported an error.

Projection queries allow developers to return only the properties they care about. For example, to retrieve the last activity time of all disconnected devices use the following query:

```
SELECT LastActivityTime FROM devices WHERE status = 'enabled'
```

Module twin queries

Querying on module twins is similar to querying on device twins, but using a different collection/namespace, i.e. instead of "from devices" you can query device.modules:

```
SELECT * FROM devices.modules
```

We don't allow join between the devices and devices.modules collections. If you want to query module twins across devices, you do it based on tags. This query will return all module twins across all devices with the scanning status:

```
SELECT * FROM devices.modules WHERE properties.reported.status = 'scanning'
```

This query will return all module twins with the scanning status, but only on the specified subset of devices:

```
SELECT * FROM devices.modules  
WHERE properties.reported.status = 'scanning'  
AND deviceId IN ['device1', 'device2']
```

C# example

The query functionality is exposed by the [C# service SDK](#) in the **RegistryManager** class.

Here is an example of a simple query:

```
var query = registryManager.CreateQuery("SELECT * FROM devices", 100);  
while (query.HasMoreResults)  
{  
    var page = await query.GetNextAsTwinAsync();  
    foreach (var twin in page)  
    {  
        // do work on twin object  
    }  
}
```

The **query** object is instantiated with a page size (up to 100). Then multiple pages are retrieved by calling the **GetNextAsTwinAsync** methods multiple times.

The query object exposes multiple **Next** values, depending on the deserialization option required by the query. For example, device twin or job objects, or plain JSON when using projections.

Node.js example

The query functionality is exposed by the [Azure IoT service SDK for Node.js](#) in the **Registry** object.

Here is an example of a simple query:

```

var query = registry.createQuery('SELECT * FROM devices', 100);
var onResults = function(err, results) {
    if (err) {
        console.error('Failed to fetch the results: ' + err.message);
    } else {
        // Do something with the results
        results.forEach(function(twin) {
            console.log(twin.deviceId);
        });

        if (query.hasMoreResults) {
            query.nextAsTwin(onResults);
        }
    }
};

query.nextAsTwin(onResults);

```

The **query** object is instantiated with a page size (up to 100). Then multiple pages are retrieved by calling the **nextAsTwin** method multiple times.

The query object exposes multiple **Next** values, depending on the deserialization option required by the query. For example, device twin or job objects, or plain JSON when using projections.

Limitations

IMPORTANT

Query results can have a few minutes of delay with respect to the latest values in device twins. If querying individual device twins by ID, use the retrieve device twin API. This API always contains the latest values and has higher throttling limits.

Currently, comparisons are supported only between primitive types (no objects), for instance

`... WHERE properties.desired.config = properties.reported.config` is supported only if those properties have primitive values.

Get started with jobs queries

[Jobs](#) provide a way to execute operations on sets of devices. Each device twin contains the information of the jobs of which it is part in a collection called **jobs**.

```
{
  "deviceId": "myDeviceId",
  "etag": "AAAAAAAAAAc=",
  "tags": {
    ...
  },
  "properties": {
    ...
  },
  "jobs": [
    {
      "deviceId": "myDeviceId",
      "jobId": "myJobId",
      "jobType": "scheduleTwinUpdate",
      "status": "completed",
      "startTimeUtc": "2016-09-29T18:18:52.7418462",
      "endTimeUtc": "2016-09-29T18:20:52.7418462",
      "createdDateTimeUtc": "2016-09-29T18:18:56.7787107Z",
      "lastUpdatedDateTimeUtc": "2016-09-29T18:18:56.8894408Z",
      "outcome": {
        "deviceMethodResponse": null
      }
    },
    ...
  ]
}
```

Currently, this collection is queryable as **devices.jobs** in the IoT Hub query language.

IMPORTANT

Currently, the jobs property is never returned when querying device twins. That is, queries that contain 'FROM devices'. The jobs property can only be accessed directly with queries using `FROM devices.jobs`.

For instance, to get all jobs (past and scheduled) that affect a single device, you can use the following query:

```
SELECT * FROM devices.jobs
WHERE devices.jobs.deviceId = 'myDeviceId'
```

Note how this query provides the device-specific status (and possibly the direct method response) of each job returned.

It is also possible to filter with arbitrary Boolean conditions on all object properties in the **devices.jobs** collection.

For instance, to retrieve all completed device twin update jobs that were created after September 2016 for a specific device, use the following query:

```
SELECT * FROM devices.jobs
WHERE devices.jobs.deviceId = 'myDeviceId'
  AND devices.jobs.jobType = 'scheduleTwinUpdate'
  AND devices.jobs.status = 'completed'
  AND devices.jobs.createdTimeUtc > '2016-09-01'
```

You can also retrieve the per-device outcomes of a single job.

```
SELECT * FROM devices.jobs  
WHERE devices.jobs.jobId = 'myJobId'
```

Limitations

Currently, queries on **devices.jobs** do not support:

- Projections, therefore only `SELECT *` is possible.
- Conditions that refer to the device twin in addition to job properties (see the preceding section).
- Performing aggregations, such as count, avg, group by.

Basics of an IoT Hub query

Every IoT Hub query consists of SELECT and FROM clauses, with optional WHERE and GROUP BY clauses. Every query is run on a collection of JSON documents, for example device twins. The FROM clause indicates the document collection to be iterated on (**devices** or **devices.jobs**). Then, the filter in the WHERE clause is applied. With aggregations, the results of this step are grouped as specified in the GROUP BY clause. For each group, a row is generated as specified in the SELECT clause.

```
SELECT <select_list>  
    FROM <from_specification>  
    [WHERE <filter_condition>]  
    [GROUP BY <group_specification>]
```

FROM clause

The **FROM <from_specification>** clause can assume only two values: **FROM devices** to query device twins, or **FROM devices.jobs** to query job per-device details.

WHERE clause

The **WHERE <filter_condition>** clause is optional. It specifies one or more conditions that the JSON documents in the FROM collection must satisfy to be included as part of the result. Any JSON document must evaluate the specified conditions to "true" to be included in the result.

The allowed conditions are described in section [Expressions and conditions](#).

SELECT clause

The **SELECT <select_list>** is mandatory and specifies what values are retrieved from the query. It specifies the JSON values to be used to generate new JSON objects. For each element of the filtered (and optionally grouped) subset of the FROM collection, the projection phase generates a new JSON object. This object is constructed with the values specified in the SELECT clause.

Following is the grammar of the SELECT clause:

```

SELECT [TOP <max number>] <projection list>

<projection_list> ::=*
  | <projection_element> AS alias [, <projection_element> AS alias]+

<projection_element> ::=
  attribute_name
  | <projection_element> '.' attribute_name
  | <aggregate>

<aggregate> ::=
  count()
  | avg(<projection_element>)
  | sum(<projection_element>)
  | min(<projection_element>)
  | max(<projection_element>)

```

Attribute_name refers to any property of the JSON document in the FROM collection. Some examples of SELECT clauses can be found in the Getting started with device twin queries section.

Currently, selection clauses different than **SELECT*** are only supported in aggregate queries on device twins.

GROUP BY clause

The **GROUP BY <group_specification>** clause is an optional step that executes after the filter specified in the WHERE clause, and before the projection specified in the SELECT. It groups documents based on the value of an attribute. These groups are used to generate aggregated values as specified in the SELECT clause.

An example of a query using GROUP BY is:

```

SELECT properties.reported.telemetryConfig.status AS status,
       COUNT() AS numberOfDevices
  FROM devices
 GROUP BY properties.reported.telemetryConfig.status

```

The formal syntax for GROUP BY is:

```

GROUP BY <group_by_element>
<group_by_element> ::=
  attribute_name
  | <group_by_element> '.' attribute_name

```

Attribute_name refers to any property of the JSON document in the FROM collection.

Currently, the GROUP BY clause is only supported when querying device twins.

IMPORTANT

The term `group` is currently treated as a special keyword in queries. In case, you use `group` as your property name, consider surrounding it with double brackets to avoid errors, e.g.,

```
SELECT * FROM devices WHERE tags.[[group]].name = 'some_value'.
```

Expressions and conditions

At a high level, an *expression*:

- Evaluates to an instance of a JSON type (such as Boolean, number, string, array, or object).
- Is defined by manipulating data coming from the device JSON document and constants using built-in operators and functions.

Conditions are expressions that evaluate to a Boolean. Any constant different than Boolean **true** is considered as **false**. This rule includes **null**, **undefined**, any object or array instance, any string, and the Boolean **false**.

The syntax for expressions is:

```

<expression> ::= 
    <constant> |
    attribute_name |
    <function_call> |
    <expression> binary_operator <expression> |
    <create_array_expression> |
    '(' <expression> ')'

<function_call> ::= 
    <function_name> '(' expression ')'

<constant> ::= 
    <undefined_constant>
    | <null_constant>
    | <number_constant>
    | <string_constant>
    | <array_constant>

<undefined_constant> ::= undefined
<null_constant> ::= null
<number_constant> ::= decimal_literal | hexadecimal_literal
<string_constant> ::= string_literal
<array_constant> ::= '[' <constant> [, <constant>]+ ']'

```

To understand what each symbol in the expressions syntax stands for, refer to the following table:

SYMBOL	DEFINITION
attribute_name	Any property of the JSON document in the FROM collection.
binary_operator	Any binary operator listed in the Operators section.
function_name	Any function listed in the Functions section.
decimal_literal	A float expressed in decimal notation.
hexadecimal_literal	A number expressed by the string '0x' followed by a string of hexadecimal digits.
string_literal	String literals are Unicode strings represented by a sequence of zero or more Unicode characters or escape sequences. String literals are enclosed in single quotes or double quotes. Allowed escapes: \', \", \\, \uXXXX for Unicode characters defined by 4 hexadecimal digits.

Operators

The following operators are supported:

FAMILY	OPERATORS
Arithmetic	+, -, *, /, %
Logical	AND, OR, NOT
Comparison	=, !=, <, >, <=, >=, <>

Functions

When querying twins and jobs the only supported function is:

FUNCTION	DESCRIPTION
IS_DEFINED(property)	Returns a Boolean indicating if the property has been assigned a value (including <code>null</code>).

In routes conditions, the following math functions are supported:

FUNCTION	DESCRIPTION
ABS(x)	Returns the absolute (positive) value of the specified numeric expression.
EXP(x)	Returns the exponential value of the specified numeric expression (e^x).
POWER(x,y)	Returns the value of the specified expression to the specified power (x^y).
SQUARE(x)	Returns the square of the specified numeric value.
CEILING(x)	Returns the smallest integer value greater than, or equal to, the specified numeric expression.
FLOOR(x)	Returns the largest integer less than or equal to the specified numeric expression.
SIGN(x)	Returns the positive (+1), zero (0), or negative (-1) sign of the specified numeric expression.
SQRT(x)	Returns the square root of the specified numeric value.

In routes conditions, the following type checking and casting functions are supported:

FUNCTION	DESCRIPTION
AS_NUMBER	Converts the input string to a number. <code>noop</code> if input is a number; <code>Undefined</code> if string does not represent a number.
IS_ARRAY	Returns a Boolean value indicating if the type of the specified expression is an array.

FUNCTION	DESCRIPTION
IS_BOOL	Returns a Boolean value indicating if the type of the specified expression is a Boolean.
IS_DEFINED	Returns a Boolean indicating if the property has been assigned a value.
IS_NULL	Returns a Boolean value indicating if the type of the specified expression is null.
IS_NUMBER	Returns a Boolean value indicating if the type of the specified expression is a number.
IS_OBJECT	Returns a Boolean value indicating if the type of the specified expression is a JSON object.
IS_PRIMITIVE	Returns a Boolean value indicating if the type of the specified expression is a primitive (string, Boolean, numeric, or <code>null</code>).
IS_STRING	Returns a Boolean value indicating if the type of the specified expression is a string.

In routes conditions, the following string functions are supported:

FUNCTION	DESCRIPTION
CONCAT(x, y, ...)	Returns a string that is the result of concatenating two or more string values.
LENGTH(x)	Returns the number of characters of the specified string expression.
LOWER(x)	Returns a string expression after converting uppercase character data to lowercase.
UPPER(x)	Returns a string expression after converting lowercase character data to uppercase.
SUBSTRING(string, start [, length])	Returns part of a string expression starting at the specified character zero-based position and continues to the specified length, or to the end of the string.
INDEX_OF(string, fragment)	Returns the starting position of the first occurrence of the second string expression within the first specified string expression, or -1 if the string is not found.
STARTS_WITH(x, y)	Returns a Boolean indicating whether the first string expression starts with the second.
ENDS_WITH(x, y)	Returns a Boolean indicating whether the first string expression ends with the second.
CONTAINS(x,y)	Returns a Boolean indicating whether the first string expression contains the second.

Next steps

Learn how to execute queries in your apps using [Azure IoT SDKs](#).

Reference - IoT Hub quotas and throttling

3/12/2019 • 5 minutes to read

Quotas and throttling

Each Azure subscription can have at most 50 IoT hubs, and at most 1 Free hub.

Each IoT hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. The message size used to calculate the daily quota is 0.5 KB for a free tier hub and 4KB for all other tiers. For more information, see [Azure IoT Hub Pricing](#).

The tier also determines the throttling limits that IoT Hub enforces on all operations.

Operation throttles

Operation throttles are rate limitations that are applied in minute ranges, and are intended to prevent abuse. IoT Hub tries to avoid returning errors whenever possible, but starts returning `429 ThrottlingException` if the throttle is violated for too long.

At any given time, you can increase quotas or throttle limits by increasing the number of provisioned units in an IoT hub.

The following table shows the enforced throttles. Values refer to an individual hub.

THROTTLE	FREE, B1, AND S1	B2 AND S2	B3 AND S3
Identity registry operations (create, retrieve, list, update, delete)	1.67/sec/unit (100/min/unit)	1.67/sec/unit (100/min/unit)	83.33/sec/unit (5000/min/unit)
New device connections (this limit applies to the rate at which <i>new connections</i> are established, not the total number of connections)	Higher of 100/sec or 12/sec/unit For example, two S1 units are $2 \times 12 = 24$ new connections/sec, but you have at least 100 new connections/sec across your units. With nine S1 units, you have 108 new connections/sec (9×12) across your units.	120 new connections/sec/unit	6000 new connections/sec/unit
Device-to-cloud sends	Higher of 100/sec or 12/sec/unit For example, two S1 units are $2 \times 12 = 24$ /sec, but you have at least 100/sec across your units. With nine S1 units, you have 108/sec (9×12) across your units.	120/sec/unit	6000/sec/unit

THROTTLE	FREE, B1, AND S1	B2 AND S2	B3 AND S3
Cloud-to-device sends ¹	1.67/sec/unit (100/min/unit)	1.67/sec/unit (100/min/unit)	83.33/sec/unit (5000/min/unit)
Cloud-to-device receives ¹ (only when device uses HTTPS)	16.67/sec/unit (1000/min/unit)	16.67/sec/unit (1000/min/unit)	833.33/sec/unit (50000/min/unit)
File upload	1.67 file upload notifications/sec/unit (100/min/unit)	1.67 file upload notifications/sec/unit (100/min/unit)	83.33 file upload notifications/sec/unit (5000/min/unit)
Direct methods ¹	160KB/sec/unit ²	480KB/sec/unit ²	24MB/sec/unit ²
Twin (device and module) reads ¹	100/sec	Higher of 100/sec or 10/sec/unit	500/sec/unit
Twin updates (device and module) ¹	50/sec	Higher of 50/sec or 5/sec/unit	250/sec/unit
Jobs operations ^{1,3} (create, update, list, delete)	1.67/sec/unit (100/min/unit)	1.67/sec/unit (100/min/unit)	83.33/sec/unit (5000/min/unit)
Jobs device operations ¹ (update twin, invoke direct method)	10/sec	Higher of 10/sec or 1/sec/unit	50/sec/unit
Configurations and edge deployments ¹ (create, update, list, delete)	0.33/sec/unit (20/min/unit)	0.33/sec/unit (20/min/unit)	0.33/sec/unit (20/min/unit)
Device stream initiation rate ⁴	5 new streams/sec	5 new streams/sec	5 new streams/sec
Maximum number of concurrently connected device streams ⁴	50	50	50
Maximum device stream data transfer ⁴ (aggregate volume per day)	300 MB	300 MB	300 MB

¹This feature is not available in the basic tier of IoT Hub. For more information, see [How to choose the right IoT Hub](#).

²Throttling meter size is 4 KB.

³You can only have one active device import/export job at a time.

⁴IoT Hub device streams are only available for S1, S2, S3, and F1 SKU's.

The *device connections* throttle governs the rate at which new device connections can be established with an IoT hub. The *device connections* throttle does not govern the maximum number of simultaneously connected devices. The *device connections* rate throttle depends on the number of units that are provisioned for the IoT hub.

For example, if you buy a single S1 unit, you get a throttle of 100 connections per second. Therefore, to connect 100,000 devices, it takes at least 1000 seconds (approximately 16 minutes). However, you can have

as many simultaneously connected devices as you have devices registered in your identity registry.

For an in-depth discussion of IoT Hub throttling behavior, see the blog post [IoT Hub throttling and you](#).

IMPORTANT

Identity registry operations are intended for run-time use in device management and provisioning scenarios. Reading or updating a large number of device identities is supported through [import and export jobs](#).

Other limits

IoT Hub enforces other operational limits:

OPERATION	LIMIT
File upload URIs	10000 SAS URIs can be out for a storage account at one time. 10 SAS URIs/device can be out at one time.
Jobs ¹	Job history is retained up to 30 days Maximum concurrent jobs is 1 (for Free) and S1, 5 (for S2), 10 (for S3).
Additional endpoints	Paid SKU hubs may have 10 additional endpoints. Free SKU hubs may have one additional endpoint.
Message routing rules	Paid SKU hubs may have 100 routing rules. Free SKU hubs may have five routing rules.
Device-to-cloud messaging	Maximum message size 256 KB
Cloud-to-device messaging ¹	Maximum message size 64 KB. Maximum pending messages for delivery is 50.
Direct method ¹	Maximum direct method payload size is 128 KB.
Automatic device configurations ¹	100 configurations per paid SKU hub. 20 configurations per free SKU hub.
Automatic Edge deployments ¹	20 modules per deployment. 100 deployments per paid SKU hub. 20 deployments per free SKU hub.
Twins ¹	Maximum size per twin section (tags, desired properties, reported properties) is 8 KB

¹This feature is not available in the basic tier of IoT Hub. For more information, see [How to choose the right IoT Hub](#).

NOTE

Currently, the maximum number of devices you can connect to a single IoT hub is 1,000,000. If you want to increase this limit, contact [Microsoft Support](#).

Latency

IoT Hub strives to provide low latency for all operations. However, due to network conditions and other unpredictable factors it cannot guarantee a maximum latency. When designing your solution, you should:

- Avoid making any assumptions about the maximum latency of any IoT Hub operation.
- Provision your IoT hub in the Azure region closest to your devices.
- Consider using Azure IoT Edge to perform latency-sensitive operations on the device or on a gateway close to the device.

Multiple IoT Hub units affect throttling as described previously, but do not provide any additional latency benefits or guarantees.

If you see unexpected increases in operation latency, contact [Microsoft Support](#).

Next steps

Other reference topics in this IoT Hub developer guide include:

- [IoT Hub endpoints](#)

Azure IoT Hub pricing information

3/11/2019 • 3 minutes to read

[Azure IoT Hub pricing](#) provides the general information on different SKUs and pricing for IoT Hub. This article contains additional details on how the various IoT Hub functionalities are metered as messages by IoT Hub.

NOTE

Some of the features mentioned in this article, like cloud-to-device messaging, device twins, and device management, are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

Charges per operation

OPERATION	BILLING INFORMATION
Identity registry operations (create, retrieve, list, update, delete)	Not charged.
Device-to-cloud messages	Successfully sent messages are charged in 4-KB chunks on ingress into IoT Hub. For example, a 6-KB message is charged 2 messages.
Cloud-to-device messages	Successfully sent messages are charged in 4-KB chunks, for example a 6-KB message is charged 2 messages.
File uploads	File transfer to Azure Storage is not metered by IoT Hub. File transfer initiation and completion messages are charged as messaged metered in 4-KB increments. For example, transferring a 10-MB file is charged as two messages in addition to the Azure Storage cost.
Direct methods	Successful method requests are charged in 4-KB chunks, and responses are charged in 4-KB chunks as additional messages. Requests to disconnected devices are charged as messages in 4-KB chunks. For example, a method with a 4-KB body that results in a response with no body from the device is charged as two messages. A method with a 6-KB body that results in a 1-KB response from the device is charged as two messages for the request plus another message for the response.
Device and module twin reads	Twin reads from the device or module and from the solution back end are charged as messages in 512-byte chunks. For example, reading a 6-KB twin is charged as 12 messages.
Device and module twin updates (tags and properties)	Twin updates from the device or module and from the solution back end are charged as messages in 512-byte chunks. For example, reading a 6-KB twin is charged as 12 messages.
Device and module twin queries	Queries are charged as messages depending on the result size in 512-byte chunks.

OPERATION	BILLING INFORMATION
Jobs operations (create, update, list, delete)	Not charged.
Jobs per-device operations	Jobs operations (such as twin updates, and methods) are charged as normal. For example, a job resulting in 1000 method calls with 1-KB requests and empty-body responses is charged 1000 messages.
Keep-alive messages	When using AMQP or MQTT protocols, messages exchanged to establish the connection and messages exchanged in the negotiation are not charged.

NOTE

All sizes are computed considering the payload size in bytes (protocol framing is ignored). For messages, which have properties and body, the size is computed in a protocol-agnostic way. For more information, see [IoT Hub message format](#).

Example #1

A device sends one 1-KB device-to-cloud message per minute to IoT Hub, which is then read by Azure Stream Analytics. The solution back end invokes a method (with a 512-byte payload) on the device every 10 minutes to trigger a specific action. The device responds to the method with a result of 200 bytes.

The device consumes:

- One message * 60 minutes * 24 hours = 1440 messages per day for the device-to-cloud messages.
- Two request plus response * 6 times per hour * 24 hours = 288 messages for the methods.

This calculation gives a total of 1728 messages per day.

Example #2

A device sends one 100-KB device-to-cloud message every hour. It also updates its device twin with 1-KB payloads every four hours. The solution back end, once per day, reads the 14-KB device twin and updates it with 512-byte payloads to change configurations.

The device consumes:

- 25 (100 KB / 4 KB) messages * 24 hours for device-to-cloud messages.
- Two messages (1 KB / 0.5 KB) * six times per day for device twin updates.

This calculation gives a total of 612 messages per day.

The solution back end consumes 28 messages (14 KB / 0.5 KB) to read the device twin, plus one message to update it, for a total of 29 messages.

In total, the device and the solution back end consume 641 messages per day.

Understand and use Azure IoT Hub SDKs

2/28/2019 • 4 minutes to read

There are two categories of software development kits (SDKs) for working with IoT Hub:

- **IoT Hub Device SDKs** enable you to build apps that run on your IoT devices using device client or module client. These apps send telemetry to your IoT hub, and optionally receive messages, job, method, or twin updates from your IoT hub. You can also use module client to author [modules](#) for [Azure IoT Edge runtime](#).
- **IoT Hub Service SDKs** enable you to build backend applications to manage your IoT hub, and optionally send messages, schedule jobs, invoke direct methods, or send desired property updates to your IoT devices or modules.

In addition, we also provide a set of SDKs for working with the [Device Provisioning Service](#).

- **Provisioning Device SDKs** enable you to build apps that run on your IoT devices to communicate with the Device Provisioning Service.
- **Provisioning Service SDKs** enable you to build backend applications to manage your enrollments in the Device Provisioning Service.

Learn about the [benefits of developing using Azure IoT SDKs](#).

NOTE

Some of the features mentioned in this article, like cloud-to-device messaging, device twins, and device management, are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

OS platform and hardware compatibility

Supported platforms for the SDKs can be found in [Azure IoT SDKs Platform Support](#).

For more information about SDK compatibility with specific hardware devices, see the [Azure Certified for IoT device catalog](#) or individual repository.

Azure IoT Hub Device SDKs

The Microsoft Azure IoT device SDKs contain code that facilitates building applications that connect to and are managed by Azure IoT Hub services.

Azure IoT Hub device SDK for .NET:

- Download from [Nuget](#). The namespace is Microsoft.Azure.Devices.Clients, which contains IoT Hub Device Clients (DeviceClient, ModuleClient).
- [Source code](#)
- [API reference](#)
- [Module reference](#)

Azure IoT Hub device SDK for C (ANSI C - C99):

- Install from [apt-get](#), [MBED](#), [Arduino IDE](#) or [iOS](#)
- [Source code](#)

- [Compile the C Device SDK](#)
- [API reference](#)
- [Module reference](#)
- [Porting the C SDK to other platforms](#)
- [Developer documentation](#) for information on cross-compiling, getting started on different platforms, etc.

Azure IoT Hub device SDK for Java:

- Add to [Maven project](#)
- [Source code](#)
- [API reference](#)
- [Module reference](#)

Azure IoT Hub device SDK for Node.js:

- Install from [npm](#)
- [Source code](#)
- [API reference](#)
- [Module reference](#)

Azure IoT Hub device SDK for Python:

- Install from [pip](#)
- [Source code](#)
- API reference: see [C API reference](#)

Azure IoT Hub device SDK for iOS:

- Install from [CocoaPod](#)
- [Samples](#)
- API reference: see [C API reference](#)

Azure IoT Hub Service SDKs

The Azure IoT service SDKs contain code to facilitate building applications that interact directly with IoT Hub to manage devices and security.

Azure IoT Hub service SDK for .NET:

- Download from [Nuget](#). The namespace is Microsoft.Azure.Devices, which contains IoT Hub Service Clients (RegistryManager, ServiceClients).
- [Source code](#)
- [API reference](#)

Azure IoT Hub service SDK for Java:

- Add to [Maven project](#)
- [Source code](#)
- [API reference](#)

Azure IoT Hub service SDK for Node.js:

- Download from [npm](#)
- [Source code](#)
- [API reference](#)

Azure IoT Hub service SDK for Python:

- Download from [pip](#)
- [Source code](#)

Azure IoT Hub service SDK for C:

- Download from [apt-get, MBED, Arduino IDE, or NuGet](#)
- [Source code](#)

Azure IoT Hub service SDK for iOS:

- Install from [CocoaPod](#)
- [Samples](#)

NOTE

See the readme files in the GitHub repositories for information about using language and platform-specific package managers to install binaries and dependencies on your development machine.

Microsoft Azure Provisioning SDKs

The **Microsoft Azure Provisioning SDKs** enable you to provision devices to your IoT Hub using the [Device Provisioning Service](#).

Azure Provisioning device and service SDKs for C#:

- Download from [Device SDK](#) and [Service SDK](#) from NuGet.
- [Source code](#)
- [API reference](#)

Azure Provisioning device and service SDKs for C:

- Install from [apt-get, MBED, Arduino IDE or iOS](#)
- [Source code](#)
- [API reference](#)

Azure Provisioning device and service SDKs for Java:

- Add to [Maven](#) project
- [Source code](#)
- [API reference](#)

Azure Provisioning device and service SDKs for Nodejs:

- [Source code](#)
- [API reference](#)
- Download [Device SDK](#) and [Service SDK](#) from npm

Azure Provisioning device and service SDKs for Python:

- [Source code](#)
- Download [Device SDK](#) and [Service SDK](#) from pip

Next steps

Azure IoT SDKs also provide a set of tools to help with development:

- [iothub-diagnostics](#): a cross-platform command line tool to help diagnose issues related to connection with IoT Hub.
- [device-explorer](#): a Windows desktop application to connect to your IoT Hub.

Relevant docs related to development using the Azure IoT SDKs:

- Learn about [how to manage connectivity and reliable messaging](#) using the IoT Hub SDKs.
- Learn about how to [develop for mobile platforms](#) such as iOS and Android.
- [Azure IoT SDK platform support](#)

Other reference topics in this IoT Hub developer guide include:

- [IoT Hub endpoints](#)
- [IoT Hub query language for device twins, jobs, and message routing](#)
- [Quotas and throttling](#)
- [IoT Hub MQTT support](#)
- [IoT Hub REST API reference](#)

Communicate with your IoT hub using the MQTT protocol

3/6/2019 • 13 minutes to read

IoT Hub enables devices to communicate with the IoT Hub device endpoints using:

- [MQTT v3.1.1](#) on port 8883
- MQTT v3.1.1 over WebSocket on port 443.

IoT Hub is not a full-featured MQTT broker and does not support all the behaviors specified in the MQTT v3.1.1 standard. This article describes how devices can use supported MQTT behaviors to communicate with IoT Hub.

NOTE

Some of the features mentioned in this article, like cloud-to-device messaging, device twins, and device management, are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

All device communication with IoT Hub must be secured using TLS/SSL. Therefore, IoT Hub doesn't support non-secure connections over port 1883.

Connecting to IoT Hub

A device can use the MQTT protocol to connect to an IoT hub using:

- Either the libraries in the [Azure IoT SDKs](#).
- Or the MQTT protocol directly.

Using the device SDKs

[Device SDKs](#) that support the MQTT protocol are available for Java, Node.js, C, C#, and Python. The device SDKs use the standard IoT Hub connection string to establish a connection to an IoT hub. To use the MQTT protocol, the client protocol parameter must be set to **MQTT**. By default, the device SDKs connect to an IoT Hub with the **CleanSession** flag set to **0** and use **QoS 1** for message exchange with the IoT hub.

When a device is connected to an IoT hub, the device SDKs provide methods that enable the device to exchange messages with an IoT hub.

The following table contains links to code samples for each supported language and specifies the parameter to use to establish a connection to IoT Hub using the MQTT protocol.

LANGUAGE	PROTOCOL PARAMETER
Node.js	azure-iot-device-mqtt
Java	IoTClientProtocol.MQTT
C	MQTT_Protocol
C#	TransportType.Mqtt

LANGUAGE	PROTOCOL PARAMETER
Python	IoTHubTransportProvider.MQTT

Migrating a device app from AMQP to MQTT

If you are using the [device SDKs](#), switching from using AMQP to MQTT requires changing the protocol parameter in the client initialization as stated previously.

When doing so, make sure to check the following items:

- AMQP returns errors for many conditions, while MQTT terminates the connection. As a result your exception handling logic might require some changes.
- MQTT does not support the *reject* operations when receiving [cloud-to-device messages](#). If your back-end app needs to receive a response from the device app, consider using [direct methods](#).

Using the MQTT protocol directly (as a device)

If a device cannot use the device SDKs, it can still connect to the public device endpoints using the MQTT protocol on port 8883. In the **CONNECT** packet the device should use the following values:

- For the **ClientId** field, use the **deviceId**.
- For the **Username** field, use `{iothubhostname}/{device_id}/?api-version=2018-06-30`, where `{iothubhostname}` is the full CName of the IoT hub.

For example, if the name of your IoT hub is **contoso.azure-devices.net** and if the name of your device is **MyDevice01**, the full **Username** field should contain:

```
contoso.azure-devices.net/MyDevice01/?api-version=2018-06-30
```

- For the **Password** field, use a SAS token. The format of the SAS token is the same as for both the HTTPS and AMQP protocols:

```
SharedAccessSignature sig={signature-string}&se={expiry}&sr={URL-encoded-resourceURI}
```

NOTE

If you use X.509 certificate authentication, SAS token passwords are not required. For more information, see [Set up X.509 security in your Azure IoT Hub](#)

For more information about how to generate SAS tokens, see the device section of [Using IoT Hub security tokens](#).

When testing, you can also use the cross-platform [Azure IoT Tools for Visual Studio Code](#) or the [Device Explorer](#) tool to quickly generate a SAS token that you can copy and paste into your own code:

For Azure IoT Tools:

1. Expand the **AZURE IOT HUB DEVICES** tab in the bottom left corner of Visual Studio Code.
2. Right-click your device and select **Generate SAS Token for Device**.
3. Set **expiration time** and press 'Enter'.
4. The SAS token is created and copied to clipboard.

For Device Explorer:

1. Go to the **Management** tab in **Device Explorer**.

2. Click **SAS Token** (top right).
3. On **SASTokenForm**, select your device in the **DeviceID** drop down. Set your **TTL**.
4. Click **Generate** to create your token.

The SAS token that's generated has the following structure:

```
HostName={your hub name}.azure-devices.net;DeviceId=javadevice;SharedAccessSignature=SharedAccessSignature sr={your hub name}.azure-devices.net%2Fdevices%2FMyDevice01%2Fapi-version%3D2016-11-14&sig=vSgHBMUG.....Ntg%3d&se=1456481802
```

The part of this token to use as the **Password** field to connect using MQTT is:

```
SharedAccessSignature sr={your hub name}.azure-devices.net%2Fdevices%2FMyDevice01%2Fapi-version%3D2016-11-14&sig=vSgHBMUG.....Ntg%3d&se=1456481802
```

For MQTT connect and disconnect packets, IoT Hub issues an event on the **Operations Monitoring** channel. This event has additional information that can help you to troubleshoot connectivity issues.

The device app can specify a **Will** message in the **CONNECT** packet. The device app should use

`devices/{device_id}/messages/events/` or `devices/{device_id}/messages/events/{property_bag}` as the **Will** topic name to define **Will** messages to be forwarded as a telemetry message. In this case, if the network connection is closed, but a **DISCONNECT** packet was not previously received from the device, then IoT Hub sends the **Will** message supplied in the **CONNECT** packet to the telemetry channel. The telemetry channel can be either the default **Events** endpoint or a custom endpoint defined by IoT Hub routing. The message has the **iothub-MessageType** property with a value of **Will** assigned to it.

Using the MQTT protocol directly (as a module)

Connecting to IoT Hub over MQTT using a module identity is similar to the device (described [above](#)) but you need to use the following:

- Set the client id to `{device_id}/{module_id}`.
- If authenticating with username and password, set the username to `<hubname>.azure-devices.net/{device_id}/{module_id}/?api-version=2018-06-30` and use the SAS token associated with the module identity as your password.
- Use `devices/{device_id}/modules/{module_id}/messages/events/` as topic for publishing telemetry.
- Use `devices/{device_id}/modules/{module_id}/messages/events/` as WILL topic.
- The twin GET and PATCH topics are identical for modules and devices.
- The twin status topic is identical for modules and devices.

TLS/SSL configuration

To use the MQTT protocol directly, your client *must* connect over TLS/SSL. Attempts to skip this step fail with connection errors.

In order to establish a TLS connection, you may need to download and reference the DigiCert Baltimore Root Certificate. This certificate is the one that Azure uses to secure the connection. You can find this certificate in the [Azure-iot-sdk-c](#) repository. More information about these certificates can be found on [Digicert's website](#).

An example of how to implement this using the Python version of the [Paho MQTT library](#) by the Eclipse Foundation might look like the following.

First, install the Paho library from your command-line environment:

```
pip install paho-mqtt
```

Then, implement the client in a Python script. Replace the placeholders as follows:

- <local path to digicert.cer> is the path to a local file that contains the DigiCert Baltimore Root certificate. You can create this file by copying the certificate information from `certs.c` in the Azure IoT SDK for C. Include the lines -----BEGIN CERTIFICATE----- and -----END CERTIFICATE-----, remove the " marks at the beginning and end of every line, and remove the \r\n characters at the end of every line.
- <device id from device registry> is the ID of a device you added to your IoT hub.
- <generated SAS token> is a SAS token for the device created as described previously in this article.
- <iot hub name> the name of your IoT hub.

```

from paho.mqtt import client as mqtt
import ssl

path_to_root_cert = "<local path to digicert.cer>"
device_id = "<device id from device registry>"
sas_token = "<generated SAS token>"
iot_hub_name = "<iot hub name>

def on_connect(client, userdata, flags, rc):
    print ("Device connected with result code: " + str(rc))
def on_disconnect(client, userdata, rc):
    print ("Device disconnected with result code: " + str(rc))
def on_publish(client, userdata, mid):
    print ("Device sent message")

client = mqtt.Client(client_id=device_id, protocol=mqtt.MQTTv311)

client.on_connect = on_connect
client.on_disconnect = on_disconnect
client.on_publish = on_publish

client.username_pw_set(username=iot_hub_name+".azure-devices.net/" + device_id, password=sas_token)

client.tls_set(ca_certs=path_to_root_cert, certfile=None, keyfile=None, cert_reqs=ssl.CERT_REQUIRED,
tls_version=ssl.PROTOCOL_TLSv1, ciphers=None)
client.tls_insecure_set(False)

client.connect(iot_hub_name+".azure-devices.net", port=8883)

client.publish("devices/" + device_id + "/messages/events/", "{id=123}", qos=1)
client.loop_forever()

```

Sending device-to-cloud messages

After making a successful connection, a device can send messages to IoT Hub using

`devices/{device_id}/messages/events/` or `devices/{device_id}/messages/events/{property_bag}` as a **Topic**

Name. The `{property_bag}` element enables the device to send messages with additional properties in a url-encoded format. For example:

```
RFC 2396-encoded(<PropertyName1>)=RFC 2396-encoded(<PropertyValue1>)&RFC 2396-encoded(<PropertyName2>)=RFC
2396-encoded(<PropertyValue2>)...
```

NOTE

This `{property_bag}` element uses the same encoding as for query strings in the HTTPS protocol.

The following is a list of IoT Hub implementation-specific behaviors:

- IoT Hub does not support QoS 2 messages. If a device app publishes a message with **QoS 2**, IoT Hub closes the network connection.

- IoT Hub does not persist Retain messages. If a device sends a message with the **REtain** flag set to 1, IoT Hub adds the **x-opt-retain** application property to the message. In this case, instead of persisting the retain message, IoT Hub passes it to the backend app.
- IoT Hub only supports one active MQTT connection per device. Any new MQTT connection on behalf of the same device ID causes IoT Hub to drop the existing connection.

For more information, see [Messaging developer's guide](#).

Receiving cloud-to-device messages

To receive messages from IoT Hub, a device should subscribe using `devices/{device_id}/messages/devicebound/#` as a **Topic Filter**. The multi-level wildcard `#` in the Topic Filter is used only to allow the device to receive additional properties in the topic name. IoT Hub does not allow the usage of the `#` or `?` wildcards for filtering of subtopics. Since IoT Hub is not a general-purpose pub-sub messaging broker, it only supports the documented topic names and topic filters.

The device does not receive any messages from IoT Hub, until it has successfully subscribed to its device-specific endpoint, represented by the `devices/{device_id}/messages/devicebound/#` topic filter. After a subscription has been established, the device receives cloud-to-device messages that were sent to it after the time of the subscription. If the device connects with **CleanSession** flag set to **0**, the subscription is persisted across different sessions. In this case, the next time the device connects with **CleanSession 0** it receives any outstanding messages sent to it while disconnected. If the device uses **CleanSession** flag set to **1** though, it does not receive any messages from IoT Hub until it subscribes to its device-endpoint.

IoT Hub delivers messages with the **Topic Name** `devices/{device_id}/messages/devicebound/`, or `devices/{device_id}/messages/devicebound/{property_bag}` when there are message properties. `{property_bag}` contains url-encoded key/value pairs of message properties. Only application properties and user-settable system properties (such as **messageId** or **correlationId**) are included in the property bag. System property names have the prefix `$`, application properties use the original property name with no prefix.

When a device app subscribes to a topic with **QoS 2**, IoT Hub grants maximum QoS level 1 in the **SUBACK** packet. After that, IoT Hub delivers messages to the device using QoS 1.

Retrieving a device twin's properties

First, a device subscribes to `$iothub/twin/res/#`, to receive the operation's responses. Then, it sends an empty message to topic `$iothub/twin/GET/?$rid={request id}`, with a populated value for **request ID**. The service then sends a response message containing the device twin data on topic `$iothub/twin/res/{status}/?$rid={request id}`, using the same **request ID** as the request.

Request ID can be any valid value for a message property value, as per [IoT Hub messaging developer's guide](#), and status is validated as an integer.

The response body contains the properties section of the device twin, as shown in the following response example:

```
{
  "desired": {
    "telemetrySendFrequency": "5m",
    "$version": 12
  },
  "reported": {
    "telemetrySendFrequency": "5m",
    "batteryLevel": 55,
    "$version": 123
  }
}
```

The possible status codes are:

STATUS	DESCRIPTION
204	Success (no content is returned)
429	Too many requests (throttled), as per IoT Hub throttling
5**	Server errors

For more information, see [Device twins developer's guide](#).

Update device twin's reported properties

To update reported properties, the device issues a request to IoT Hub via a publication over a designated MQTT topic. After processing the request, IoT Hub responds the success or failure status of the update operation via a publication to another topic. This topic can be subscribed by the device in order to notify it about the result of its twin update request. To implement this type of request/response interaction in MQTT, we leverage the notion of request id (`$rid`) provided initially by the device in its update request. This request id is also included in the response from IoT Hub to allow the device to correlate the response to its particular earlier request.

The following sequence describes how a device updates the reported properties in the device twin in IoT Hub:

1. A device must first subscribe to the `$iothub/twin/res/#` topic to receive the operation's responses from IoT Hub.
2. A device sends a message that contains the device twin update to the `$iothub/twin/PATCH/properties/reported/?$rid={request id}` topic. This message includes a **request ID** value.
3. The service then sends a response message that contains the new ETag value for the reported properties collection on topic `$iothub/twin/res/{status}/?$rid={request id}`. This response message uses the same **request ID** as the request.

The request message body contains a JSON document, that contains new values for reported properties. Each member in the JSON document updates or add the corresponding member in the device twin's document. A member set to `null`, deletes the member from the containing object. For example:

```
{  
    "telemetrySendFrequency": "35m",  
    "batteryLevel": 60  
}
```

The possible status codes are:

STATUS	DESCRIPTION
200	Success
400	Bad Request. Malformed JSON
429	Too many requests (throttled), as per IoT Hub throttling
5**	Server errors

The python code snippet below, demonstrates the twin reported properties update process over MQTT (using

Paho MQTT client):

```
from paho.mqtt import client as mqtt

# authenticate the client with IoT Hub (not shown here)

client.subscribe("$iothub/twin/res/#")
rid = "1"
twin_reported_property_patch = "{\"firmware_version\": \"v1.1\"}"
client.publish("$iothub/twin/PATCH/properties/reported/?$rid=" + rid, twin_reported_property_patch, qos=0)
```

Upon success of twin reported properties update operation above, the publication message from IoT Hub will have the following topic: `$iothub/twin/res/204/?$rid=1&$version=6`, where `204` is the status code indicating success, `$rid=1` corresponds to the request ID provided by the device in the code, and `$version` corresponds to the version of reported properties section of device twins after the update.

For more information, see [Device twins developer's guide](#).

Receiving desired properties update notifications

When a device is connected, IoT Hub sends notifications to the topic

`$iothub/twin/PATCH/properties/desired/?$version={new version}`, which contain the content of the update performed by the solution back end. For example:

```
{
    "telemetrySendFrequency": "5m",
    "route": null,
    "$version": 8
}
```

As for property updates, `null` values means that the JSON object member is being deleted. Also, note that `$version` indicates the new version of the desired properties section of the twin.

IMPORTANT

IoT Hub generates change notifications only when devices are connected. Make sure to implement the [device reconnection flow](#) to keep the desired properties synchronized between IoT Hub and the device app.

For more information, see [Device twins developer's guide](#).

Respond to a direct method

First, a device has to subscribe to `$iothub/methods/POST/#`. IoT Hub sends method requests to the topic `$iothub/methods/POST/{method name}/?$rid={request id}`, with either a valid JSON or an empty body.

To respond, the device sends a message with a valid JSON or empty body to the topic `$iothub/methods/res/{status}/?$rid={request id}`. In this message, the **request ID** must match the one in the request message, and **status** must be an integer.

For more information, see [Direct method developer's guide](#).

Additional considerations

As a final consideration, if you need to customize the MQTT protocol behavior on the cloud side, you should review the [Azure IoT protocol gateway](#). This software enables you to deploy a high-performance custom protocol gateway that interfaces directly with IoT Hub. The Azure IoT protocol gateway enables you to customize the device protocol to accommodate brownfield MQTT deployments or other custom protocols. This approach does require, however, that you run and operate a custom protocol gateway.

Next steps

To learn more about the MQTT protocol, see the [MQTT documentation](#).

To learn more about planning your IoT Hub deployment, see:

- [Azure Certified for IoT device catalog](#)
- [Support additional protocols](#)
- [Compare with Event Hubs](#)
- [Scaling, HA, and DR](#)

To further explore the capabilities of IoT Hub, see:

- [IoT Hub developer guide](#)
- [Deploying AI to edge devices with Azure IoT Edge](#)

Glossary of IoT Hub terms

3/6/2019 • 20 minutes to read

This article lists some of the common terms used in the IoT Hub articles.

Advanced Message Queueing Protocol

[Advanced Message Queueing Protocol \(AMQP\)](#) is one of the messaging protocols that [IoT Hub](#) supports for communicating with devices. For more information about the messaging protocols that IoT Hub supports, see [Send and receive messages with IoT Hub](#).

Automatic Device Management

Automatic Device Management in Azure IoT Hub automates many of the repetitive and complex tasks of managing large device fleets over the entirety of their lifecycles. With Automatic Device Management, you can target a set of devices based on their properties, define a desired configuration, and let IoT Hub update devices whenever they come into scope. Consists of [automatic device configurations](#) and [IoT Edge automatic deployments](#).

Automatic device configuration

Your solution back end can use [automatic device configurations](#) to assign desired properties to a set of [device twins](#) and report status using system metrics and custom metrics.

Azure classic CLI

The [Azure classic CLI](#) is a cross-platform, open-source, shell-based, command tool for creating and managing resources in Microsoft Azure. This version of the CLI should be used for classic deployments only.

Azure CLI

The [Azure CLI](#) is a cross-platform, open-source, shell-based, command tool for creating and managing resources in Microsoft Azure.

Azure IoT device SDKs

There are *device SDKs* available for multiple languages that enable you to create [device apps](#) that interact with an IoT hub. The IoT Hub tutorials show you how to use these device SDKs. You can find the source code and further information about the device SDKs in this GitHub [repository](#).

Azure IoT service SDKs

There are *service SDKs* available for multiple languages that enable you to create [back-end apps](#) that interact with an IoT hub. The IoT Hub tutorials show you how to use these service SDKs. You can find the source code and further information about the service SDKs in this GitHub [repository](#).

Azure IoT Tools

The [Azure IoT Tools](#) is a cross-platform, open-source Visual Studio Code extension that helps you manage Azure IoT Hub and devices in VS Code. With Azure IoT Tools, IoT developers could develop IoT project in VS Code with ease.

Azure portal

The [Microsoft Azure portal](#) is a central place where you can provision and manage your Azure resources. It organizes its content using *blades*.

Azure PowerShell

[Azure PowerShell](#) is a collection of cmdlets you can use to manage Azure with Windows PowerShell. You can use the cmdlets to create, test, deploy, and manage solutions and services delivered through the Azure platform.

Azure Resource Manager

[Azure Resource Manager](#) enables you to work with the resources in your solution as a group. You can deploy, update, or delete the resources for your solution in a single, coordinated operation.

Azure Service Bus

[Service Bus](#) provides cloud-enabled communication with enterprise messaging and relayed communication that helps you connect on-premises solutions with the cloud. Some IoT Hub tutorials make use Service Bus [queues](#).

Azure Storage

[Azure Storage](#) is a cloud storage solution. It includes the Blob Storage service that you can use to store unstructured object data. Some IoT Hub tutorials use blob storage.

Back-end app

In the context of [IoT Hub](#), a back-end app is an app that connects to one of the service-facing endpoints on an IoT hub. For example, a back-end app might retrieve [device-to-cloud](#) messages or manage the [identity registry](#). Typically, a back-end app runs in the cloud, but in many of the tutorials the back-end apps are console apps running on your local development machine.

Built-in endpoints

Every IoT hub includes a built-in [endpoint](#) that is Event Hub-compatible. You can use any mechanism that works with Event Hubs to read device-to-cloud messages from this endpoint.

Cloud gateway

A cloud gateway enables connectivity for devices that cannot connect directly to [IoT Hub](#). A cloud gateway is hosted in the cloud in contrast to a [field gateway](#) that runs local to your devices. A typical use case for a cloud gateway is to implement protocol translation for your devices.

Cloud-to-device

Refers to messages sent from an IoT hub to a connected device. Often, these messages are commands that instruct the device to take an action. For more information, see [Send and receive messages with IoT Hub](#).

Configuration

In the context of [automatic device configuration](#), a configuration within IoT Hub defines the desired configuration for a set of devices twins and provides a set of metrics to report status and progress.

Connection string

You use connection strings in your app code to encapsulate the information required to connect to an endpoint. A connection string typically includes the address of the endpoint and security information, but connection string formats vary across services. There are two types of connection string associated with the IoT Hub service:

- *Device connection strings* enable devices to connect to the device-facing endpoints on an IoT hub.
- *IoT Hub connection strings* enable back-end apps to connect to the service-facing endpoints on an IoT hub.

Custom endpoints

You can create custom [endpoints](#) on an IoT hub to deliver messages dispatched by a [routing rule](#). Custom endpoints connect directly to an Event hub, a Service Bus queue, or a Service Bus topic.

Custom gateway

A gateway enables connectivity for devices that cannot connect directly to [IoT Hub](#). You can use Azure IoT Edge to build custom gateways that implement custom logic to handle messages, custom protocol conversions, and other processing on the edge.

Data-point message

A data-point message is a [device-to-cloud](#) message that contains [telemetry](#) data such as wind speed or temperature.

Desired configuration

In the context of a [device twin](#), desired configuration refers to the complete set of properties and metadata in the device twin that should be synchronized with the device.

Desired properties

In the context of a [device twin](#), desired properties is a subsection of the device twin that is used with [reported properties](#) to synchronize device configuration or condition. Desired properties can only be set by a [back-end app](#) and are observed by the [device app](#).

Device-to-cloud

Refers to messages sent from a connected device to [IoT Hub](#). These messages may be [data-point](#) or [interactive](#) messages. For more information, see [Send and receive messages with IoT Hub](#).

Device

In the context of IoT, a device is typically a small-scale, standalone computing device that may collect data or control other devices. For example, a device might be an environmental monitoring device, or a controller for the watering and ventilation systems in a greenhouse. The [device catalog](#) provides a list of hardware devices certified to work with [IoT Hub](#).

Device app

A device app runs on your [device](#) and handles the communication with your [IoT hub](#). Typically, you use one of the [Azure IoT device SDKs](#) when you implement a device app. In many of the IoT tutorials, you use a [simulated device](#) for convenience.

Device condition

Refers to device state information, such as the connectivity method currently in use, as reported by a [device app](#). [Device apps](#) can also report their capabilities. You can query for condition and capability information using device twins.

Device data

Device data refers to the per-device data stored in the IoT Hub [identity registry](#). It is possible to import and export this data.

Device explorer

The [device explorer](#) is a tool that runs on Windows and enables you to manage your devices in the [identity registry](#). The tool can also send and receive messages to your devices.

Device identity

The device identity is the unique identifier assigned to every device registered in the [identity registry](#).

Device management

Device management encompasses the full lifecycle associated with managing the devices in your IoT solution including planning, provisioning, configuring, monitoring, and retiring.

Device management patterns

[IoT hub](#) enables common device management patterns including rebooting, performing factory resets, and performing firmware updates on your devices.

Device REST API

You can use the [Device REST API](#) from a device to send device-to-cloud messages to an IoT hub, and receive [cloud-to-device](#) messages from an IoT hub. Typically, you should use one of the higher-level [device SDKs](#) as shown in the IoT Hub tutorials.

Device provisioning

Device provisioning is the process of adding the initial [device data](#) to the stores in your solution. To enable a new device to connect to your hub, you must add a device ID and keys to the IoT Hub [identity registry](#). As part of the provisioning process, you might need to initialize device-specific data in other solution stores.

Device twin

A [device twin](#) is JSON document that stores device state information such as metadata, configurations, and conditions. [IoT Hub](#) persists a device twin for each device that you provision in your IoT hub. Device twins enable you to synchronize [device conditions](#) and configurations between the device and the solution back end. You can query device twins to locate specific devices and query the status of long-running operations.

Direct method

A [direct method](#) is a way for you to trigger a method to execute on a device by invoking an API on your IoT hub.

Endpoint

An IoT hub exposes multiple [endpoints](#) that enable your apps to connect to the IoT hub. There are device-facing endpoints that enable devices to perform operations such as sending [device-to-cloud](#) messages and receiving [cloud-to-device](#) messages. There are service-facing management endpoints that enable [back-end apps](#) to perform operations such as [device identity](#) management and device twin management. There are service-facing [built-in endpoints](#) for reading device-to-cloud messages. You can create [custom endpoints](#) to receive device-to-cloud messages dispatched by a [routing rule](#).

Event Hubs service

[Event Hubs](#) is a highly scalable data ingress service that can ingest millions of events per second. The service enables you to process and analyze the massive amounts of data produced by your connected devices and applications. For a comparison with the IoT Hub service, see [Comparison of Azure IoT Hub and Azure Event Hubs](#).

Event Hub-compatible endpoint

To read [device-to-cloud](#) messages sent to your IoT hub, you can connect to an endpoint on your hub and use any Event Hub-compatible method to read those messages. Event Hub-compatible methods include using the [Event Hubs SDKs](#) and [Azure Stream Analytics](#).

Field gateway

A field gateway enables connectivity for devices that cannot connect directly to [IoT Hub](#) and is typically deployed locally with your devices. For more information, see [What is Azure IoT Hub?](#)

Free account

You can create a [free Azure account](#) to complete the IoT Hub tutorials and experiment with the IoT Hub service (and other Azure services).

Gateway

A gateway enables connectivity for devices that cannot connect directly to [IoT Hub](#). See also [Field Gateway](#), [Cloud Gateway](#), and [Custom Gateway](#).

Identity registry

The [identity registry](#) is the built-in component of an IoT hub that stores information about the individual devices permitted to connect to an IoT hub.

Interactive message

An interactive message is a [cloud-to-device](#) message that triggers an immediate action in the solution back end. For example, a device might send an alarm about a failure that should be automatically logged in to a CRM system.

Automatic Device Management

Automatic Device Management in Azure IoT Hub automates many of the repetitive and complex tasks of managing large device fleets over the entirety of their lifecycles. With Automatic Device Management, you can target a set of devices based on their properties, define a desired configuration, and let IoT Hub update devices whenever they come into scope. Consists of [automatic device configurations](#) and [IoT Edge automatic deployments](#).

IoT Edge

Azure IoT Edge enables cloud-driven deployment of Azure services and solution-specific code to on-premises devices. IoT Edge devices can aggregate data from other devices to perform computing and analytics before the data is sent to the cloud. For more information, see [Azure IoT Edge](#).

IoT Edge agent

The part of the IoT Edge runtime responsible for deploying and monitoring modules.

IoT Edge device

IoT Edge devices have the IoT Edge runtime installed and are flagged as **IoT Edge device** in the device details. Learn how to [deploy Azure IoT Edge on a simulated device in Linux - preview](#).

IoT Edge automatic deployment

An IoT Edge automatic deployment configures a target set of IoT Edge devices to run a set of IoT Edge modules. Each deployment continuously ensures that all devices that match its target condition are running the specified set of modules, even when new devices are created or are modified to match the target condition. Each IoT Edge device only receives the highest priority deployment whose target condition it meets. Learn more about [IoT Edge automatic deployment](#).

IoT Edge deployment manifest

A Json document containing the information to be copied in one or more IoT Edge devices' module twin(s) to deploy a set of modules, routes, and associated module desired properties.

IoT Edge gateway device

An IoT Edge device with downstream device. The downstream device can be either IoT Edge or not IoT Edge device.

IoT Edge hub

The part of the IoT Edge runtime responsible for module to module communications, upstream (toward IoT Hub) and downstream (away from IoT Hub) communications.

IoT Edge leaf device

An IoT Edge device with no downstream device.

IoT Edge module

An IoT Edge module is a Docker container that you can deploy to IoT Edge devices. It performs a specific task, such as ingesting a message from a device, transforming a message, or sending a message to an IoT hub. It communicates with other modules and sends data to the IoT Edge runtime. [Understand the requirements and tools for developing IoT Edge modules](#).

IoT Edge module identity

A record in the IoT Hub module identity registry detailing the existence and security credentials to be used by a module to authenticate with an edge hub or IoT Hub.

IoT Edge module image

The docker image that is used by the IoT Edge runtime to instantiate module instances.

IoT Edge module twin

A Json document persisted in the IoT Hub that stores the state information for a module instance.

IoT Edge priority

When two IoT Edge deployments target the same device, the deployment with higher priority gets applied. If two deployments have the same priority, the deployment with the later creation date gets applied. Learn more about [priority](#).

IoT Edge runtime

IoT Edge runtime includes everything that Microsoft distributes to be installed on an IoT Edge device. It includes Edge agent, Edge hub, and the IoT Edge security daemon.

IoT Edge set modules to a single device

An operation that copies the content of an IoT Edge manifest on one device's module twin. The underlying API is a generic 'apply configuration', which simply takes an IoT Edge manifest as an input.

IoT Edge target condition

In an IoT Edge deployment, Target condition is any Boolean condition on device twins' tags to select the target devices of the deployment, for example **tag.environment = prod**. The target condition is continuously evaluated to include any new devices that meet the requirements or remove devices that no longer do. Learn more about [target condition](#)

IoT Hub

IoT Hub is a fully managed Azure service that enables reliable and secure bidirectional communications between millions of devices and a solution back end. For more information, see [What is Azure IoT Hub?](#) Using your [Azure subscription](#), you can create IoT hubs to handle your IoT messaging workloads.

IoT Hub metrics

[IoT Hub metrics](#) give you data about the state of the IoT hubs in your [Azure subscription](#). IoT Hub metrics enable you to assess the overall health of the service and the devices connected to it. IoT Hub metrics can help you see what is going on with your IoT hub and investigate root-cause issues without needing to contact Azure support.

IoT Hub query language

The [IoT Hub query language](#) is a SQL-like language that enables you to query your and device twins.

IoT Hub Resource REST API

You can use the [IoT Hub Resource REST API](#) to manage the IoT hubs in your [Azure subscription](#) performing operations such as creating, updating, and deleting hubs.

IoT solution accelerators

Azure IoT solution accelerators package together multiple Azure services into solutions. These solutions enable you to get started quickly with end-to-end implementations of common IoT scenarios. For more information, see [What are Azure IoT solution accelerators?](#)

The IoT extension for Azure CLI

The [IoT extension for Azure CLI](#) is a cross-platform, command-line tool. The tool enables you to manage your devices in the [identity registry](#), send and receive messages and files from your devices, and monitor your IoT hub operations.

Job

Your solution back end can use [jobs](#) to schedule and track activities on a set of devices registered with your IoT hub. Activities include updating device twin [desired properties](#), updating device twin [tags](#), and invoking [direct methods](#). [IoT Hub](#) also uses to [import to and export](#) from the [identity registry](#).

Modules

On the device side, the IoT Hub device SDKs enable you to create [modules](#) where each one opens an independent connection to IoT Hub. This functionality enables you to use separate namespaces for different components on your device.

Module identity and module twin provide the same capabilities as [device identity](#) and [device twin](#) but at a finer granularity. This finer granularity enables capable devices, such as operating system-based devices or firmware devices managing multiple components, to isolate configuration and conditions for each of those components.

Module identity

The module identity is the unique identifier assigned to every module that belong to a device. Module identity is also registered in the [identity registry](#).

Module twin

Similar to device twin, a module twin is JSON document that stores module state information such as metadata, configurations, and conditions. IoT Hub persists a module twin for each module identity that you provision under a device identity in your IoT hub. Module twins enable you to synchronize module conditions and configurations between the module and the solution back end. You can query module twins to locate specific modules and query the status of long-running operations.

MQTT

[MQTT](#) is one of the messaging protocols that [IoT Hub](#) supports for communicating with devices. For more information about the messaging protocols that IoT Hub supports, see [Send and receive messages with IoT Hub](#).

Operations monitoring

IoT Hub [operations monitoring](#) enables you to monitor the status of operations on your IoT hub in real time. [IoT Hub](#) tracks events across several categories of operations. You can opt into sending events from one or more categories to an IoT Hub endpoint for processing. You can monitor the data for errors or set up more complex processing based on data patterns.

Physical device

A physical device is a real device such as a Raspberry Pi that connects to an IoT hub. For convenience, many of the

IoT Hub tutorials use [simulated devices](#) to enable you to run samples on your local machine.

Primary and secondary keys

When you connect to a device-facing or service-facing endpoint on an IoT hub, your [connection string](#) includes key to grant you access. When you add a device to the [identity registry](#) or add a [shared access policy](#) to your hub, the service generates a primary and secondary key. Having two keys enables you to roll over from one key to another when you update a key without losing access to the IoT hub.

Protocol gateway

A protocol gateway is typically deployed in the cloud and provides protocol translation services for devices connecting to [IoT Hub](#). For more information, see [What is Azure IoT Hub?](#)

Quotas and throttling

There are various [quotas](#) that apply to your use of [IoT Hub](#), many of the quotas vary based on the tier of the IoT hub. [IoT Hub](#) also applies [throttles](#) to your use of the service at run time.

Reported configuration

In the context of a [device twin](#), reported configuration refers to the complete set of properties and metadata in the device twin that should be reported to the solution back end.

Reported properties

In the context of a [device twin](#), reported properties is a subsection of the device twin used with [desired properties](#) to synchronize device configuration or condition. Reported properties can only be set by the [device app](#) and can be read and queried by a [back-end app](#).

Resource group

[Azure Resource Manager](#) uses resource groups to group related resources together. You can use a resource group to perform operations on all the resources on the group simultaneously.

Retry policy

You use a retry policy to handle [transient errors](#) when you connect to a cloud service.

Routing rules

You configure [routing rules](#) in your IoT hub to route device-to-cloud messages to a [built-in endpoint](#) or to [custom endpoints](#) for processing by your solution back end.

SASL PLAIN

SASL PLAIN is a protocol that the AMQP protocol uses to transfer security tokens.

Service REST API

You can use the [Service REST API](#) from the solution back end to manage your devices. The API enables you to retrieve and update [device twin](#) properties, invoke [direct methods](#), and schedule [jobs](#). Typically, you should use one of the higher-level [service SDKs](#) as shown in the IoT Hub tutorials.

Shared access signature

Shared Access Signatures (SAS) are an authentication mechanism based on SHA-256 secure hashes or URIs. SAS authentication has two components: a *Shared Access Policy* and a *Shared Access Signature* (often called a token). A device uses SAS to authenticate with an IoT hub. [Back-end apps](#) also use SAS to authenticate with the service-facing endpoints on an IoT hub. Typically, you include the SAS token in the [connection string](#) that an app uses to establish a connection to an IoT hub.

Shared access policy

A shared access policy defines the permissions granted to anyone who has a valid [primary or secondary key](#) associated with that policy. You can manage the shared access policies and keys for your hub in the [portal](#).

Simulated device

For convenience, many of the IoT Hub tutorials use simulated devices to enable you to run samples on your local machine. In contrast, a [physical device](#) is a real device such as a Raspberry Pi that connects to an IoT hub.

Solution

A *solution* can refer to a Visual Studio solution that includes one or more projects. A *solution* might also refer to an IoT solution that includes elements such as devices, [device apps](#), an IoT hub, other Azure services, and [back-end apps](#).

Subscription

An Azure subscription is where billing takes place. Each Azure resource you create or Azure service you use is associated with a single subscription. Many quotas also apply at the level of a subscription.

System properties

In the context of a [device twin](#), system properties are read-only and include information regarding the device usage such as last activity time and connection state.

Tags

In the context of a [device twin](#), tags are device metadata stored and retrieved by the solution back end in the form of a JSON document. Tags are not visible to apps on a device.

Telemetry

Devices collect telemetry data, such as wind speed or temperature, and use data-point messages to send the telemetry to an IoT hub.

Token service

You can use a token service to implement an authentication mechanism for your devices. It uses an IoT Hub [shared access policy](#) with **DeviceConnect** permissions to create *device-scoped* tokens. These tokens enable a device to connect to your IoT hub. A device uses a custom authentication mechanism to authenticate with the token service. If the device authenticates successfully, the token service issues a SAS token for the device to use to access your IoT hub.

Twin queries

Device and module twin queries use the SQL-like IoT Hub query language to retrieve information from your device twins or module twins. You can use the same IoT Hub query language to retrieve information about running in your IoT hub.

Twin synchronization

Twin synchronization uses the [desired properties](#) in your device twins or module twins to configure your devices or modules and retrieve [reported properties](#) from them to store in the twin.

X.509 client certificate

A device can use an X.509 certificate to authenticate with [IoT Hub](#). Using an X.509 certificate is an alternative to using a [SAS token](#).

Device Authentication using X.509 CA Certificates

8/17/2018 • 4 minutes to read

This article describes how to use X.509 Certificate Authority (CA) certificates to authenticate devices connecting IoT Hub. In this article you will learn:

- How to get an X.509 CA certificate
- How to register the X.509 CA certificate to IoT Hub
- How to sign devices using X.509 CA certificates
- How devices signed with X.509 CA are authenticated

Overview

The X.509 CA feature enables device authentication to IoT Hub using a Certificate Authority (CA). It greatly simplifies initial device enrollment process, and supply chain logistics during device manufacturing. [Learn more in this scenario article about the value of using X.509 CA certificates](#) for device authentication. We encourage you to read this scenario article before proceeding as it explains why the steps that follow exist.

Prerequisite

Using the X.509 CA feature requires that you have an IoT Hub account. [Learn how to create an IoT Hub instance](#) if you don't already have one.

How to get an X.509 CA certificate

The X.509 CA certificate is at the top of the chain of certificates for each of your devices. You may purchase or create one depending on how you intend to use it.

For production environment, we recommend that you purchase an X.509 CA certificate from a public root certificate authority. Purchasing a CA certificate has the benefit of the root CA acting as a trusted third party to vouch for the legitimacy of your devices. Consider this option if you intend your devices to be part of an open IoT network where they are expected to interact with third-party products or services.

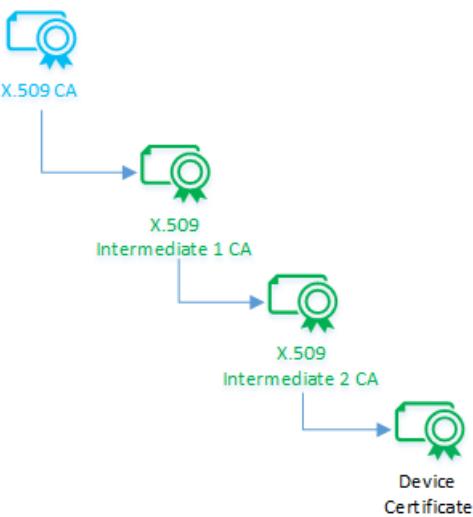
You may also create a self-signed X.509 CA for experimentation or for use in closed IoT networks.

Regardless of how you obtain your X.509 CA certificate, make sure to keep its corresponding private key secret and protected at all times. This is necessary for trust building trust in the X.509 CA authentication.

Learn how to [create a self-signed CA certificate](#), which you can use for experimentation throughout this feature description.

Sign devices into the certificate chain of trust

The owner of an X.509 CA certificate can cryptographically sign an intermediate CA who can in turn sign another intermediate CA, and so on, until the last intermediate CA terminates this process by signing a device. The result is a cascaded chain of certificates known as a certificate chain of trust. In real life this plays out as delegation of trust towards signing devices. This delegation is important because it establishes a cryptographically variable chain of custody and avoids sharing of signing keys.



Learn here how to [create a certificate chain](#) as done when signing devices.

How to register the X.509 CA certificate to IoT Hub

Register your X.509 CA certificate to IoT Hub where it will be used to authenticate your devices during registration and connection. Registering the X.509 CA certificate is a two-step process that comprises certificate file upload and proof of possession.

The upload process entails uploading a file that contains your certificate. This file should never contain any private keys.

The proof of possession step involves a cryptographic challenge and response process between you and IoT Hub. Given that digital certificate contents are public and therefore susceptible to eavesdropping, IoT Hub would like to ascertain that you really own the CA certificate. It shall do so by generating a random challenge that you must sign with the CA certificate's corresponding private key. If you kept the private key secret and protected as earlier advised, then only you will possess the knowledge to complete this step. Secrecy of private keys is the source of trust in this method. After signing the challenge, complete this step by uploading a file containing the results.

Learn here how to [register your CA certificate](#).

How to create a device on IoT Hub

To preclude device impersonation, IoT Hub requires you to let it know what devices to expect. You do this by creating a device entry in the IoT Hub's device registry. This process is automated when using IoT Hub [Device Provisioning Service](#).

Learn here how to [manually create a device in IoT Hub](#).

Authenticating devices signed with X.509 CA certificates

With X.509 CA certificate registered and devices signed into a certificate chain of trust, what remains is device authentication when the device connects, even for the first time. When an X.509 CA signed device connects, it uploads its certificate chain for validation. The chain includes all intermediate CA and device certificates. With this information, IoT Hub authenticates the device in a two-step process. IoT Hub cryptographically validates the certificate chain for internal consistency, and then issues a proof-of-possession challenge to the device. IoT Hub declares the device authentic on a successful proof-of-possession response from the device. This declaration assumes that the device's private key is protected and that only the device can successfully respond to this challenge. We recommend use of secure chips like Hardware Secure Modules (HSM) in devices to protect private keys.

A successful device connection to IoT Hub completes the authentication process and is also indicative of a proper

setup.

Learn here how to [complete this device connection step](#).

Next Steps

Learn about [the value of X.509 CA authentication](#) in IoT.

Get started with IoT Hub [Device Provisioning Service](#).

Conceptual understanding of X.509 CA certificates in the IoT industry

2/1/2019 • 11 minutes to read

This article describes the value of using X.509 certificate authority (CA) certificates in IoT device manufacturing and authentication to IoT Hub. It includes information about supply chain setup and highlight advantages.

This article describes:

- What X.509 CA certificates are and how to get them
- How to register your X.509 CA certificate to IoT Hub
- How to set up a manufacturing supply chain for X.509 CA-based authentication
- How devices signed with X.509 CA connect to IoT Hub

Overview

X.509 Certificate Authority (CA) authentication is an approach for authenticating devices to IoT Hub using a method that dramatically simplifies device identity creation and life-cycle management in the supply chain.

A distinguishing attribute of the X.509 CA authentication is a one-to-many relationship a CA certificate has with its downstream devices. This relationship enables registration of any number of devices into IoT Hub by registering an X.509 CA certificate once, otherwise device unique certificates must be pre-registered for every device before a device can connect. This one-to-many relationship also simplifies device certificates life-cycle management operations.

Another important attribute of the X.509 CA authentication is simplification of supply chain logistics. Secure authentication of devices requires that each device holds a unique secret like a key as basis for trust. In certificates-based authentication, this secret is a private key. A typical device manufacturing flow involves multiple steps and custodians. Securely managing device private keys across multiple custodians and maintaining trust is difficult and expensive. Using certificate authorities solves this problem by signing each custodian into a cryptographic chain of trust rather than entrusting them with device private keys. Each custodian in turn signs devices at their respective process step of the manufacturing flow. The overall result is an optimal supply chain with built-in accountability through use of the cryptographic chain of trust. It is worth noting that this process yields the most security when devices protect their unique private keys. To this end, we urge the use of Hardware Secure Modules (HSM) capable of internally generating private keys that will never see the light of day.

This article offers an end-to-end view of using the X.509 CA authentication, from supply chain setup to device connection, while making use of a real world example to solidify understanding.

Introduction

The X.509 CA certificate is a digital certificate whose holder can sign other certificates. This digital certificate is X.509 because it conforms to a certificate formatting standard prescribed by IETF's RFC 5280 standard, and is a certificate authority (CA) because its holder can sign other certificates.

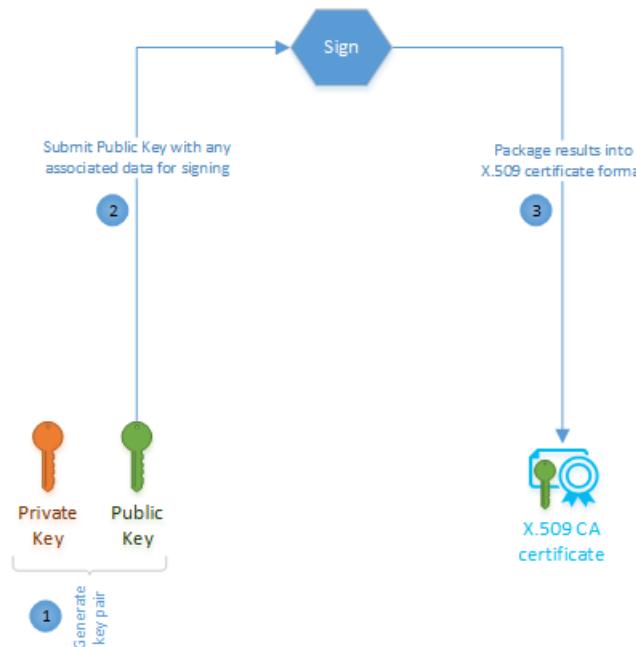
The use of X.509 CA is best understood in relation to a concrete example. Consider Company-X, a maker of Smart-X-Widgets designed for professional installation. Company-X outsources both manufacturing and installation. It contracts manufacturer Factory-Y to manufacture the Smart-X-Widgets, and service provider Technician-Z to install. Company-X desires that Smart-X-Widget directly ships from Factory-Y to Technician-Z for installation and that it connects directly to Company-X's instance of IoT Hub after installation without further

intervention from Company-X. To make this happen, Company-X need to complete a few one-time setup operations to prime Smart-X-Widget for automatic connection. With the end-to-end scenario in mind, the rest of this article is structured as follows:

- Acquire the X.509 CA certificate
- Register X.509 CA certificate to IoT Hub
- Sign devices into a certificate chain of trust
- Device connection

Acquire the X.509 CA certificate

Company-X has the option of purchasing an X.509 CA certificate from a public root certificate authority or creating one through a self-signed process. One option would be optimal over the other depending on the application scenario. Regardless of the option, the process entails two fundamental steps, generating a public/private key pair and signing the public key into a certificate.



Details on how to accomplish these steps differ with various service providers.

Purchasing an X.509 CA certificate

Purchasing a CA certificate has the benefit of having a well-known root CA act as a trusted third party to vouch for the legitimacy of IoT devices when the devices connect. Company-X would choose this option if they intend Smart-X-Widget to interact with third party products or services after initial connection to IoT Hub.

To purchase an X.509 CA certificate, Company-X would choose a root certificates services provider. An internet search for the phrase 'Root CA' will yield good leads. The root CA will guide Company-X on how to create the public/private key pair and how to generate a Certificate Signing Request (CSR) for their services. A CSR is the formal process of applying for a certificate from a certificate authority. The outcome of this purchase is a certificate for use as an authority certificate. Given the ubiquity of X.509 certificates, the certificate is likely to have been properly formatted to IETF's RFC 5280 standard.

Creating a Self-Signed X.509 CA certificate

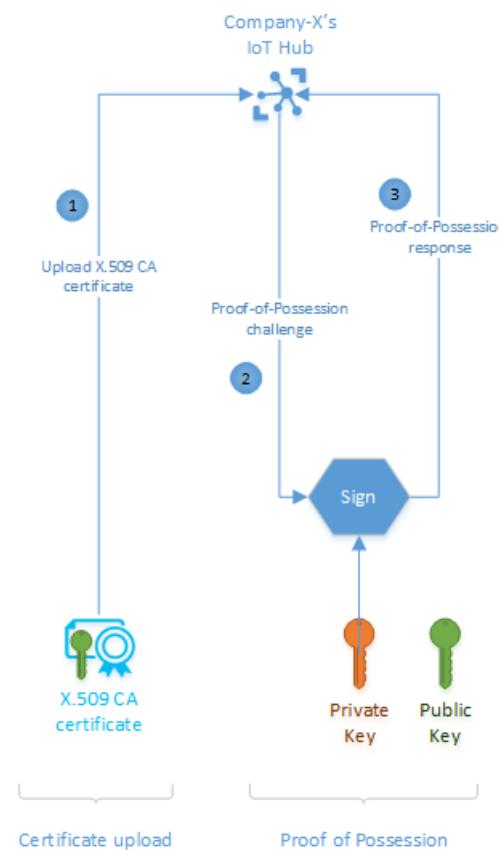
The process to create a Self-Signed X.509 CA certificate is similar to purchasing with the exception of involving a third party signer like the root certificate authority. In our example, Company-X will sign its authority certificate instead of a root certificate authority. Company-X may choose this option for testing until they're ready to purchase an authority certificate. Company-X may also use a self-signed X.509 CA certificate in production, if

Smart-X-Widget is not intended to connect to any third party services outside of the IoT Hub.

Register the X.509 certificate to IoT Hub

Company-X needs to register the X.509 CA to IoT Hub where it will serve to authenticate Smart-X-Widgets as they connect. This is a one-time process that enables the ability to authenticate and manage any number of Smart-X-Widget devices. This process is one-time because of a one-to-many relationship between authority certificate and devices and also constitutes one of the main advantages of using the X.509 CA authentication method. The alternative is to upload individual certificate thumbprints for each and every Smart-X-Widget device thereby adding to operational costs.

Registering the X.509 CA certificate is a two-step process, the certificate upload and certificate proof-of-possession.



X.509 CA Certificate Upload

The X.509 CA certificate upload process is just that, upload the CA certificate to IoT Hub. IoT Hub expects the certificate in a file. Company-X simply uploads the certificate file. The certificate file MUST NOT under any circumstances contain any private keys. Best practices from standards governing Public Key Infrastructure (PKI) mandates that knowledge of Company-X's private in this case resides exclusively within Company-X.

Proof-of-Possession of the Certificate

The X.509 CA certificate, just like any digital certificate, is public information that is susceptible to eavesdropping. As such, an eavesdropper may intercept a certificate and try to upload it as their own. In our example, IoT Hub would like to make sure that the CA certificate Company-X is uploading really belongs to Company-X. It does so by challenging Company-X to proof that they in fact possess the certificate through a [proof-of-possession \(PoP\) flow](#). The proof-of-possession flow entails IoT Hub generating a random number to be signed by Company-X using its private key. If Company-X followed PKI best practices and protected their private key then only they would be in position to correctly respond to the proof-of-possession challenge. IoT Hub proceeds to register the X.509 CA certificate upon a successful response of the proof-of-possession challenge.

A successful response to the proof-of-possession challenge from IoT Hub completes the X.509 CA registration.

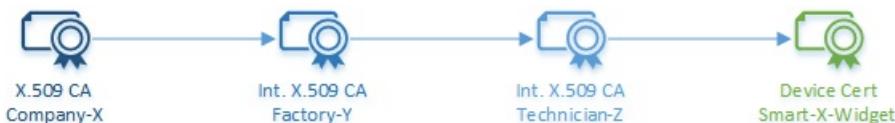
Sign Devices into a Certificate Chain of Trust

IoT requires every device to possess a unique identity. These identities are in the form certificates for certificate-based authentication schemes. In our example, this means every Smart-X-Widget must possess a unique device certificate. How does Company-X setup for this in its supply chain?

One way to go about this is to pre-generate certificates for Smart-X-Widgets and entrusting knowledge of corresponding unique device private keys with supply chain partners. For Company-X, this means entrusting Factory-Y and Technician-Z. While this is a valid method, it comes with challenges that must be overcome to ensure trust as follows:

1. Having to share device private keys with supply chain partners, besides ignoring PKI best practices of never sharing private keys, makes building trust in the supply chain expensive. It means capital systems like secure rooms to house device private keys, and processes like periodic security audits need to be installed. Both add cost to the supply chain.
2. Securely accounting for devices in the supply chain and later managing them in deployment becomes a one-to-one task for every key-to-device pair from the point of device unique certificate (hence private key) generation to device retirement. This precludes group management of devices unless the concept of groups is explicitly built into the process somehow. Secure accounting and device life-cycle management, therefore, becomes a heavy operations burden. In our example, Company-X would bear this burden.

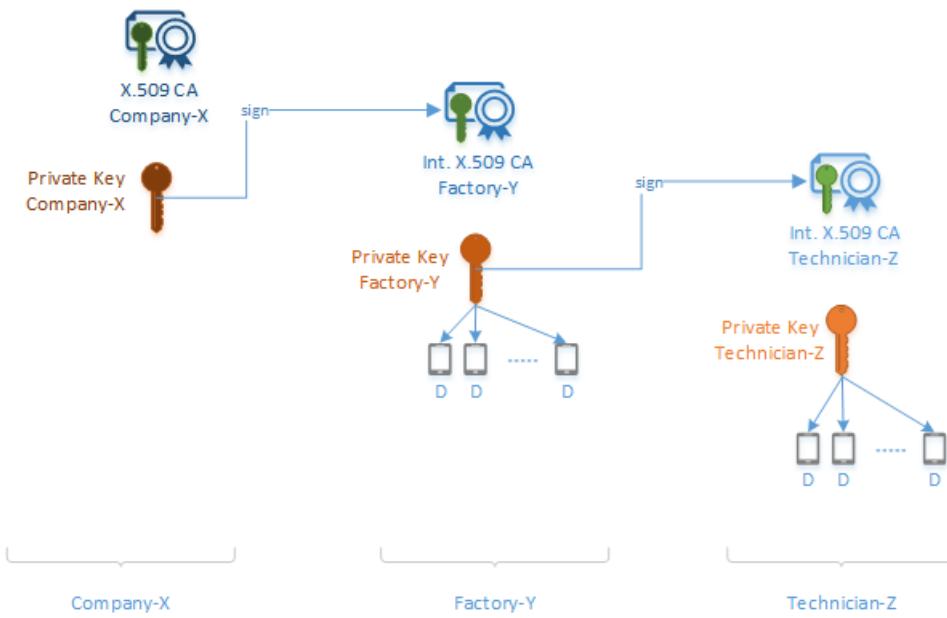
X.509 CA certificate authentication offers elegant solutions to afore listed challenges through the use of certificate chains. A certificate chain results from a CA signing an intermediate CA that in turn signs another intermediate CA and so goes on until a final intermediate CA signs a device. In our example, Company-X signs Factory-Y, which in turn signs Technician-Z that finally signs Smart-X-Widget.



Above cascade of certificates in the chain presents the logical hand-off of authority. Many supply chains follow this logical hand-off whereby each intermediate CA gets signed into the chain while receiving all upstream CA certificates, and the last intermediate CA finally signs each device and inject all the authority certificates from the chain into the device. This is common when the contract manufacturing company with a hierarchy of factories commissions a particular factory to do the manufacturing. While the hierarchy may be several levels deep (for example, by geography/product type/manufacturing line), only the factory at the end gets to interact with the device but the chain is maintained from the top of the hierarchy.

Alternate chains may have different intermediate CA interact with the device in which case the CA interacting with the device injects certificate chain content at that point. Hybrid models are also possible where only some of the CA has physical interaction with the device.

In our example, both Factory-Y and Technician-Z interact with the Smart-X-Widget. While Company-X owns Smart-X-Widget, it actually does not physically interact with it in the entire supply chain. The certificate chain of trust for Smart-X-Widget therefore comprise Company-X signing Factory-Y which in turn signs Technician-Z that will then provide final signature to Smart-X-Widget. The manufacture and installation of Smart-X-Widget comprise Factory-Y and Technician-Z using their respective intermediate CA certificates to sign each and every Smart-X-Widgets. The end result of this entire process is Smart-X-Widgets with unique device certificates and certificate chain of trust going up to Company-X CA certificate.



This is a good point to review the value of the X.509 CA method. Instead of pre-generating and handing off certificates for every Smart-X-Widget into the supply chain, Company-X only had to sign Factory-Y once. Instead of having to track every device throughout the device's life-cycle, Company-X may now track and manage devices through groups that naturally emerge from the supply chain process, for example, devices installed by Technician-Z after July of some year.

Last but not least, the CA method of authentication infuses secure accountability into the device manufacturing supply chain. Because of the certificate chain process, the actions of every member in the chain is cryptographically recorded and verifiable.

This process relies on certain assumptions that must be surfaced for completeness. It requires independent creation of device unique public/private key pair and that the private key be protected within the device. Fortunately, secure silicon chips in the form of Hardware Secure Modules (HSM) capable of internally generating keys and protecting private keys exist. Company-X only need to add one of such chips into Smart-X-Widget's component bill of materials.

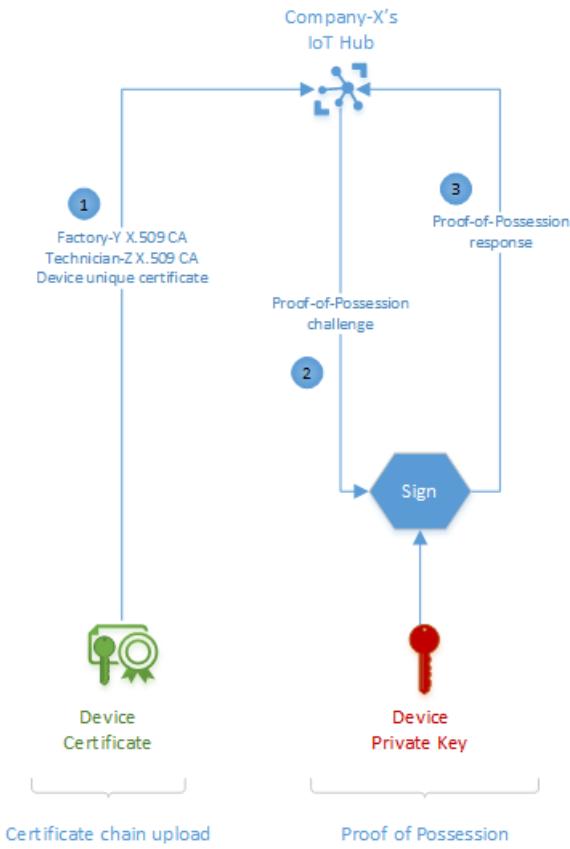
Device Connection

Previous sections above have been building up to device connection. By simply registering an X.509 CA certificate to IoT Hub one time, how do potentially millions of devices connect and get authenticated from the first time? Simple; through the same certificate upload and proof-of-possession flow we earlier encountered with registering the X.509 CA certificate.

Devices manufactured for X.509 CA authentication are equipped with device unique certificates and a certificate chain from their respective manufacturing supply chain. Device connection, even for the very first time, happens in a two-step process: certificate chain upload and proof-of-possession.

During the certificate chain upload, the device uploads its device unique certificate together with the certificate chain installed within it to IoT Hub. Using the pre-registered X.509 CA certificate, IoT Hub can cryptographically validate a couple of things, that the uploaded certificate chain is internally consistent, and that the chain was originated by the valid owner of the X.509 CA certificate. Just as with the X.509 CA registration process, IoT Hub would initiate a proof-of-possession challenge-response process to ascertain that the chain and hence device certificate actually belongs to the device uploading it. It does so by generating a random challenge to be signed by the device using its private key for validation by IoT Hub. A successful response triggers IoT Hub to accept the device as authentic and grant it connection.

In our example, each Smart-X-Widget would upload its device unique certificate together with Factory-Y and Technician-Z X.509 CA certificates and then respond to the proof-of-possession challenge from IoT Hub.



Notice that the foundation of trust rests in protecting private keys including device private keys. We therefore cannot stress enough the importance of secure silicon chips in the form of Hardware Secure Modules (HSM) for protecting device private keys, and the overall best practice of never sharing any private keys, like one factory entrusting another with its private key.

Understand and use Azure IoT Hub SDKs

2/28/2019 • 4 minutes to read

There are two categories of software development kits (SDKs) for working with IoT Hub:

- **IoT Hub Device SDKs** enable you to build apps that run on your IoT devices using device client or module client. These apps send telemetry to your IoT hub, and optionally receive messages, job, method, or twin updates from your IoT hub. You can also use module client to author [modules for Azure IoT Edge runtime](#).
- **IoT Hub Service SDKs** enable you to build backend applications to manage your IoT hub, and optionally send messages, schedule jobs, invoke direct methods, or send desired property updates to your IoT devices or modules.

In addition, we also provide a set of SDKs for working with the [Device Provisioning Service](#).

- **Provisioning Device SDKs** enable you to build apps that run on your IoT devices to communicate with the Device Provisioning Service.
- **Provisioning Service SDKs** enable you to build backend applications to manage your enrollments in the Device Provisioning Service.

Learn about the [benefits of developing using Azure IoT SDKs](#).

NOTE

Some of the features mentioned in this article, like cloud-to-device messaging, device twins, and device management, are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

OS platform and hardware compatibility

Supported platforms for the SDKs can be found in [Azure IoT SDKs Platform Support](#).

For more information about SDK compatibility with specific hardware devices, see the [Azure Certified for IoT device catalog](#) or individual repository.

Azure IoT Hub Device SDKs

The Microsoft Azure IoT device SDKs contain code that facilitates building applications that connect to and are managed by Azure IoT Hub services.

Azure IoT Hub device SDK for .NET:

- Download from [Nuget](#). The namespace is Microsoft.Azure.Devices.Clients, which contains IoT Hub Device Clients (DeviceClient, ModuleClient).
- [Source code](#)
- [API reference](#)
- [Module reference](#)

Azure IoT Hub device SDK for C (ANSI C - C99):

- Install from [apt-get](#), [MBED](#), [Arduino IDE](#) or [iOS](#)
- [Source code](#)

- [Compile the C Device SDK](#)
- [API reference](#)
- [Module reference](#)
- [Porting the C SDK to other platforms](#)
- [Developer documentation](#) for information on cross-compiling, getting started on different platforms, etc.

Azure IoT Hub device SDK for Java:

- Add to [Maven](#) project
- [Source code](#)
- [API reference](#)
- [Module reference](#)

Azure IoT Hub device SDK for Node.js:

- Install from [npm](#)
- [Source code](#)
- [API reference](#)
- [Module reference](#)

Azure IoT Hub device SDK for Python:

- Install from [pip](#)
- [Source code](#)
- API reference: see [C API reference](#)

Azure IoT Hub device SDK for iOS:

- Install from [CocoaPod](#)
- [Samples](#)
- API reference: see [C API reference](#)

Azure IoT Hub Service SDKs

The Azure IoT service SDKs contain code to facilitate building applications that interact directly with IoT Hub to manage devices and security.

Azure IoT Hub service SDK for .NET:

- Download from [Nuget](#). The namespace is Microsoft.Azure.Devices, which contains IoT Hub Service Clients (RegistryManager, ServiceClients).
- [Source code](#)
- [API reference](#)

Azure IoT Hub service SDK for Java:

- Add to [Maven](#) project
- [Source code](#)
- [API reference](#)

Azure IoT Hub service SDK for Node.js:

- Download from [npm](#)
- [Source code](#)

- [API reference](#)

Azure IoT Hub service SDK for Python:

- Download from [pip](#)
- [Source code](#)

Azure IoT Hub service SDK for C:

- Download from [apt-get](#), [MBED](#), [Arduino IDE](#), or [Nuget](#)
- [Source code](#)

Azure IoT Hub service SDK for iOS:

- Install from [CocoaPod](#)
- [Samples](#)

NOTE

See the readme files in the GitHub repositories for information about using language and platform-specific package managers to install binaries and dependencies on your development machine.

Microsoft Azure Provisioning SDKs

The **Microsoft Azure Provisioning SDKs** enable you to provision devices to your IoT Hub using the [Device Provisioning Service](#).

Azure Provisioning device and service SDKs for C#:

- Download from [Device SDK](#) and [Service SDK](#) from NuGet.
- [Source code](#)
- [API reference](#)

Azure Provisioning device and service SDKs for C:

- Install from [apt-get](#), [MBED](#), [Arduino IDE](#) or [iOS](#)
- [Source code](#)
- [API reference](#)

Azure Provisioning device and service SDKs for Java:

- Add to [Maven](#) project
- [Source code](#)
- [API reference](#)

Azure Provisioning device and service SDKs for Node.js:

- [Source code](#)
- [API reference](#)
- Download [Device SDK](#) and [Service SDK](#) from npm

Azure Provisioning device and service SDKs for Python:

- [Source code](#)
- Download [Device SDK](#) and [Service SDK](#) from pip

Next steps

Azure IoT SDKs also provide a set of tools to help with development:

- [iothub-diagnostics](#): a cross-platform command line tool to help diagnose issues related to connection with IoT Hub.
- [device-explorer](#): a Windows desktop application to connect to your IoT Hub.

Relevant docs related to development using the Azure IoT SDKs:

- Learn about [how to manage connectivity and reliable messaging](#) using the IoT Hub SDKs.
- Learn about how to [develop for mobile platforms](#) such as iOS and Android.
- [Azure IoT SDK platform support](#)

Other reference topics in this IoT Hub developer guide include:

- [IoT Hub endpoints](#)
- [IoT Hub query language for device twins, jobs, and message routing](#)
- [Quotas and throttling](#)
- [IoT Hub MQTT support](#)
- [IoT Hub REST API reference](#)

Azure IoT SDKs Platform Support

2/22/2019 • 2 minutes to read

The [Azure IoT SDKs](#) are a set of libraries to interact with IoT Hub and the Device Provisioning Service with broad language and platform support. The SDKs run on most common platforms, and developers can port the C SDK to specific platform by following the [Porting Guidance](#).

Microsoft supports a variety of operating systems/platforms/frameworks and can be extended using the Azure IoT C SDK. Some are supported officially by the team, grouped into tiers that represent the level of support users can expect. *Fully supported platforms* means that Microsoft:

- Continuously builds and runs end-to-end tests against master and the LTS supported version(s). To provide test coverage across different versions, we generally test against the latest LTS version and the most popular version. Other versions of the same platform may be supported via platform version compatibility.
- Provides installation guidance or packages if applicable.
- Fully supports the platforms on GitHub.

In addition, a list of partners has ported our C SDK on to more platforms and they are maintaining the platform abstraction layer (PAL). [Azure Certified for IoT Device Catalog](#) also features a list of OS platforms the various SDKs have been tested against. The SDKs also regularly build on these platforms, with limited testing and support:

- MBED2
- Arduino
- Windows CE 2013 (deprecate in October 2018)
- .NET Standard 1.3 with .NET Core 2.1 and .NET Framework 4.7
- Xamarin iOS, Android, UWP

Supported platforms

There are several platforms supported.

C SDK

OS	ARCH	COMPILER	TLS LIBRARY
Ubuntu 16.04 LTS	X64	gcc-5.4.0	openssl - 1.0.2g
Ubuntu 18.04 LTS	X64	gcc-7.3	WolfSSL – 1.13
Ubuntu 18.04 LTS	X64	Clang 6.0.X	Openssl – 1.1.0g
OSX 10.13.4	x64	XCode 9.4.1	Native OSX
Windows Server 2016	x64	Visual Studio 14.0.X	SChannel
Windows Server 2016	x86	Visual Studio 14.0.X	SChannel
Debian 9 Stretch	x64	gcc-7.3	Openssl – 1.1.0f

Python SDK

OS	ARCH	COMPILER	TLS LIBRARY
Windows Server 2016	x86	Python 2.7	openssl
Windows Server 2016	x64	Python 2.7	openssl
Windows Server 2016	x86	Python 3.5	openssl
Windows Server 2016	x64	Python 3.5	openssl
Ubuntu 18.04 LTS	x86	Python 2.7	openssl
Ubuntu 18.04 LTS	x86	Python 3.4	openssl
MacOS High Sierra	x64	Python 2.7	openssl

.NET SDK

OS	ARCH	FRAMEWORK	STANDARD
Ubuntu 16.04 LTS	X64	.NET Core 2.1	.NET standard 2.0
Windows Server 2016	X64	.NET Core 2.1	.NET standard 2.0
Windows Server 2016	X64	.NET Framework 4.7	.NET standard 2.0
Windows Server 2016	X64	.NET Framework 4.5.1	N/A

Node.js SDK

OS	ARCH	NODE VERSION
Ubuntu 16.04 LTS (using node 6 docker image)	X64	Node 6
Windows Server 2016	X64	Node 6

Java SDK

OS	ARCH	JAVA VERSION
Ubuntu 16.04 LTS	X64	Java 8
Windows Server 2016	X64	Java 8
Android API 28	X64	Java 8
Android Things	X64	Java 8

Partner supported platforms

Customers can extend our platform support by porting the Azure IoT C SDK, specifically, creating the platform abstraction layer (PAL) of the SDK. Microsoft works with partners to provide extended support. A list of partners

has ported the C SDK on to more platforms and maintaining the PAL.

PARTNER	DEVICES	LINK	SUPPORT
Espressif	ESP32 ESP8266	Esp-azure	GitHub
Qualcomm	Qualcomm MDM9206 LTE IoT Modem	Qualcomm LTE for IoT SDK	Forum
ST Microelectronics	STM32L4 Series STM32F4 Series STM32F7 Series STM32L4 Discovery Kit for IoT node	X-CUBE-CLOUD X-CUBE-AZURE P-NUCLEO-AZURE FP-CLD-AZURE	Support
Texas Instruments	CC3220SF Launchpad CC3220S Launchpad MSP432E4 Launchpad	Azure IoT Plugin for SimpleLink	TI E2E Forum TI E2E Forum for CC3220 TI E2E Forum for MSP432E4

Next steps

- [Device and service SDKs](#)
- [Porting Guidance](#)

Azure IoT device SDK for C

2/28/2019 • 18 minutes to read

The **Azure IoT device SDK** is a set of libraries designed to simplify the process of sending messages to and receiving messages from the **Azure IoT Hub** service. There are different variations of the SDK, each targeting a specific platform, but this article describes the **Azure IoT device SDK for C**.

NOTE

Some of the features mentioned in this article, like cloud-to-device messaging, device twins, and device management, are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

The Azure IoT device SDK for C is written in ANSI C (C99) to maximize portability. This feature makes the libraries well-suited to operate on multiple platforms and devices, especially where minimizing disk and memory footprint is a priority.

There are a broad range of platforms on which the SDK has been tested (see the [Azure Certified for IoT device catalog](#) for details). Although this article includes walkthroughs of sample code running on the Windows platform, the code described in this article is identical across the range of supported platforms.

The following video presents an overview of the Azure IoT SDK for C:

This article introduces you to the architecture of the Azure IoT device SDK for C. It demonstrates how to initialize the device library, send data to IoT Hub, and receive messages from it. The information in this article should be enough to get started using the SDK, but also provides pointers to additional information about the libraries.

SDK architecture

You can find the [Azure IoT device SDK for C](#) GitHub repository and view details of the API in the [C API reference](#).

The latest version of the libraries can be found in the **master** branch of the repository:

[Azure / azure-iot-sdk-c](#)

Code Issues 17 Pull requests 1 Projects 0 Wiki Pulse Graphs

A C99 SDK for connecting devices to Microsoft Azure IoT services

azure iothub iot device-sdk sdk c c99 azure-iot azure-iot-sdks service-sdk microsoft azure-iothub mbed serialization-library serializer embedded

2,286 commits 20 branches 33 releases 59 contributors

Branch: master New pull request Create new file Upload files Find file Clone or download ▾

Author	Commit Message	Time Ago
dcristo	Update to latest dependencies and remove linking of websockets	Latest commit dae9353 8 hours ago
.github	Added templates for contributions	9 days ago
build_all	release_2017_03_10_after_bump_version	7 days ago
c-utility @ d0b0d17	Update to latest dependencies and remove linking of websockets	8 hours ago
certs	add missing root CAs	9 days ago
configs	Update to latest dependencies and remove linking of websockets	8 hours ago
doc	moved contribution guidelines images to local repo	9 days ago
iothub_client	Add proxy_data option to MQTT transport common module	13 hours ago
iothub_service_client	prepare for VS 2017	8 days ago
jenkins	Fix script path	29 days ago
parson @ ba2a854	use latest parson	13 days ago
serializer	squelch an error message	21 hours ago
testtools	Update to latest C shared utility and fix tls io config	15 days ago
tools	Scripts to add environment variables	2 months ago
uamqp @ b9ce795	Update to latest dependencies and remove linking of websockets	8 hours ago
umqtt @ bb9ff93	Update to latest dependencies and remove linking of websockets	8 hours ago
.gitattributes	Azure IoT SDKs	2 years ago

- The core implementation of the SDK is in the **iothub_client** folder that contains the implementation of the lowest API layer in the SDK: the **IoTHubClient** library. The **IoTHubClient** library contains APIs implementing raw messaging for sending messages to IoT Hub and receiving messages from IoT Hub. When using this library, you are responsible for implementing message serialization, but other details of communicating with IoT Hub are handled for you.
- The **serializer** folder contains helper functions and samples that show you how to serialize data before sending to Azure IoT Hub using the client library. The use of the serializer is not mandatory and is provided as a convenience. To use the **serializer** library, you define a model that specifies the data to send to IoT Hub and the messages you expect to receive from it. Once the model is defined, the SDK provides you with an API surface that enables you to easily work with device-to-cloud and cloud-to-device messages without worrying about the serialization details. The library depends on other open source libraries that implement transport using protocols such as MQTT and AMQP.
- The **IoTHubClient** library depends on other open source libraries:
 - The **Azure C shared utility** library, which provides common functionality for basic tasks (such as strings, list manipulation, and IO) needed across several Azure-related C SDKs.
 - The **Azure uAMQP** library, which is a client-side implementation of AMQP optimized for resource constrained devices.
 - The **Azure uMQTT** library, which is a general-purpose library implementing the MQTT protocol and optimized for resource constrained devices.

Use of these libraries is easier to understand by looking at example code. The following sections walk you

through several of the sample applications that are included in the SDK. This walkthrough should give you a good feel for the various capabilities of the architectural layers of the SDK and an introduction to how the APIs work.

Before you run the samples

Before you can run the samples in the Azure IoT device SDK for C, you must [create an instance of the IoT Hub service](#) in your Azure subscription. Then complete the following tasks:

- Prepare your development environment
- Obtain device credentials.

Prepare your development environment

Packages are provided for common platforms (such as NuGet for Windows or apt_get for Debian and Ubuntu) and the samples use these packages when available. In some cases, you need to compile the SDK for or on your device. If you need to compile the SDK, see [Prepare your development environment](#) in the GitHub repository.

To obtain the sample application code, download a copy of the SDK from GitHub. Get your copy of the source from the **master** branch of the [GitHub repository](#).

Obtain the device credentials

Now that you have the sample source code, the next thing to do is to get a set of device credentials. For a device to be able to access an IoT hub, you must first add the device to the IoT Hub identity registry. When you add your device, you get a set of device credentials that you need for the device to be able to connect to the IoT hub. The sample applications discussed in the next section expect these credentials in the form of a **device connection string**.

There are several open source tools to help you manage your IoT hub.

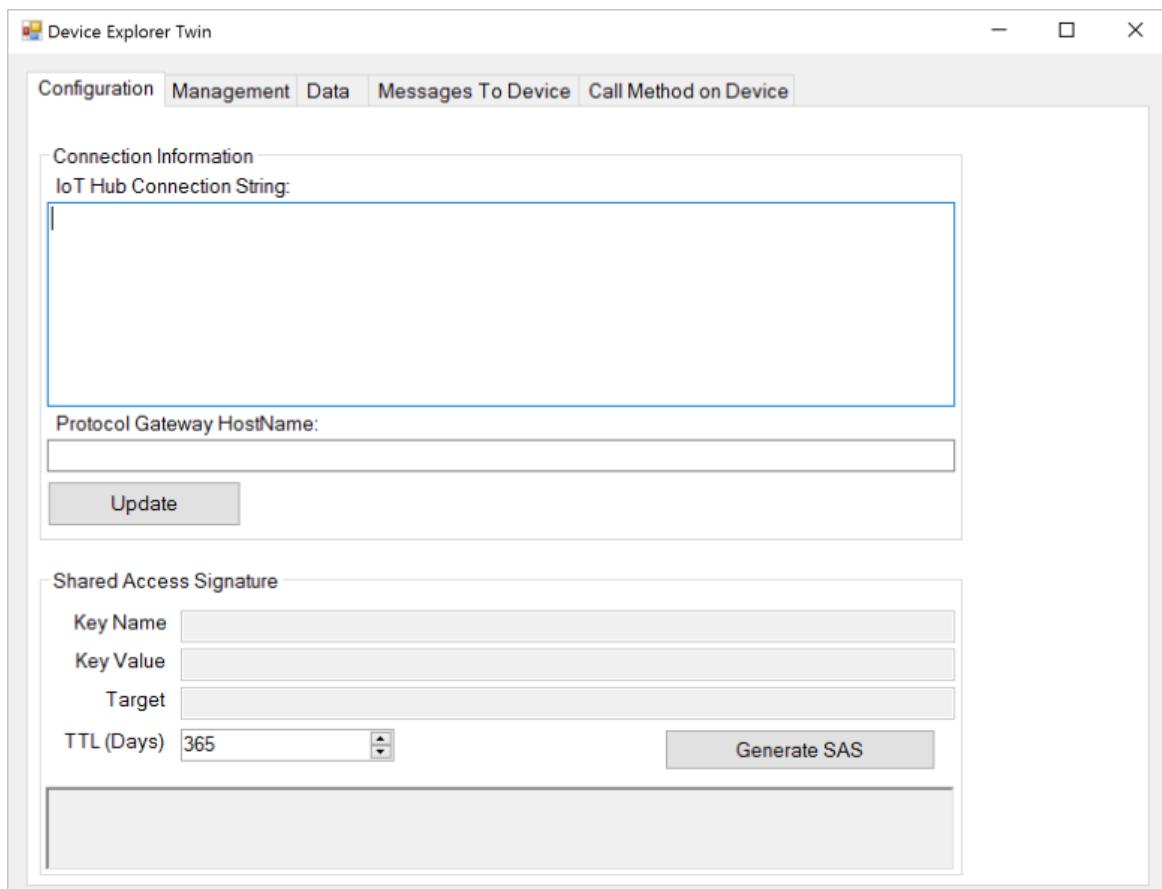
- A Windows application called [device explorer](#).
- A cross-platform Visual Studio Code extension called [Azure IoT Tools](#).
- A cross-platform Python CLI called [the IoT extension for Azure CLI](#).

This tutorial uses the graphical *device explorer* tool. You can use the *Azure IoT Tools for VS Code* if you develop in VS Code. You can also use the *the IoT extension for Azure CLI 2.0* tool if you prefer to use a CLI tool.

The device explorer tool uses the Azure IoT service libraries to perform various functions on IoT Hub, including adding devices. If you use the device explorer tool to add a device, you get a connection string for your device. You need this connection string to run the sample applications.

If you're not familiar with the device explorer tool, the following procedure describes how to use it to add a device and obtain a device connection string.

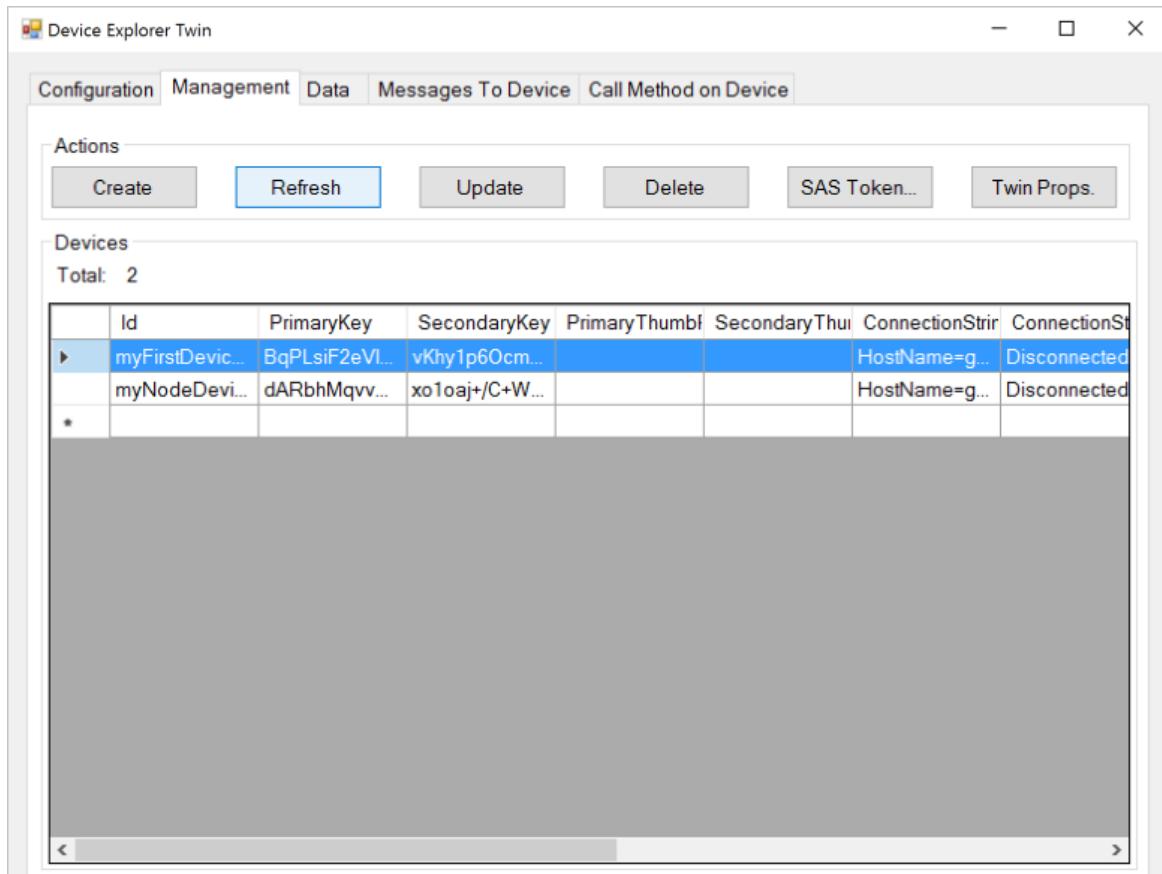
1. To install the device explorer tool, see [How to use the Device Explorer for IoT Hub devices](#).
2. When you run the program, you see this interface:



3. Enter your **IoT Hub Connection String** in the first field and click **Update**. This step configures the tool so that it can communicate with IoT Hub.

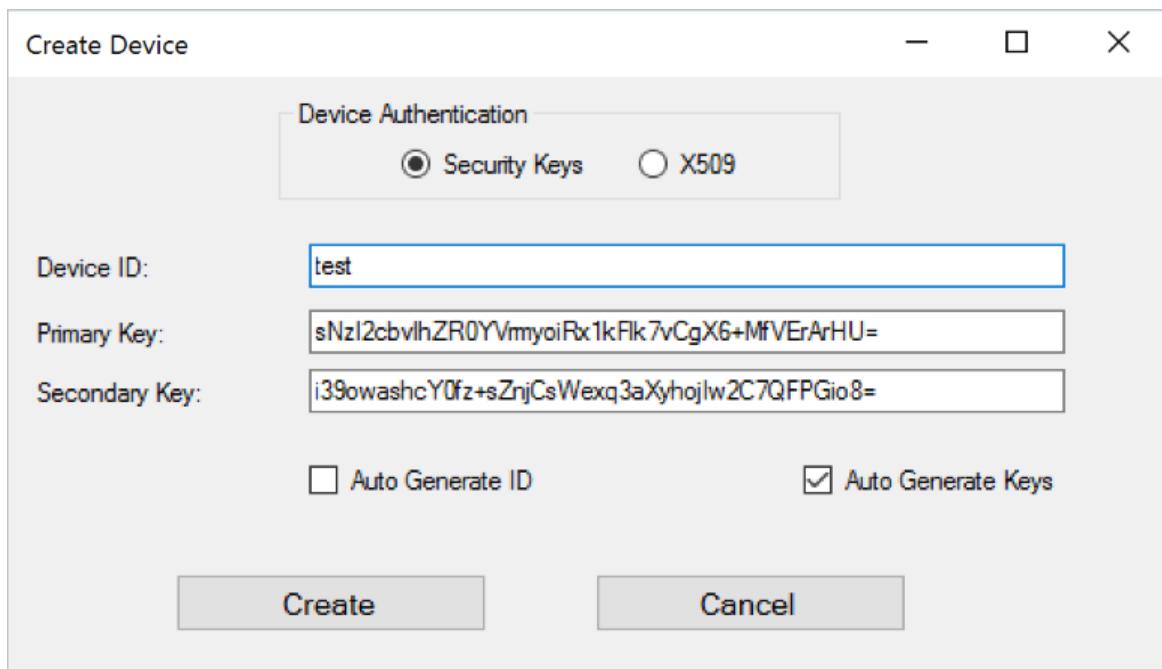
The **Connection String** can be found under **IoT Hub Service > Settings > Shared Access Policy > iothubowner**.

- When the IoT Hub connection string is configured, click the **Management** tab:

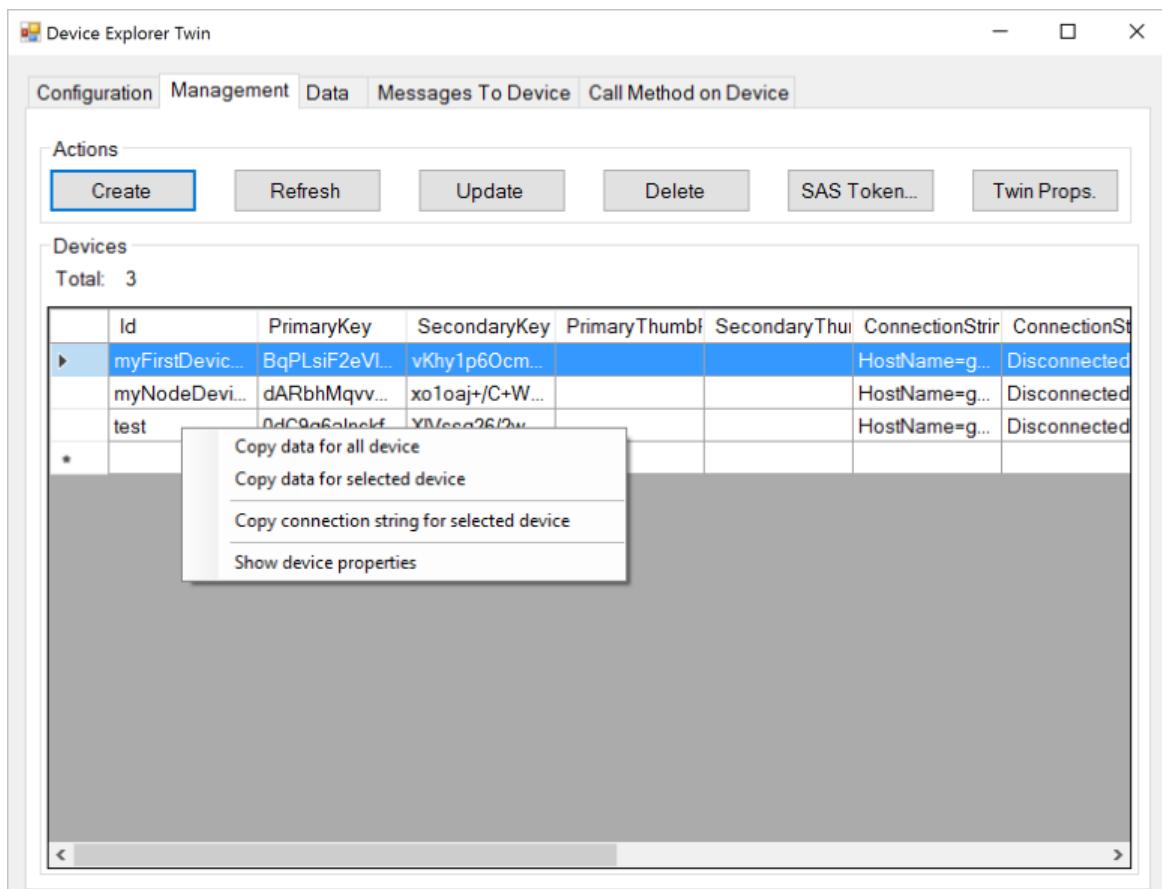


This tab is where you manage the devices registered in your IoT hub.

1. You create a device by clicking the **Create** button. A dialog displays with a set of pre-populated keys (primary and secondary). Enter a **Device ID** and then click **Create**.



2. When the device is created, the Devices list updates with all the registered devices, including the one you just created. If you right-click your new device, you see this menu:



3. If you choose **Copy connection string for selected device**, the device connection string is copied to the clipboard. Keep a copy of the device connection string. You need it when running the sample applications described in the following sections.

When you've completed the steps above, you're ready to start running some code. Most samples have a

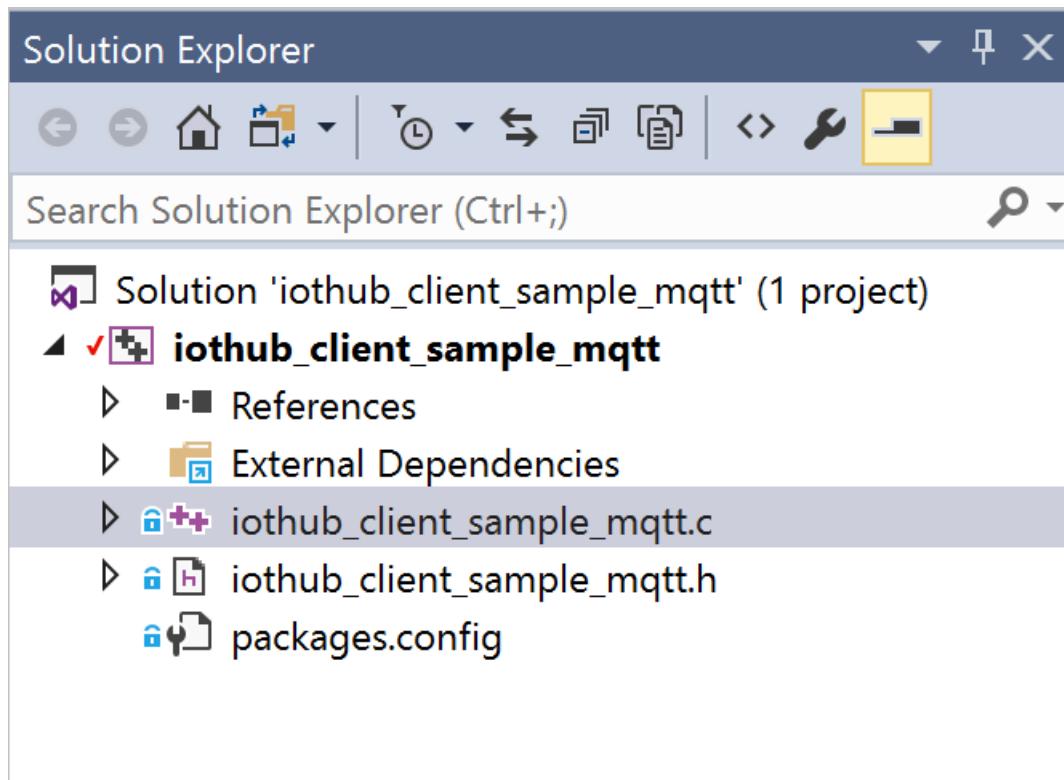
constant at the top of the main source file that enables you to enter a connection string. For example, the corresponding line from the **iothub_client_samples_iothub_convenience_sample** application appears as follows.

```
static const char* connectionString = "[device connection string]";
```

Use the IoTHubClient library

Within the **iothub_client** folder in the [azure-iot-sdk-c](#) repository, there is a **samples** folder that contains an application called **iothub_client_sample_mqtt**.

The Windows version of the **iothub_client_samples_iothub_convenience_sample** application includes the following Visual Studio solution:



NOTE

If you open this project in Visual Studio 2017, accept the prompts to retarget the project to the latest version.

This solution contains a single project. There are four NuGet packages installed in this solution:

- Microsoft.Azure.C.SharedUtility
- Microsoft.Azure.IoTHub.MqttTransport
- Microsoft.Azure.IoTHub.IoTHttpClient
- Microsoft.Azure.umqtt

You always need the **Microsoft.Azure.C.SharedUtility** package when you are working with the SDK. This sample uses the MQTT protocol, therefore you must include the **Microsoft.Azure.umqtt** and **Microsoft.Azure.IoTHub.MqttTransport** packages (there are equivalent packages for AMQP and HTTPS). Because the sample uses the **IoTHttpClient** library, you must also include the **Microsoft.Azure.IoTHub.IoTHttpClient** package in your solution.

You can find the implementation for the sample application in the

iothub_client_samples_iothub_convenience_sample source file.

The following steps use this sample application to walk you through what's required to use the **IoTHubClient** library.

Initialize the library

NOTE

Before you start working with the libraries, you may need to perform some platform-specific initialization. For example, if you plan to use AMQP on Linux you must initialize the OpenSSL library. The samples in the [GitHub repository](#) call the utility function **platform_init** when the client starts and call the **platform_deinit** function before exiting. These functions are declared in the platform.h header file. Examine the definitions of these functions for your target platform in the [repository](#) to determine whether you need to include any platform-specific initialization code in your client.

To start working with the libraries, first allocate an IoT Hub client handle:

```
if ((iotHubClientHandle =
    IoTHubClient_LL_CreateFromConnectionString(connectionString, MQTT_Protocol)) == NULL)
{
    (void)printf("ERROR: iotHubClientHandle is NULL!\r\n");
}
else
{
    ...
}
```

You pass a copy of the device connection string you obtained from the device explorer tool to this function. You also designate the communications protocol to use. This example uses MQTT, but AMQP and HTTPS are also options.

When you have a valid **IOTHUB_CLIENT_HANDLE**, you can start calling the APIs to send and receive messages to and from IoT Hub.

Send messages

The sample application sets up a loop to send messages to your IoT hub. The following snippet:

- Creates a message.
- Adds a property to the message.
- Sends a message.

First, create a message:

```

size_t iterator = 0;
do
{
    if (iterator < MESSAGE_COUNT)
    {
        sprintf_s(msgText, sizeof(msgText), "{\"deviceId\":\"myFirstDevice\", \"windSpeed\":%.2f}",
avgWindSpeed + (rand() % 4 + 2));
        if ((messages[iterator].messageHandle = IoTHubMessage_CreateFromByteArray((const unsigned
char*)msgText, strlen(msgText))) == NULL)
        {
            (void)printf("ERROR: iotHubMessageHandle is NULL!\r\n");
        }
        else
        {
            messages[iterator].messageTrackingId = iterator;
            MAP_HANDLE propMap = IoTHubMessage_Properties(messages[iterator].messageHandle);
            (void)sprintf_s(propText, sizeof(propText), "PropMsg_%zu", iterator);
            if (Map_AddOrUpdate(propMap, "PropName", propText) != MAP_OK)
            {
                (void)printf("ERROR: Map_AddOrUpdate Failed!\r\n");
            }

            if (IoTHubClient_LL_SendEventAsync(iotHubClientHandle, messages[iterator].messageHandle,
SendConfirmationCallback, &messages[iterator]) != IOTHUB_CLIENT_OK)
            {
                (void)printf("ERROR: IoTHubClient_LL_SendEventAsync.....FAILED!\r\n");
            }
            else
            {
                (void)printf("IoTHubClient_LL_SendEventAsync accepted message [%d] for transmission to IoT
Hub.\r\n", (int)iterator);
            }
        }
    }
    IoTHubClient_LL_DoWork(iotHubClientHandle);
    ThreadAPI_Sleep(1);

    iterator++;
} while (g_continueRunning);

```

Every time you send a message, you specify a reference to a callback function that's invoked when the data is sent. In this example, the callback function is called **SendConfirmationCallback**. The following snippet shows this callback function:

```

static void SendConfirmationCallback(IOTHUB_CLIENT_CONFIRMATION_RESULT result, void* userContextCallback)
{
    EVENT_INSTANCE* eventInstance = (EVENT_INSTANCE*)userContextCallback;
    (void)printf("Confirmation[%d] received for message tracking id = %zu with result = %s\r\n",
callbackCounter, eventInstance->messageTrackingId, ENUM_TO_STRING(IOTHUB_CLIENT_CONFIRMATION_RESULT,
result));
    /* Some device specific action code goes here... */
    callbackCounter++;
    IoTHubMessage_Destroy(eventInstance->messageHandle);
}

```

Note the call to the **IoTHubMessage_Destroy** function when you're done with the message. This function frees the resources allocated when you created the message.

Receive messages

Receiving a message is an asynchronous operation. First, you register the callback to invoke when the device receives a message:

```
if (IoTHubClient_LL_SetMessageCallback(iotHubClientHandle, ReceiveMessageCallback, &receiveContext) !=  
    IOTHUB_CLIENT_OK)  
{  
    (void)printf("ERROR: IoTHubClient_LL_SetMessageCallback.....FAILED!\r\n");  
}  
else  
{  
    (void)printf("IoTHubClient_LL_SetMessageCallback...successful.\r\n");  
    ...
```

The last parameter is a void pointer to whatever you want. In the sample, it's a pointer to an integer but it could be a pointer to a more complex data structure. This parameter enables the callback function to operate on shared state with the caller of this function.

When the device receives a message, the registered callback function is invoked. This callback function retrieves:

- The message id and correlation id from the message.
- The message content.
- Any custom properties from the message.

```

static IOTHUBMESSAGE_DISPOSITION_RESULT ReceiveMessageCallback(IOTHUB_MESSAGE_HANDLE message, void*
userContextCallback)
{
    int* counter = (int*)userContextCallback;
    const char* buffer;
    size_t size;
    MAP_HANDLE mapProperties;
    const char* messageId;
    const char* correlationId;

    // Message properties
    if ((messageId = IoTHubMessage_GetMessageId(message)) == NULL)
    {
        messageId = "<null>";
    }

    if ((correlationId = IoTHubMessage_GetCorrelationId(message)) == NULL)
    {
        correlationId = "<null>";
    }

    // Message content
    if (IoTHubMessage_GetByteArray(message, (const unsigned char**)&buffer, &size) != IOTHUB_MESSAGE_OK)
    {
        (void)printf("unable to retrieve the message data\r\n");
    }
    else
    {
        (void)printf("Received Message [%d]\r\n Message ID: %s\r\n Correlation ID: %s\r\n Data: <<%.*s>>
& Size=%d\r\n", *counter, messageId, correlationId, (int)size, buffer, (int)size);
        // If we receive the work 'quit' then we stop running
        if (size == (strlen("quit") * sizeof(char)) && memcmp(buffer, "quit", size) == 0)
        {
            g_continueRunning = false;
        }
    }

    // Retrieve properties from the message
    mapProperties = IoTHubMessage_Properties(message);
    if (mapProperties != NULL)
    {
        const char*const* keys;
        const char*const* values;
        size_t propertyCount = 0;
        if (Map_GetInternals(mapProperties, &keys, &values, &propertyCount) == MAP_OK)
        {
            if (propertyCount > 0)
            {
                size_t index;

                printf(" Message Properties:\r\n");
                for (index = 0; index < propertyCount; index++)
                {
                    (void)printf("\tKey: %s Value: %s\r\n", keys[index], values[index]);
                }
                (void)printf("\r\n");
            }
        }
    }

    /* Some device specific action code goes here... */
    (*counter)++;
    return IOTHUBMESSAGE_ACCEPTED;
}

```

Use the **IoTHubMessage_GetByteArray** function to retrieve the message, which in this example is a string.

Uninitialize the library

When you're done sending events and receiving messages, you can uninitialized the IoT library. To do so, issue the following function call:

```
IoTHubClient_LL_Destroy(iotHubClientHandle);
```

This call frees up the resources previously allocated by the **IoTHubClient_CreateFromConnectionString** function.

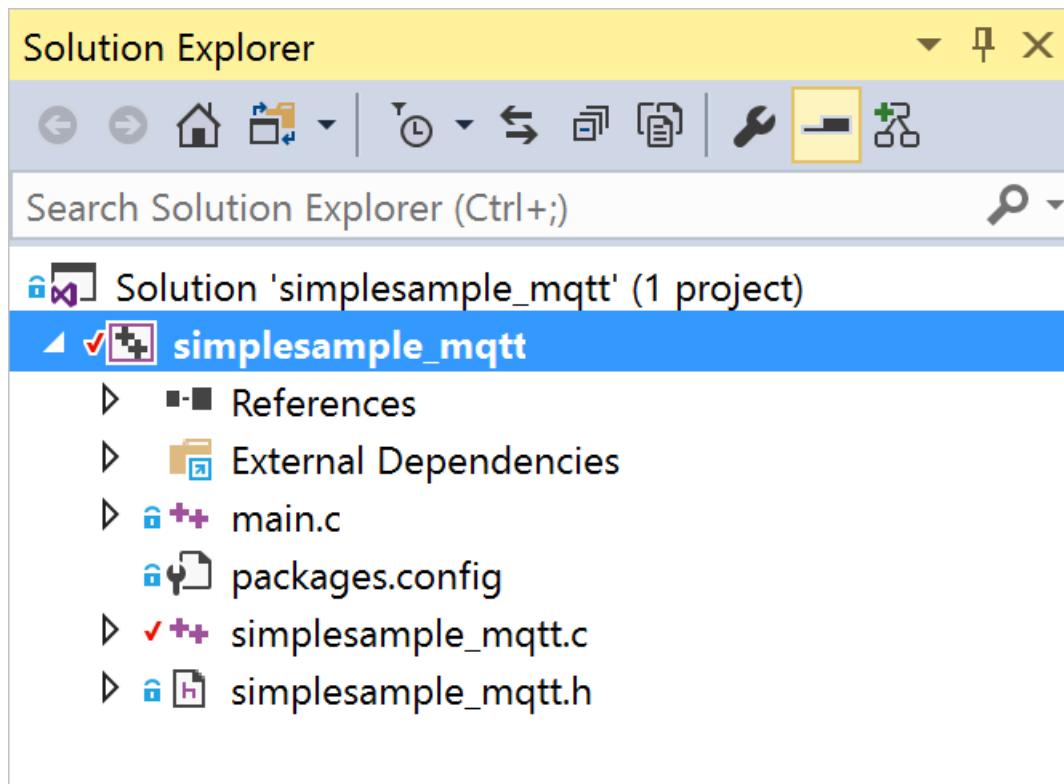
As you can see, it's easy to send and receive messages with the **IoTHubClient** library. The library handles the details of communicating with IoT Hub, including which protocol to use (from the perspective of the developer, this is a simple configuration option).

The **IoTHubClient** library also provides precise control over how to serialize the data your device sends to IoT Hub. In some cases this level of control is an advantage, but in others it is an implementation detail that you don't want to be concerned with. If that's the case, you might consider using the **serializer** library, which is described in the next section.

Use the serializer library

Conceptually the **serializer** library sits on top of the **IoTHubClient** library in the SDK. It uses the **IoTHubClient** library for the underlying communication with IoT Hub, but it adds modeling capabilities that remove the burden of dealing with message serialization from the developer. How this library works is best demonstrated by an example.

Inside the **serializer** folder in the [azure-iot-sdk-c repository](#), is a **samples** folder that contains an application called **simplesample_mqtt**. The Windows version of this sample includes the following Visual Studio solution:



NOTE

If you open this project in Visual Studio 2017, accept the prompts to retarget the project to the latest version.

As with the previous sample, this one includes several NuGet packages:

- Microsoft.Azure.C.SharedUtility
- Microsoft.Azure.IoTHub.MqttTransport
- Microsoft.Azure.IoTHub.IoTHubClient
- Microsoft.Azure.IoTHub.Serializer
- Microsoft.Azure.umqtt

You've seen most of these packages in the previous sample, but **Microsoft.Azure.IoTHub.Serializer** is new. This package is required when you use the **serializer** library.

You can find the implementation of the sample application in the `**iothub_client_samples_iothub_convenience_sample*` file.

The following sections walk you through the key parts of this sample.

Initialize the library

To start working with the **serializer** library, call the initialization APIs:

```
if (serializer_init(NULL) != SERIALIZER_OK)
{
    (void)printf("Failed on serializer_init\r\n");
}
else
{
    IOTHUB_CLIENT_LL_HANDLE iotHubClientHandle =
    IoTHubClient_LL_CreateFromConnectionString(connectionString, MQTT_Protocol);
    srand((unsigned int)time(NULL));
    int avgWindSpeed = 10;

    if (iotHubClientHandle == NULL)
    {
        (void)printf("Failed on IoTHubClient_LL_Create\r\n");
    }
    else
    {
        ContosoAnemometer* myWeather = CREATE_MODEL_INSTANCE(WeatherStation, ContosoAnemometer);
        if (myWeather == NULL)
        {
            (void)printf("Failed on CREATE_MODEL_INSTANCE\r\n");
        }
        else
        {
            ...
        }
    }
}
```

The call to the **serializer_init** function is a one-time call and initializes the underlying library. Then, you call the **IoTHubClient_LL_CreateFromConnectionString** function, which is the same API as in the **IoTHubClient** sample. This call sets your device connection string (this call is also where you choose the protocol you want to use). This sample uses MQTT as the transport, but could use AMQP or HTTPS.

Finally, call the **CREATE_MODEL_INSTANCE** function. **WeatherStation** is the namespace of the model and **ContosoAnemometer** is the name of the model. Once the model instance is created, you can use it to start sending and receiving messages. However, it's important to understand what a model is.

Define the model

A model in the **serializer** library defines the messages that your device can send to IoT Hub and the messages, called *actions* in the modeling language, which it can receive. You define a model using a set of C macros as in the **iothub_client_samples_iothub_convenience_sample** sample application:

```

BEGIN_NAMESPACE(WeatherStation);

DECLARE_MODEL(ContosoAnemometer,
WITH_DATA(ascii_char_ptr, DeviceId),
WITH_DATA(int, WindSpeed),
WITH_ACTION(TurnFanOn),
WITH_ACTION(TurnFanOff),
WITH_ACTION(SetAirResistance, int, Position)
);

END_NAMESPACE(WeatherStation);

```

The **BEGIN_NAMESPACE** and **END_NAMESPACE** macros both take the namespace of the model as an argument. It's expected that anything between these macros is the definition of your model or models, and the data structures that the models use.

In this example, there is a single model called **ContosoAnemometer**. This model defines two pieces of data that your device can send to IoT Hub: **DeviceId** and **WindSpeed**. It also defines three actions (messages) that your device can receive: **TurnFanOn**, **TurnFanOff**, and **SetAirResistance**. Each data element has a type, and each action has a name (and optionally a set of parameters).

The data and actions defined in the model define an API surface that you can use to send messages to IoT Hub, and respond to messages sent to the device. Use of this model is best understood through an example.

Send messages

The model defines the data you can send to IoT Hub. In this example, that means one of the two data items defined using the **WITH_DATA** macro. There are several steps required to send **DeviceId** and **WindSpeed** values to an IoT hub. The first is to set the data you want to send:

```

myWeather->DeviceId = "myFirstDevice";
myWeather->WindSpeed = avgWindSpeed + (rand() % 4 + 2);

```

The model you defined earlier enables you to set the values by setting members of a **struct**. Next, serialize the message you want to send:

```

unsigned char* destination;
size_t destinationSize;
if (SERIALIZE(&destination, &destinationSize, myWeather->DeviceId, myWeather->WindSpeed) != CODEFIRST_OK)
{
    (void)printf("Failed to serialize\r\n");
}
else
{
    sendMessage(iotHubClientHandle, destination, destinationSize);
    free(destination);
}

```

This code serializes the device-to-cloud to a buffer (referenced by **destination**). The code then invokes the **sendMessage** function to send the message to IoT Hub:

```

static void sendMessage(IOTHUB_CLIENT_LL_HANDLE iotHubClientHandle, const unsigned char* buffer, size_t
size)
{
    static unsigned int messageTrackingId;
    IOTHUB_MESSAGE_HANDLE messageHandle = IoTHubMessage_CreateFromByteArray(buffer, size);
    if (messageHandle == NULL)
    {
        printf("unable to create a new IoTHubMessage\r\n");
    }
    else
    {
        if (IoTHubClient_LL_SendEventAsync(iotHubClientHandle, messageHandle, sendCallback, (void*)
(uintptr_t)messageTrackingId) != IOTHUB_CLIENT_OK)
        {
            printf("failed to hand over the message to IoTHubClient");
        }
        else
        {
            printf("IoTHubClient accepted the message for delivery\r\n");
        }
        IoTHubMessage_Destroy(messageHandle);
    }
    messageTrackingId++;
}

```

The second to last parameter of **IoTHubClient_LL_SendEventAsync** is a reference to a callback function that's called when the data is successfully sent. Here's the callback function in the sample:

```

void sendCallback(IOTHUB_CLIENT_CONFIRMATION_RESULT result, void* userContextCallback)
{
    unsigned int messageTrackingId = (unsigned int)(uintptr_t)userContextCallback;

    (void)printf("Message Id: %u Received.\r\n", messageTrackingId);

    (void)printf("Result Call Back Called! Result is: %s \r\n",
ENUM_TO_STRING(IOTHUB_CLIENT_CONFIRMATION_RESULT, result));
}

```

The second parameter is a pointer to user context; the same pointer passed to **IoTHubClient_LL_SendEventAsync**. In this case, the context is a simple counter, but it can be anything you want.

That's all there is to sending device-to-cloud messages. The only thing left to cover is how to receive messages.

Receive messages

Receiving a message works similarly to the way messages work in the **IoTHubClient** library. First, you register a message callback function:

```

if (IoTHubClient_LL_SetMessageCallback(iotHubClientHandle,
    IoTHubMessage, myWeather) != IOTHUB_CLIENT_OK)
{
    printf("unable to IoTHubClient_SetMessageCallback\r\n");
}
else
{
    ...
}

```

Then, you write the callback function that's invoked when a message is received:

```

static IOTHUBMESSAGE_DISPOSITION_RESULT IoTHubMessage(IOTHUB_MESSAGE_HANDLE message, void*
userContextCallback)
{
    IOTHUBMESSAGE_DISPOSITION_RESULT result;
    const unsigned char* buffer;
    size_t size;
    if (IoTHubMessage_GetByteArray(message, &buffer, &size) != IOTHUB_MESSAGE_OK)
    {
        printf("unable to IoTHubMessage_GetByteArray\r\n");
        result = IOTHUBMESSAGE_ABANDONED;
    }
    else
    {
        /*buffer is not zero terminated*/
        char* temp = malloc(size + 1);
        if (temp == NULL)
        {
            printf("failed to malloc\r\n");
            result = IOTHUBMESSAGE_ABANDONED;
        }
        else
        {
            (void)memcpy(temp, buffer, size);
            temp[size] = '\0';
            EXECUTE_COMMAND_RESULT executeCommandResult = EXECUTE_COMMAND(userContextCallback, temp);
            result =
                (executeCommandResult == EXECUTE_COMMAND_ERROR) ? IOTHUBMESSAGE_ABANDONED :
                (executeCommandResult == EXECUTE_COMMAND_SUCCESS) ? IOTHUBMESSAGE_ACCEPTED :
                IOTHUBMESSAGE_REJECTED;
            free(temp);
        }
    }
    return result;
}

```

This code is boilerplate -- it's the same for any solution. This function receives the message and takes care of routing it to the appropriate function through the call to **EXECUTE_COMMAND**. The function called at this point depends on the definition of the actions in your model.

When you define an action in your model, you're required to implement a function that's called when your device receives the corresponding message. For example, if your model defines this action:

```
WITH_ACTION(SetAirResistance, int, Position)
```

Define a function with this signature:

```

EXECUTE_COMMAND_RESULT SetAirResistance(ContosoAnemometer* device, int Position)
{
    (void)device;
    (void)printf("Setting Air Resistance Position to %d.\r\n", Position);
    return EXECUTE_COMMAND_SUCCESS;
}

```

Note how the name of the function matches the name of the action in the model and that the parameters of the function match the parameters specified for the action. The first parameter is always required and contains a pointer to the instance of your model.

When the device receives a message that matches this signature, the corresponding function is called. Therefore, aside from having to include the boilerplate code from **IoTHubMessage**, receiving messages is just a matter of defining a simple function for each action defined in your model.

Uninitialize the library

When you're done sending data and receiving messages, you can uninitialized the IoT library:

```
...
    DESTROY_MODEL_INSTANCE(myWeather);
}
IoTHubClient_LL_Destroy(iotHubClientHandle);
}
serializer_deinit();
```

Each of these three functions aligns with the three initialization functions described previously. Calling these APIs ensures that you free previously allocated resources.

Next Steps

This article covered the basics of using the libraries in the **Azure IoT device SDK for C**. It provided you with enough information to understand what's included in the SDK, its architecture, and how to get started working with the Windows samples. The next article continues the description of the SDK by explaining [more about the IoTHubClient library](#).

To learn more about developing for IoT Hub, see the [Azure IoT SDKs](#).

To further explore the capabilities of IoT Hub, see:

- [Deploying AI to edge devices with Azure IoT Edge](#)

Azure IoT device SDK for C – more about IoTHubClient

10/24/2018 • 13 minutes to read

Azure IoT device SDK for C is the first article in this series introducing the **Azure IoT device SDK for C**. That article explained that there are two architectural layers in SDK. At the base is the **IoTHubClient** library that directly manages communication with IoT Hub. There's also the **serializer** library that builds on top of that to provide serialization services. In this article, we'll provide additional detail on the **IoTHubClient** library.

NOTE

Some of the features mentioned in this article, like cloud-to-device messaging, device twins, and device management, are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

The previous article described how to use the **IoTHubClient** library to send events to IoT Hub and receive messages. This article extends that discussion by explaining how to more precisely manage *when* you send and receive data, introducing you to the **lower-level APIs**. We'll also explain how to attach properties to events (and retrieve them from messages) using the property handling features in the **IoTHubClient** library. Finally, we'll provide additional explanation of different ways to handle messages received from IoT Hub.

The article concludes by covering a couple of miscellaneous topics, including more about device credentials and how to change the behavior of the **IoTHubClient** through configuration options.

We'll use the **IoTHubClient** SDK samples to explain these topics. If you want to follow along, see the **iothub_client_sample_http** and **iothub_client_sample_amqp** applications that are included in the Azure IoT device SDK for C. Everything described in the following sections is demonstrated in these samples.

You can find the [Azure IoT device SDK for C](#) GitHub repository and view details of the API in the [C API reference](#).

The lower-level APIs

The previous article described the basic operation of the **IoTHubClient** within the context of the **iothub_client_sample_amqp** application. For example, it explained how to initialize the library using this code.

```
IOTHUB_CLIENT_HANDLE iotHubClientHandle;  
iotHubClientHandle = IoTHubClient_CreateFromConnectionString(connectionString, AMQP_Protocol);
```

It also described how to send events using this function call.

```
IoTHubClient_SendEventAsync(iotHubClientHandle, message.messageHandle, SendConfirmationCallback, &message);
```

The article also described how to receive messages by registering a callback function.

```
int receiveContext = 0;  
IoTHubClient_SetMessageCallback(iotHubClientHandle, ReceiveMessageCallback, &receiveContext);
```

The article also showed how to free resources using code such as the following.

```
IoTHubClient_Destroy(iotHubClientHandle);
```

There are companion functions for each of these APIs:

- `IoTHubClient_LL_CreateFromConnectionString`
- `IoTHubClient_LL_SendEventAsync`
- `IoTHubClient_LL_SetMessageCallback`
- `IoTHubClient_LL_Destroy`

These functions all include **LL** in the API name. Other the **LL** part of the name, the parameters of each of these functions are identical to their non-LL counterparts. However, the behavior of these functions is different in one important way.

When you call **IoTHubClient_CreateFromConnectionString**, the underlying libraries create a new thread that runs in the background. This thread sends events to, and receives messages from, IoT Hub. No such thread is created when working with the **LL** APIs. The creation of the background thread is a convenience to the developer. You don't have to worry about explicitly sending events and receiving messages from IoT Hub -- it happens automatically in the background. In contrast, the **LL** APIs give you explicit control over communication with IoT Hub, if you need it.

To understand this concept better, let's look at an example:

When you call **IoTHubClient_SendEventAsync**, what you're actually doing is putting the event in a buffer. The background thread created when you call **IoTHubClient_CreateFromConnectionString** continually monitors this buffer and sends any data that it contains to IoT Hub. This happens in the background at the same time that the main thread is performing other work.

Similarly, when you register a callback function for messages using **IoTHubClient_SetMessageCallback**, you're instructing the SDK to have the background thread invoke the callback function when a message is received, independent of the main thread.

The **LL** APIs don't create a background thread. Instead, a new API must be called to explicitly send and receive data from IoT Hub. This is demonstrated in the following example.

The **iothub_client_sample_http** application that's included in the SDK demonstrates the lower-level APIs. In that sample, we send events to IoT Hub with code such as the following:

```
EVENT_INSTANCE message;
sprintf_s(msgText, sizeof(msgText), "Message_%d_From_IoTHubClient_LL_Over_HTTP", i);
message.messageHandle = IoTHubMessage_CreateFromByteArray((const unsigned char*)msgText, strlen(msgText));

IoTHubClient_LL_SendEventAsync(iotHubClientHandle, message.messageHandle, SendConfirmationCallback, &message)
```

The first three lines create the message, and the last line sends the event. However, as mentioned previously, sending the event means that the data is simply placed in a buffer. Nothing is transmitted on the network when we call **IoTHubClient_LL_SendEventAsync**. In order to actually ingress the data to IoT Hub, you must call **IoTHubClient_LL_DoWork**, as in this example:

```
while (1)
{
    IoTHubClient_LL_DoWork(iotHubClientHandle);
    ThreadAPI_Sleep(100);
}
```

This code (from the **iothub_client_sample_http** application) repeatedly calls **IoTHubClient_LL_DoWork**. Each time **IoTHubClient_LL_DoWork** is called, it sends some events from the buffer to IoT Hub and it retrieves a queued message being sent to the device. The latter case means that if we registered a callback function for messages, then the callback is invoked (assuming any messages are queued up). We would have registered such a callback function with code such as the following:

```
IoTHubClient_LL_SetMessageCallback(iotHubClientHandle, ReceiveMessageCallback, &receiveContext)
```

The reason that **IoTHubClient_LL_DoWork** is often called in a loop is that each time it's called, it sends *some* buffered events to IoT Hub and retrieves *the next* message queued up for the device. Each call isn't guaranteed to send all buffered events or to retrieve all queued messages. If you want to send all events in the buffer and then continue on with other processing you can replace this loop with code such as the following:

```
IOTHUB_CLIENT_STATUS status;

while ((IoTHubClient_LL_GetSendStatus(iotHubClientHandle, &status) == IOTHUB_CLIENT_OK) && (status == IOTHUB_CLIENT_SEND_STATUS_BUSY))
{
    IoTHubClient_LL_DoWork(iotHubClientHandle);
    ThreadAPI_Sleep(100);
}
```

This code calls **IoTHubClient_LL_DoWork** until all events in the buffer have been sent to IoT Hub. Note this does not also imply that all queued messages have been received. Part of the reason for this is that checking for "all" messages isn't as deterministic an action. What happens if you retrieve "all" of the messages, but then another one is sent to the device immediately after? A better way to deal with that is with a programmed timeout. For example, the message callback function could reset a timer every time it's invoked. You can then write logic to continue processing if, for example, no messages have been received in the last X seconds.

When you're finished ingressing events and receiving messages, be sure to call the corresponding function to clean up resources.

```
IoTHubClient_LL_Destroy(iotHubClientHandle);
```

Basically there's only one set of APIs to send and receive data with a background thread and another set of APIs that does the same thing without the background thread. A lot of developers may prefer the non-LL APIs, but the lower-level APIs are useful when the developer wants explicit control over network transmissions. For example, some devices collect data over time and only ingress events at specified intervals (for example, once an hour or once a day). The lower-level APIs give you the ability to explicitly control when you send and receive data from IoT Hub. Others will simply prefer the simplicity that the lower-level APIs provide. Everything happens on the main thread rather than some work happening in the background.

Whichever model you choose, be sure to be consistent in which APIs you use. If you start by calling **IoTHubClient_LL_CreateFromConnectionString**, be sure you only use the corresponding lower-level APIs for any follow-up work:

- **IoTHubClient_LL_SendEventAsync**
- **IoTHubClient_LL_SetMessageCallback**
- **IoTHubClient_LL_Destroy**
- **IoTHubClient_LL_DoWork**

The opposite is true as well. If you start with **IoTHubClient_CreateFromConnectionString**, then use the non-LL APIs for any additional processing.

In the Azure IoT device SDK for C, see the **iothub_client_sample_http** application for a complete example of the lower-level APIs. The **iothub_client_sample_amqp** application can be referenced for a full example of the non-LL APIs.

Property handling

So far when we've described sending data, we've been referring to the body of the message. For example, consider this code:

```
EVENT_INSTANCE message;
sprintf_s(msgText, sizeof(msgText), "Hello World");
message.messageHandle = IoTHubMessage_CreateFromByteArray((const unsigned char*)msgText, strlen(msgText));
IoTHubClient_LL_SendEventAsync(iotHubClientHandle, message.messageHandle, SendConfirmationCallback, &message)
```

This example sends a message to IoT Hub with the text "Hello World." However, IoT Hub also allows properties to be attached to each message. Properties are name/value pairs that can be attached to the message. For example, we can modify the previous code to attach a property to the message:

```
MAP_HANDLE propMap = IoTHubMessage.Properties(message.messageHandle);
sprintf_s(propText, sizeof(propText), "%d", i);
Map_AddOrUpdate(propMap, "SequenceNumber", propText);
```

We start by calling **IoTHubMessage.Properties** and passing it the handle of our message. What we get back is a **MAP_HANDLE** reference that enables us to start adding properties. The latter is accomplished by calling **Map.AddOrUpdate**, which takes a reference to a MAP_HANDLE, the property name, and the property value. With this API we can add as many properties as we like.

When the event is read from **Event Hubs**, the receiver can enumerate the properties and retrieve their corresponding values. For example, in .NET this would be accomplished by accessing the [Properties collection on the EventData object](#).

In the previous example, we're attaching properties to an event that we send to IoT Hub. Properties can also be attached to messages received from IoT Hub. If we want to retrieve properties from a message, we can use code such as the following in our message callback function:

```

static IOTHUBMESSAGE_DISPOSITION_RESULT ReceiveMessageCallback(IOTHUB_MESSAGE_HANDLE message, void*
userContextCallback)
{
    . . .

    // Retrieve properties from the message
    MAP_HANDLE mapProperties = IoTHubMessage_Properties(message);
    if (mapProperties != NULL)
    {
        const char*const* keys;
        const char*const* values;
        size_t propertyCount = 0;
        if (Map_GetInternals(mapProperties, &keys, &values, &propertyCount) == MAP_OK)
        {
            if (propertyCount > 0)
            {
                printf("Message Properties:\r\n");
                for (size_t index = 0; index < propertyCount; index++)
                {
                    printf("\tKey: %s Value: %s\r\n", keys[index], values[index]);
                }
                printf("\r\n");
            }
        }
    }
    . . .
}

```

The call to **IoTHubMessage_Properties** returns the **MAP_HANDLE** reference. We then pass that reference to **Map_GetInternals** to obtain a reference to an array of the name/value pairs (as well as a count of the properties). At that point it's a simple matter of enumerating the properties to get to the values we want.

You don't have to use properties in your application. However, if you need to set them on events or retrieve them from messages, the **IoTHubClient** library makes it easy.

Message handling

As stated previously, when messages arrive from IoT Hub the **IoTHubClient** library responds by invoking a registered callback function. There is a return parameter of this function that deserves some additional explanation. Here's an excerpt of the callback function in the **iothub_client_sample_http** sample application:

```

static IOTHUBMESSAGE_DISPOSITION_RESULT ReceiveMessageCallback(IOTHUB_MESSAGE_HANDLE message, void*
userContextCallback)
{
    . . .
    return IOTHUBMESSAGE_ACCEPTED;
}

```

Note that the return type is **IOTHUBMESSAGE_DISPOSITION_RESULT** and in this particular case we return **IOTHUBMESSAGE_ACCEPTED**. There are other values we can return from this function that change how the **IoTHubClient** library reacts to the message callback. Here are the options.

- **IOTHUBMESSAGE_ACCEPTED** – The message has been processed successfully. The **IoTHubClient** library will not invoke the callback function again with the same message.
- **IOTHUBMESSAGE_REJECTED** – The message was not processed and there is no desire to do so in the future. The **IoTHubClient** library should not invoke the callback function again with the same message.
- **IOTHUBMESSAGE_ABANDONED** – The message was not processed successfully, but the

IoTHubClient library should invoke the callback function again with the same message.

For the first two return codes, the **IoTHubClient** library sends a message to IoT Hub indicating that the message should be deleted from the device queue and not delivered again. The net effect is the same (the message is deleted from the device queue), but whether the message was accepted or rejected is still recorded. Recording this distinction is useful to senders of the message who can listen for feedback and find out if a device has accepted or rejected a particular message.

In the last case a message is also sent to IoT Hub, but it indicates that the message should be redelivered. Typically you'll abandon a message if you encounter some error but want to try to process the message again. In contrast, rejecting a message is appropriate when you encounter an unrecoverable error (or if you simply decide you don't want to process the message).

In any case, be aware of the different return codes so that you can elicit the behavior you want from the **IoTHubClient** library.

Alternate device credentials

As explained previously, the first thing to do when working with the **IoTHubClient** library is to obtain a **IOTHUB_CLIENT_HANDLE** with a call such as the following:

```
IOTHUB_CLIENT_HANDLE iotHubClientHandle;
iotHubClientHandle = IoTHubClient_CreateFromConnectionString(connectionString, AMQP_Protocol);
```

The arguments to **IoTHubClient_CreateFromConnectionString** are the device connection string and a parameter that indicates the protocol we use to communicate with IoT Hub. The device connection string has a format that appears as follows:

```
HostName=IOTHUBNAME.IOTHUBSUFFIX;DeviceId=DEVICEID;SharedAccessKey=SHAREDACCESSKEY
```

There are four pieces of information in this string: IoT Hub name, IoT Hub suffix, device ID, and shared access key. You obtain the fully qualified domain name (FQDN) of an IoT hub when you create your IoT hub instance in the Azure portal — this gives you the IoT hub name (the first part of the FQDN) and the IoT hub suffix (the rest of the FQDN). You get the device ID and the shared access key when you register your device with IoT Hub (as described in the [previous article](#)).

IoTHubClient_CreateFromConnectionString gives you one way to initialize the library. If you prefer, you can create a new **IOTHUB_CLIENT_HANDLE** by using these individual parameters rather than the device connection string. This is achieved with the following code:

```
IOTHUB_CLIENT_CONFIG iotHubClientConfig;
iotHubClientConfig.iotHubName = "";
iotHubClientConfig.deviceId = "";
iotHubClientConfig.deviceKey = "";
iotHubClientConfig.iotHubSuffix = "";
iotHubClientConfig.protocol = HTTP_Protocol;
IOTHUB_CLIENT_HANDLE iotHubClientHandle = IoTHubClient_LL_Create(&iotHubClientConfig);
```

This accomplishes the same thing as **IoTHubClient_CreateFromConnectionString**.

It may seem obvious that you would want to use **IoTHubClient_CreateFromConnectionString** rather than this more verbose method of initialization. Keep in mind, however, that when you register a device in IoT Hub what you get is a device ID and device key (not a connection string). The *device explorer* SDK tool introduced in the [previous article](#) uses libraries in the **Azure IoT service SDK** to create the device connection string from the device ID, device key, and IoT Hub host name. So calling **IoTHubClient_LL_Create** may be preferable because it

saves you the step of generating a connection string. Use whichever method is convenient.

Configuration options

So far everything described about the way the **IoTHubClient** library works reflects its default behavior. However, there are a few options that you can set to change how the library works. This is accomplished by leveraging the **IoTHubClient_LL_SetOption** API. Consider this example:

```
unsigned int timeout = 30000;
IoTHubClient_LL_SetOption(iotHubClientHandle, "timeout", &timeout);
```

There are a couple of options that are commonly used:

- **SetBatching** (bool) – If **true**, then data sent to IoT Hub is sent in batches. If **false**, then messages are sent individually. The default is **false**. Note that the **SetBatching** option only applies to the HTTPS protocol and not to the MQTT or AMQP protocols.
- **Timeout** (unsigned int) – This value is represented in milliseconds. If sending an HTTPS request or receiving a response takes longer than this time, then the connection times out.

The batching option is important. By default, the library ingresses events individually (a single event is whatever you pass to **IoTHubClient_LL_SendEventAsync**). If the batching option is **true**, the library collects as many events as it can from the buffer (up to the maximum message size that IoT Hub will accept). The event batch is sent to IoT Hub in a single HTTPS call (the individual events are bundled into a JSON array). Enabling batching typically results in big performance gains since you're reducing network round-trips. It also significantly reduces bandwidth since you are sending one set of HTTPS headers with an event batch rather than a set of headers for each individual event. Unless you have a specific reason to do otherwise, typically you'll want to enable batching.

Next steps

This article describes in detail the behavior of the **IoTHubClient** library found in the [Azure IoT device SDK for C](#). With this information, you should have a good understanding of the capabilities of the **IoTHubClient** library. The second article in this series is [Azure IoT device SDK for C - Serializer](#), which provides similar detail on the **serializer** library.

To learn more about developing for IoT Hub, see the [Azure IoT SDKs](#).

To further explore the capabilities of IoT Hub, see [Deploying AI to edge devices with Azure IoT Edge](#).

Azure IoT device SDK for C – more about serializer

10/24/2018 • 21 minutes to read

The first article in this series introduced the [Introduction to Azure IoT device SDK for C](#). The next article provided a more detailed description of the [Azure IoT device SDK for C -- IoTHubClient](#). This article completes coverage of the SDK by providing a more detailed description of the remaining component: the **serializer** library.

NOTE

Some of the features mentioned in this article, like cloud-to-device messaging, device twins, and device management, are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

The introductory article described how to use the **serializer** library to send events to and receive messages from IoT Hub. In this article, we extend that discussion by providing a more complete explanation of how to model your data with the **serializer** macro language. The article also includes more detail about how the library serializes messages (and in some cases how you can control the serialization behavior). We'll also describe some parameters you can modify that determine the size of the models you create.

Finally, the article revisits some topics covered in previous articles such as message and property handling. As we'll find out, those features work in the same way using the **serializer** library as they do with the **IoTHubClient** library.

Everything described in this article is based on the **serializer** SDK samples. If you want to follow along, see the **simplesample_amqp** and **simplesample_http** applications included in the Azure IoT device SDK for C.

You can find the [Azure IoT device SDK for C](#) GitHub repository and view details of the API in the [C API reference](#).

The modeling language

The [Azure IoT device SDK for C](#) article in this series introduced the **Azure IoT device SDK for C** modeling language through the example provided in the **simplesample_amqp** application:

```
BEGIN_NAMESPACE(WeatherStation);

DECLARE_MODEL(ContosoAnemometer,
WITH_DATA(ascii_char_ptr, DeviceId),
WITH_DATA(double, WindSpeed),
WITH_ACTION(TurnFanOn),
WITH_ACTION(TurnFanOff),
WITH_ACTION(SetAirResistance, int, Position)
);

END_NAMESPACE(WeatherStation);
```

As you can see, the modeling language is based on C macros. You always begin your definition with **BEGIN_NAMESPACE** and always end with **END_NAMESPACE**. It's common to name the namespace for your company or, as in this example, the project that you're working on.

What goes inside the namespace are model definitions. In this case, there is a single model for an anemometer. Once again, the model can be named anything, but typically the model is named for the device or type of data you

want to exchange with IoT Hub.

Models contain a definition of the events you can ingress to IoT Hub (the *data*) as well as the messages you can receive from IoT Hub (the *actions*). As you can see from the example, events have a type and a name; actions have a name and optional parameters (each with a type).

What's not demonstrated in this sample are additional data types that are supported by the SDK. We'll cover that next.

NOTE

IoT Hub refers to the data a device sends to it as *events*, while the modeling language refers to it as *data* (defined using **WITH_DATA**). Likewise, IoT Hub refers to the data you send to devices as *messages*, while the modeling language refers to it as *actions* (defined using **WITH_ACTION**). Be aware that these terms may be used interchangeably in this article.

Supported data types

The following data types are supported in models created with the **serializer** library:

TYPE	DESCRIPTION
double	double precision floating point number
int	32 bit integer
float	single precision floating point number
long	long integer
int8_t	8 bit integer
int16_t	16 bit integer
int32_t	32 bit integer
int64_t	64 bit integer
bool	boolean
ascii_char_ptr	ASCII string
EDM_DATE_TIME_OFFSET	date time offset
EDM_GUID	GUID
EDM_BINARY	binary
DECLARE_STRUCT	complex data type

Let's start with the last data type. The **DECLARE_STRUCT** allows you to define complex data types, which are groupings of the other primitive types. These groupings allow us to define a model that looks like this:

```

DECLARE_STRUCT(TestType,
double, aDouble,
int, aInt,
float, aFloat,
long, aLong,
int8_t, aInt8,
uint8_t, auInt8,
int16_t, aInt16,
int32_t, aInt32,
int64_t, aInt64,
bool, aBool,
ascii_char_ptr, aAsciiCharPtr,
EDM_DATE_TIME_OFFSET, aDateTimeOffset,
EDM_GUID, aGuid,
EDM_BINARY, aBinary
);

DECLARE_MODEL(TestModel,
WITH_DATA(TestType, Test)
);

```

Our model contains a single data event of type **TestType**. **TestType** is a complex type that includes several members, which collectively demonstrate the primitive types supported by the **serializer** modeling language.

With a model like this, we can write code to send data to IoT Hub that appears as follows:

```

TestModel* testModel = CREATE_MODEL_INSTANCE(MyThermostat, TestModel);

testModel->Test.aDouble = 1.1;
testModel->Test.aInt = 2;
testModel->Test.aFloat = 3.0f;
testModel->Test.aLong = 4;
testModel->Test.aInt8 = 5;
testModel->Test.auInt8 = 6;
testModel->Test.aInt16 = 7;
testModel->Test.aInt32 = 8;
testModel->Test.aInt64 = 9;
testModel->Test.aBool = true;
testModel->Test.aAsciiCharPtr = "ascii string 1";

time_t now;
time(&now);
testModel->Test.aDateTimeOffset = GetDateTimeOffset(now);

EDM_GUID guid = { { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E,
0x0F } };
testModel->Test.aGuid = guid;

unsigned char binaryArray[3] = { 0x01, 0x02, 0x03 };
EDM_BINARY binaryData = { sizeof(binaryArray), &binaryArray };
testModel->Test.aBinary = binaryData;

SendAsync(iotHubClientHandle, (const void*)&(testModel->Test));

```

Basically, we're assigning a value to every member of the **Test** structure and then calling **SendAsync** to send the **Test** data event to the cloud. **SendAsync** is a helper function that sends a single data event to IoT Hub:

```

void SendAsync(IOTHUB_CLIENT_LL_HANDLE iotHubClientHandle, const void *dataEvent)
{
    unsigned char* destination;
    size_t destinationSize;
    if (SERIALIZE(&destination, &destinationSize, *(const unsigned char*)dataEvent) ==
    {
        // null terminate the string
        char* destinationAsString = (char*)malloc(destinationSize + 1);
        if (destinationAsString != NULL)
        {
            memcpy(destinationAsString, destination, destinationSize);
            destinationAsString[destinationSize] = '\0';
            IOTHUB_MESSAGE_HANDLE messageHandle = IoTHubMessage_CreateFromString(destinationAsString);
            if (messageHandle != NULL)
            {
                IoTHubClient_SendEventAsync(iotHubClientHandle, messageHandle, sendCallback, (void*)0);

                IoTHubMessage_Destroy(messageHandle);
            }
            free(destinationAsString);
        }
        free(destination);
    }
}

```

This function serializes the given data event and sends it to IoT Hub using **IoTHubClient_SendEventAsync**. This is the same code discussed in previous articles (**SendAsync** encapsulates the logic into a convenient function).

One other helper function used in the previous code is **GetDateTimeOffset**. This function transforms the given time into a value of type **EDM_DATE_TIME_OFFSET**:

```

EDM_DATE_TIME_OFFSET GetDateTimeOffset(time_t time)
{
    struct tm newTime;
    gmtime_s(&newTime, &time);
    EDM_DATE_TIME_OFFSET dateOffset;
    dateOffset.dateTime = newTime;
    dateOffset.fractionalSecond = 0;
    dateOffset.hasFractionalSecond = 0;
    dateOffset.hasTimeZone = 0;
    dateOffset.timeZoneHour = 0;
    dateOffset.timeZoneMinute = 0;
    return dateOffset;
}

```

If you run this code, the following message is sent to IoT Hub:

```
{"aDouble":1.100000000000000, "aInt":2, "aFloat":3.00000, "aLong":4, "aInt8":5, "auInt8":6, "aInt16":7,
 "aInt32":8, "aInt64":9, "aBool":true, "aAsciiCharPtr":"ascii string 1", "aDateTimeOffset":"2015-09-
 14T21:18:21Z", "aGuid":"00010203-0405-0607-0809-0A0B0C0D0E0F", "aBinary":"AQID"}
```

Note that the serialization is in JSON, which is the format generated by the **serializer** library. Also note that each member of the serialized JSON object matches the members of the **TestType** that we defined in our model. The values also exactly match those used in the code. However, note that the binary data is base64-encoded: "AQID" is the base64 encoding of {0x01, 0x02, 0x03}.

This example demonstrates the advantage of using the **serializer** library -- it enables us to send JSON to the cloud, without having to explicitly deal with serialization in our application. All we have to worry about is setting the values of the data events in our model and then calling simple APIs to send those events to the cloud.

With this information, we can define models that include the range of supported data types, including complex types (we could even include complex types within other complex types). However, the serialized JSON generated by the example above brings up an important point. *How* we send data with the **serializer** library determines exactly how the JSON is formed. That particular point is what we'll cover next.

More about serialization

The previous section highlights an example of the output generated by the **serializer** library. In this section, we'll explain how the library serializes data and how you can control that behavior using the serialization APIs.

In order to advance the discussion on serialization, we'll work with a new model based on a thermostat. First, let's provide some background on the scenario we're trying to address.

We want to model a thermostat that measures temperature and humidity. Each piece of data is going to be sent to IoT Hub differently. By default, the thermostat ingresses a temperature event once every 2 minutes; a humidity event is ingressed once every 15 minutes. When either event is ingressed, it must include a timestamp that shows the time that the corresponding temperature or humidity was measured.

Given this scenario, we'll demonstrate two different ways to model the data, and we'll explain the effect that modeling has on the serialized output.

Model 1

Here's the first version of a model that supports the previous scenario:

```
BEGIN_NAMESPACE(Contoso);

DECLARE_STRUCT(TemperatureEvent,
int, Temperature,
EDM_DATE_TIME_OFFSET, Time);

DECLARE_STRUCT(HumidityEvent,
int, Humidity,
EDM_DATE_TIME_OFFSET, Time);

DECLARE_MODEL(Thermostat,
WITH_DATA(TemperatureEvent, Temperature),
WITH_DATA(HumidityEvent, Humidity)
);

END_NAMESPACE(Contoso);
```

Note that the model includes two data events: **Temperature** and **Humidity**. Unlike previous examples, the type of each event is a structure defined using **DECLARE_STRUCT**. **TemperatureEvent** includes a temperature measurement and a timestamp; **HumidityEvent** contains a humidity measurement and a timestamp. This model gives us a natural way to model the data for the scenario described above. When we send an event to the cloud, we'll either send a temperature/timestamp or a humidity/timestamp pair.

We can send a temperature event to the cloud using code such as the following:

```

time_t now;
time(&now);
thermostat->Temperature.Temperature = 75;
thermostat->Temperature.Time = GetDateTimeOffset(now);

unsigned char* destination;
size_t destinationSize;
if (SERIALIZE(&destination, &destinationSize, thermostat->Temperature) == IOT_AGENT_OK)
{
    sendMessage(iotHubClientHandle, destination, destinationSize);
}

```

We'll use hard-coded values for temperature and humidity in the sample code, but imagine that we're actually retrieving these values by sampling the corresponding sensors on the thermostat.

The code above uses the **GetDateTimeOffset** helper that was introduced previously. For reasons that will become clear later, this code explicitly separates the task of serializing and sending the event. The previous code serializes the temperature event into a buffer. Then, **sendMessage** is a helper function (included in **simplesample_amqp**) that sends the event to IoT Hub:

```

static void sendMessage(IOTHUB_CLIENT_HANDLE iotHubClientHandle, const unsigned char* buffer, size_t size)
{
    static unsigned int messageTrackingId;
    IOTHUB_MESSAGE_HANDLE messageHandle = IoTHubMessage_CreateFromByteArray(buffer, size);
    if (messageHandle != NULL)
    {
        IoTHubClient_SendEventAsync(iotHubClientHandle, messageHandle, sendCallback, (void*)
(uintptr_t)messageTrackingId);

        IoTHubMessage_Destroy(messageHandle);
    }
    free((void*)buffer);
}

```

This code is a subset of the **SendAsync** helper described in the previous section, so we won't go over it again here.

When we run the previous code to send the Temperature event, this serialized form of the event is sent to IoT Hub:

```
{"Temperature":75, "Time":"2015-09-17T18:45:56Z"}
```

We're sending a temperature which is of type **TemperatureEvent** and that struct contains a **Temperature** and **Time** member. This is directly reflected in the serialized data.

Similarly, we can send a humidity event with this code:

```

thermostat->Humidity.Humidity = 45;
thermostat->Humidity.Time = GetDateTimeOffset(now);
if (SERIALIZE(&destination, &destinationSize, thermostat->Humidity) == IOT_AGENT_OK)
{
    sendMessage(iotHubClientHandle, destination, destinationSize);
}

```

The serialized form that's sent to IoT Hub appears as follows:

```
{"Humidity":45, "Time":"2015-09-17T18:45:56Z"}
```

Again, this is as expected.

With this model, you can imagine how additional events can easily be added. You define more structures using **DECLARE_STRUCT**, and include the corresponding event in the model using **WITH_DATA**.

Now, let's modify the model so that it includes the same data but with a different structure.

Model 2

Consider this alternative model to the one above:

```
DECLARE_MODEL(Thermostat,
    WITH_DATA(int, Temperature),
    WITH_DATA(int, Humidity),
    WITH_DATA(EDM_DATE_TIME_OFFSET, Time)
);
```

In this case we've eliminated the **DECLARE_STRUCT** macros and are simply defining the data items from our scenario using simple types from the modeling language.

Just for the moment, ignore the **Time** event. With that aside, here's the code to ingress **Temperature**:

```
time_t now;
time(&now);
thermostat->Temperature = 75;

unsigned char* destination;
size_t destinationSize;
if (SERIALIZE(&destination, &destinationSize, thermostat->Temperature) == IOT_AGENT_OK)
{
    sendMessage(iotHubClientHandle, destination, destinationSize);
}
```

This code sends the following serialized event to IoT Hub:

```
{"Temperature":75}
```

And the code for sending the **Humidity** event appears as follows:

```
thermostat->Humidity = 45;
if (SERIALIZE(&destination, &destinationSize, thermostat->Humidity) == IOT_AGENT_OK)
{
    sendMessage(iotHubClientHandle, destination, destinationSize);
}
```

This code sends this to IoT Hub:

```
{"Humidity":45}
```

So far there are still no surprises. Now let's change how we use the **SERIALIZE** macro.

The **SERIALIZE** macro can take multiple data events as arguments. This enables us to serialize the **Temperature** and **Humidity** event together and send them to IoT Hub in one call:

```
if (SERIALIZE(&destination, &destinationSize, thermostat->Temperature, thermostat->Humidity) == IOT_AGENT_OK)
{
    sendMessage(iotHubClientHandle, destination, destinationSize);
}
```

You might guess that the result of this code is that two data events are sent to IoT Hub:

```
[ {"Temperature":75}, {"Humidity":45} ]
```

In other words, you might expect that this code is the same as sending **Temperature** and **Humidity** separately. It's just a convenience to pass both events to **SERIALIZE** in the same call. However, that's not the case. Instead, the code above sends this single data event to IoT Hub:

```
{"Temperature":75, "Humidity":45}
```

This may seem strange because our model defines **Temperature** and **Humidity** as two *separate* events:

```
DECLARE_MODEL(Thermostat,  
WITH_DATA(int, Temperature),  
WITH_DATA(int, Humidity),  
WITH_DATA(EDM_DATE_TIME_OFFSET, Time)  
);
```

More to the point, we didn't model these events where **Temperature** and **Humidity** are in the same structure:

```
DECLARE_STRUCT(TemperatureAndHumidityEvent,  
int, Temperature,  
int, Humidity,  
);  
  
DECLARE_MODEL(Thermostat,  
WITH_DATA(TemperatureAndHumidityEvent, TemperatureAndHumidity),  
);
```

If we used this model, it would be easier to understand how **Temperature** and **Humidity** would be sent in the same serialized message. However it may not be clear why it works that way when you pass both data events to **SERIALIZE** using model 2.

This behavior is easier to understand if you know the assumptions that the **serializer** library is making. To make sense of this let's go back to our model:

```
DECLARE_MODEL(Thermostat,  
WITH_DATA(int, Temperature),  
WITH_DATA(int, Humidity),  
WITH_DATA(EDM_DATE_TIME_OFFSET, Time)  
);
```

Think of this model in object-oriented terms. In this case we're modeling a physical device (a thermostat) and that device includes attributes like **Temperature** and **Humidity**.

We can send the entire state of our model with code such as the following:

```
if (SERIALIZE(&destination, &destinationSize, thermostat->Temperature, thermostat->Humidity, thermostat->Time)  
== IOT_AGENT_OK)  
{  
    sendMessage(iotHubClientHandle, destination, destinationSize);  
}
```

Assuming the values of Temperature, Humidity and Time are set, we would see an event like this sent to IoT Hub:

```
{"Temperature":75, "Humidity":45, "Time":"2015-09-17T18:45:56Z"}
```

Sometimes you may only want to send *some* properties of the model to the cloud (this is especially true if your model contains a large number of data events). It's useful to send only a subset of data events, such as in our earlier example:

```
{"Temperature":75, "Time":"2015-09-17T18:45:56Z"}
```

This generates exactly the same serialized event as if we had defined a **TemperatureEvent** with a **Temperature** and **Time** member, just as we did with model 1. In this case we were able to generate exactly the same serialized event by using a different model (model 2) because we called **SERIALIZE** in a different way.

The important point is that if you pass multiple data events to **SERIALIZE**, then it assumes each event is a property in a single JSON object.

The best approach depends on you and how you think about your model. If you're sending "events" to the cloud and each event contains a defined set of properties, then the first approach makes a lot of sense. In that case you would use **DECLARE_STRUCT** to define the structure of each event and then include them in your model with the **WITH_DATA** macro. Then you send each event as we did in the first example above. In this approach you would only pass a single data event to **SERIALIZER**.

If you think about your model in an object-oriented fashion, then the second approach may suit you. In this case, the elements defined using **WITH_DATA** are the "properties" of your object. You pass whatever subset of events to **SERIALIZE** that you like, depending on how much of your "object's" state you want to send to the cloud.

Neither approach is right or wrong. Just be aware of how the **serializer** library works, and pick the modeling approach that best fits your needs.

Message handling

So far this article has only discussed sending events to IoT Hub, and hasn't addressed receiving messages. The reason for this is that what we need to know about receiving messages has largely been covered in the article [Azure IoT device SDK for C](#). Recall from that article that you process messages by registering a message callback function:

```
IoTHubClient_SetMessageCallback(iotHubClientHandle, IoTHubMessage, myWeather)
```

You then write the callback function that's invoked when a message is received:

```

static IOTHUBMESSAGE_DISPOSITION_RESULT IoTHubMessage(IOTHUB_MESSAGE_HANDLE message, void*
userContextCallback)
{
    IOTHUBMESSAGE_DISPOSITION_RESULT result;
    const unsigned char* buffer;
    size_t size;
    if (IoTHubMessage_GetByteArray(message, &buffer, &size) != IOTHUB_MESSAGE_OK)
    {
        printf("unable to IoTHubMessage_GetByteArray\r\n");
        result = EXECUTE_COMMAND_ERROR;
    }
    else
    {
        /*buffer is not zero terminated*/
        char* temp = malloc(size + 1);
        if (temp == NULL)
        {
            printf("failed to malloc\r\n");
            result = EXECUTE_COMMAND_ERROR;
        }
        else
        {
            memcpy(temp, buffer, size);
            temp[size] = '\0';
            EXECUTE_COMMAND_RESULT executeCommandResult = EXECUTE_COMMAND(userContextCallback, temp);
            result =
                (executeCommandResult == EXECUTE_COMMAND_ERROR) ? IOTHUBMESSAGE_ABANDONED :
                (executeCommandResult == EXECUTE_COMMAND_SUCCESS) ? IOTHUBMESSAGE_ACCEPTED :
                IOTHUBMESSAGE_REJECTED;
            free(temp);
        }
    }
    return result;
}

```

This implementation of **IoTHubMessage** calls the specific function for each action in your model. For example, if your model defines this action:

```
WITH_ACTION(SetAirResistance, int, Position)
```

You must define a function with this signature:

```

EXECUTE_COMMAND_RESULT SetAirResistance(ContosoAnemometer* device, int Position)
{
    (void)device;
    (void)printf("Setting Air Resistance Position to %d.\r\n", Position);
    return EXECUTE_COMMAND_SUCCESS;
}

```

SetAirResistance is then called when that message is sent to your device.

What we haven't explained yet is what the serialized version of message looks like. In other words, if you want to send a **SetAirResistance** message to your device, what does that look like?

If you're sending a message to a device, you would do so through the Azure IoT service SDK. You still need to know what string to send to invoke a particular action. The general format for sending a message appears as follows:

```
{"Name" : "", "Parameters" : "" }
```

You're sending a serialized JSON object with two properties: **Name** is the name of the action (message) and **Parameters** contains the parameters of that action.

For example, to invoke **SetAirResistance** you can send this message to a device:

```
{"Name" : "SetAirResistance", "Parameters" : { "Position" : 5 }}
```

The action name must exactly match an action defined in your model. The parameter names must match as well. Also note case sensitivity. **Name** and **Parameters** are always uppercase. Make sure to match the case of your action name and parameters in your model. In this example, the action name is "SetAirResistance" and not "setairresistance".

The two other actions **TurnFanOn** and **TurnFanOff** can be invoked by sending these messages to a device:

```
{"Name" : "TurnFanOn", "Parameters" : {}}
{"Name" : "TurnFanOff", "Parameters" : {}}
```

This section described everything you need to know when sending events and receiving messages with the **serializer** library. Before moving on, let's cover some parameters you can configure that control how large your model is.

Macro configuration

If you're using the **Serializer** library an important part of the SDK to be aware of is found in the `azure-c-shared-utility` library.

If you have cloned the `Azure-iot-sdk-c` repository from GitHub using the --recursive option, then you will find this shared utility library here:

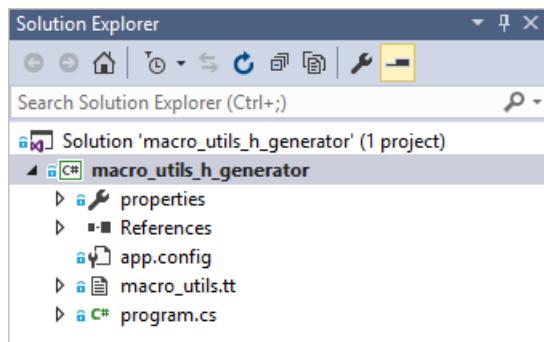
```
.\\c-utility
```

If you have not cloned the library, you can find it [here](#).

Within the shared utility library, you will find the following folder:

```
azure-c-shared-utility\\macro\\_utils\\_h\\generator.
```

This folder contains a Visual Studio solution called **macro_utils_h_generator.sln**:



The program in this solution generates the **macro_utils.h** file. There's a default `macro_utils.h` file included with the SDK. This solution allows you to modify some parameters and then recreate the header file based on these parameters.

The two key parameters to be concerned with are **nArithmetic** and **nMacroParameters** which are defined in

these two lines found in macro_utils.tt:

```
<#int nArithmetic=1024;#>
<#int nMacroParameters=124; /*127 parameters in one macro definition in C99 in chapter 5.2.4.1 Translation
limits*/#>
```

These values are the default parameters included with the SDK. Each parameter has the following meaning:

- nMacroParameters – Controls how many parameters you can have in one DECLARE_MODEL macro definition.
- nArithmetic – Controls the total number of members allowed in a model.

The reason these parameters are important is because they control how large your model can be. For example, consider this model definition:

```
DECLARE_MODEL(MyModel,
WITH_DATA(int, MyData)
);
```

As mentioned previously, **DECLARE_MODEL** is just a C macro. The names of the model and the **WITH_DATA** statement (yet another macro) are parameters of **DECLARE_MODEL**. **nMacroParameters** defines how many parameters can be included in **DECLARE_MODEL**. Effectively, this defines how many data event and action declarations you can have. As such, with the default limit of 124 this means that you can define a model with a combination of about 60 actions and data events. If you try to exceed this limit, you'll receive compiler errors that look similar to this:

⚠ C4013	'FOR_EACH_2_1' undefined; assuming extern returning int	SimpleSample_HT simplesample_htt 103
✖ C2065	'FIELD_AS_STRING': undeclared identifier	SimpleSample_HT simplesample_htt 103
⚠ C4013	'FOR_EACH_1_COUNTED_COUNT_ARG_DISPATCH_EMPTY_intint' undefined; assuming extern returning int	SimpleSample_HT simplesample_htt 103
✖ C2143	syntax error: missing ')' before ','	SimpleSample_HT simplesample_htt 103

The **nArithmetic** parameter is more about the internal workings of the macro language than your application. It controls the total number of members you can have in your model, including **DECLARE_STRUCT** macros. If you start seeing compiler errors such as this, then you should try increasing **nArithmetic**:

✖ C2143	syntax error: missing ')' before '('	simplesample_htt simplesample_htt 34
✖ C2122	'ContosoAnemometer': prototype parameter in name list illegal	simplesample_htt simplesample_htt 34
✖ C2081	'Devideld': name in formal parameter list illegal	simplesample_htt simplesample_htt 34
✖ C2091	function returns function	simplesample_htt simplesample_htt 34

If you want to change these parameters, modify the values in the macro_utils.tt file, recompile the macro_utils_h_generator.sln solution, and run the compiled program. When you do so, a new macro_utils.h file is generated and placed in the .\common\inc directory.

In order to use the new version of macro_utils.h, remove the **serializer** NuGet package from your solution and in its place include the **serializer** Visual Studio project. This enables your code to compile against the source code of the serializer library. This includes the updated macro_utils.h. If you want to do this for **simplesample_amqp**, start by removing the NuGet package for the serializer library from the solution:

NuGet Package Manager: simplesample_amqp

The screenshot shows the NuGet Package Manager interface with the following details:

- Package source: nuget.org
- Filter: Installed
- Include prerelease: unchecked
- Search (Ctrl+E): Search bar
- Settings: gear icon

Prerelease	Package	Description	Checkmark
Apache.QPID.Proton.Azurelet	Apache Qpid Proton C AMQP 1.0 library with modifications for Azure IoT Hub	description: Qpid Proton is a high-performance, lightweight messaging library. It can be used in the widest range of messaging applic...	✓
Microsoft.Azure.IoTHub.AmqpTransport	This nuget package can be used to connect to IoTHub over AMQP.		✓
Microsoft.Azure.IoTHub.Common	This nuget package contains common functionalities shared between AMQP, MQTT and HTTP transports to IoTHub		✓
Microsoft.Azure.IoTHub.IoTHubClient	This nuget package contains common interface shared between AMQP, MQTT and HTTP transports to IoTHub		✓
Microsoft.Azure.IoTHub.Serializer	This nuget package contains a serializer library for IoTHub		✓

Then add this project to your Visual Studio solution:

```
.\c\serializer\build\windows\serializer.vcxproj
```

When you're done, your solution should look like this:

The Solution Explorer shows two projects:

- serializer**: Contains External Dependencies, Header Files, References, Resource Files, and Source Files.
- simplesample_amqp**: Contains External Dependencies, Header Files, References, Resource Files, Source Files (with files main.c and simplesample_amqp.c), and packages.config.

Now when you compile your solution, the updated macro_utils.h is included in your binary.

Note that increasing these values high enough can exceed compiler limits. To this point, the **nMacroParameters** is the main parameter with which to be concerned. The C99 spec specifies that a minimum of 127 parameters are allowed in a macro definition. The Microsoft compiler follows the spec exactly (and has a limit of 127), so you won't be able to increase **nMacroParameters** beyond the default. Other compilers might allow you to do so (for example, the GNU compiler supports a higher limit).

So far we've covered just about everything you need to know about how to write code with the **serializer** library. Before concluding, let's revisit some topics from previous articles that you may be wondering about.

The lower-level APIs

The sample application on which this article focused is **simplesample_amqp**. This sample uses the higher-level (the non-**LL**) APIs to send events and receive messages. If you use these APIs, a background thread runs which takes care of both sending events and receiving messages. However, you can use the lower-level (**LL**) APIs to eliminate this background thread and take explicit control over when you send events or receive messages from the cloud.

As described in a [previous article](#), there is a set of functions that consists of the higher-level APIs:

- `IoTHubClient_CreateFromConnectionString`
- `IoTHubClient_SendEventAsync`
- `IoTHubClient_SetMessageCallback`
- `IoTHubClient_Destroy`

These APIs are demonstrated in **simplesample_amqp**.

There is also an analogous set of lower-level APIs.

- `IoTHubClient_LL_CreateFromConnectionString`
- `IoTHubClient_LL_SendEventAsync`
- `IoTHubClient_LL_SetMessageCallback`
- `IoTHubClient_LL_Destroy`

Note that the lower-level APIs work exactly the same way as described in the previous articles. You can use the first set of APIs if you want a background thread to handle sending events and receiving messages. You use the second set of APIs if you want explicit control over when you send and receive data from IoT Hub. Either set of APIs work equally well with the **serializer** library.

For an example of how the lower-level APIs are used with the **serializer** library, see the **simplesample_http** application.

Additional topics

A few other topics worth mentioning again are property handling, using alternate device credentials, and configuration options. These are all topics covered in a [previous article](#). The main point is that all of these features work in the same way with the **serializer** library as they do with the **IoTHubClient** library. For example, if you want to attach properties to an event from your model, you use **IoTHubMessage_Properties** and **Map_AddOrUpdate**, the same way as described previously:

```
MAP_HANDLE propMap = IoTHubMessage_Properties(message.messageHandle);
sprintf_s(propText, sizeof(propText), "%d", i);
Map_AddOrUpdate(propMap, "SequenceNumber", propText);
```

Whether the event was generated from the **serializer** library or created manually using the **IoTHubClient** library does not matter.

For the alternate device credentials, using **IoTHubClient_LL_Create** works just as well as **IoTHubClient_CreateFromConnectionString** for allocating an **IOTHUB_CLIENT_HANDLE**.

Finally, if you're using the **serializer** library, you can set configuration options with **IoTHubClient_LL_SetOption** just as you did when using the **IoTHubClient** library.

A feature that is unique to the **serializer** library are the initialization APIs. Before you can start working with the library, you must call **serializer_init**:

```
serializer_init(NULL);
```

This is done just before you call **IoTHubClient_CreateFromConnectionString**.

Similarly, when you're done working with the library, the last call you'll make is to **serializer_deinit**:

```
serializer_deinit();
```

Otherwise, all of the other features listed above work the same in the **serializer** library as they do in the **IoTHubClient** library. For more information about any of these topics, see the [previous article](#) in this series.

Next steps

This article describes in detail the unique aspects of the **serializer** library contained in the **Azure IoT device SDK for C**. With the information provided you should have a good understanding of how to use models to send events and receive messages from IoT Hub.

This also concludes the three-part series on how to develop applications with the **Azure IoT device SDK for C**. This should be enough information to not only get you started but give you a thorough understanding of how the APIs work. For additional information, there are a few samples in the SDK not covered here. Otherwise, the [Azure IoT SDK documentation](#) is a good resource for additional information.

To learn more about developing for IoT Hub, see the [Azure IoT SDKs](#).

To further explore the capabilities of IoT Hub, see [Deploying AI to edge devices with Azure IoT Edge](#).

Develop for constrained devices using Azure IoT C SDK

9/28/2018 • 3 minutes to read

Azure IoT Hub C SDK is written in ANSI C (C99), which makes it well-suited to operate a variety of platforms with small disk and memory footprint. The recommended RAM is at least 64 KB, but the exact memory footprint depends on the protocol used, the number of connections opened, as well as the platform targeted.

C SDK is available in package form from apt-get, NuGet, and MBED. To target constrained devices, you may want to build the SDK locally for your target platform. This documentation demonstrates how to remove certain features to shrink the footprint of the C SDK using [cmake](#). In addition, this documentation discusses the best practice programming models for working with constrained devices.

Building the C SDK for constrained devices

Build the C SDK for constrained devices.

Prerequisites

Follow this [C SDK setup guide](#) to prepare your development environment for building the C SDK. Before you get to the step for building with cmake, you can invoke cmake flags to remove unused features.

Remove additional protocol libraries

C SDK supports five protocols today: MQTT, MQTT over WebSocket, AMQPs, AMQP over WebSocket, and HTTPS. Most scenarios require one to two protocols running on a client, hence you can remove the protocol library you are not using from the SDK. Additional information about choosing the appropriate communication protocol for your scenario can be found in [Choose an IoT Hub communication protocol](#). For example, MQTT is a lightweight protocol that is often better suited for constrained devices.

You can remove the AMQP and HTTP libraries using the following cmake command:

```
cmake -Duse_amqp=OFF -Duse_http=OFF <Path_to_cmake>
```

Remove SDK logging capability

The C SDK provides extensive logging throughout to help with debugging. You can remove the logging capability for production devices using the following cmake command:

```
cmake -Dno_logging=OFF <Path_to_cmake>
```

Remove upload to blob capability

You can upload large files to Azure Storage using the built-in capability in the SDK. Azure IoT Hub acts as a dispatcher to an associated Azure Storage account. You can use this feature to send media files, large telemetry batches, and logs. You can get more information in [uploading files with IoT Hub](#). If your application does not require this functionality, you can remove this feature using the following cmake command:

```
cmake -Ddont_use_uploadtoblob=ON <Path_to_cmake>
```

Running strip on Linux environment

If your binaries run on Linux system, you can leverage the [strip command](#) to reduce the size of the final application after compiling.

```
strip -s <Path_to_executable>
```

Programming models for constrained devices

Next, look at programming models for constrained devices.

Avoid using the Serializer

The C SDK has an optional [C SDK serializer](#), which allows you to use declarative mapping tables to define methods and device twin properties. The serializer is designed to simplify development, but it adds overhead, which is not optimal for constrained devices. In this case, consider using primitive client APIs and parse JSON by using a lightweight parser such as [parson](#).

Use the lower layer (*LL*)

The C SDK supports two programming models. One set has APIs with an *LL* infix, which stands for lower layer. This set of APIs is lighter weight and do not spin up worker threads, which means the user must manually control scheduling. For example, for the device client, the *LL* APIs can be found in this [header file](#).

Another set of APIs without the *LL* index is called the convenience layer, where a worker thread is spun automatically. For example, the convenience layer APIs for the device client can be found in this [IoT Device Client header file](#). For constrained devices where each extra thread can take a substantial percentage of system resources, consider using *LL* APIs.

Next steps

To learn more about Azure IoT C SDK architecture:

- [Azure IoT C SDK source code](#)
- [Azure IoT device SDK for C introduction](#)

Develop for mobile devices using Azure IoT SDKs

1/4/2019 • 2 minutes to read

Things in the Internet of Things may refer to a wide range of devices with varying capability: sensors, microcontrollers, smart devices, industrial gateways, and even mobile devices. A mobile device can be an IoT device, where it is sending device-to-cloud telemetry and managed by the cloud. It can also be the device running a back-end service application, which manages other IoT devices. In both cases, [Azure IoT Hub SDKs](#) can be used to develop applications that work for mobile devices.

Develop for native iOS platform

Azure IoT Hub SDKs provide native iOS platform support through Azure IoT Hub C SDK. You can think of it as an iOS SDK that you can incorporate in your Swift or Objective C XCode project. There are two ways to use the C SDK on iOS:

- Use the CocoaPod libraries in XCode project directly.
- Download the source code for C SDK and build for iOS platform following the [build instruction](#) for MacOS.

Azure IoT Hub C SDK is written in C99 for maximum portability to various platforms. The porting process involves writing a thin adoption layer for the platform-specific components, which can be found here for [iOS](#). The features in the C SDK can be leveraged on iOS platform, including the Azure IoT Hub primitives supported and SDK-specific features such as retry policy for network reliability. The interface for iOS SDK is also similar to the interface for Azure IoT Hub C SDK.

These documentations walk through how to develop a device application or service application on an iOS device:

- [Quickstart: Send telemetry from a device to an IoT hub](#)
- [Send messages from the cloud to your device with IoT hub](#)

Develop with Azure IoT Hub CocoaPod libraries

Azure IoT Hub SDKs releases a set of Objective-C CocoaPod libraries for iOS development. To see the latest list of CocoaPod libraries, see [CocoaPods for Microsoft Azure IoT](#). Once the relevant libraries are incorporated into your XCode project, there are two ways to write IoT Hub related code:

- Objective C function: If your project is written in Objective-C, you can call APIs from Azure IoT Hub C SDK directly. If your project is written in Swift, you can call `@objc func` before creating your function, and proceed to writing all logics related to Azure IoT Hub using C or Objective-C code. A set of samples demonstrating both can be found in the [sample repository](#).
- Incorporate C samples: If you have written a C device application, you can reference it directly in your XCode project:
 - Add the sample.c file to your XCode project from XCode.
 - Add the header file to your dependency. A header file is included in the [sample repository](#) as an example. For more information, please visit Apple's documentation page for [Objective-C](#).

Develop for Android platform

Azure IoT Hub Java SDK supports Android platform. For the specific API version tested, please visit our [platform support page](#) for the latest update.

These documentations walk through how to develop a device application or service application on an Android

device using Gradle and Android Studio:

- [Quickstart: Send telemetry from a device to an IoT hub](#)
- [Quickstart: Control a device](#)

Next steps

- [IoT Hub REST API reference](#)
- [Azure IoT C SDK source code](#)

Manage connectivity and reliable messaging by using Azure IoT Hub device SDKs

2/4/2019 • 4 minutes to read

This article provides high-level guidance to help you design device applications that are more resilient. It shows you how to take advantage of the connectivity and reliable messaging features in Azure IoT device SDKs. The goal of this guide is to help you manage the following scenarios:

- Fixing a dropped network connection
- Switching between different network connections
- Reconnecting because of service transient connection errors

Implementation details may vary by language. For more information, see the API documentation or specific SDK:

- [C/Python/iOS SDK](#)
- [.NET SDK](#)
- [Java SDK](#)
- [Node SDK](#)

Designing for resiliency

IoT devices often rely on non-continuous or unstable network connections (for example, GSM or satellite). Errors can occur when devices interact with cloud-based services because of intermittent service availability and infrastructure-level or transient faults. An application that runs on a device has to manage the mechanisms for connection, reconnection, and the retry logic for sending and receiving messages. Also, the retry strategy requirements depend heavily on the device's IoT scenario, context, capabilities.

The Azure IoT Hub device SDKs aim to simplify connecting and communicating from cloud-to-device and device-to-cloud. These SDKs provide a robust way to connect to Azure IoT Hub and a comprehensive set of options for sending and receiving messages. Developers can also modify existing implementation to customize a better retry strategy for a given scenario.

The relevant SDK features that support connectivity and reliable messaging are covered in the following sections.

Connection and retry

This section gives an overview of the reconnection and retry patterns available when managing connections. It details implementation guidance for using a different retry policy in your device application and lists relevant APIs from the device SDKs.

Error patterns

Connection failures can happen at many levels:

- Network errors: disconnected socket and name resolution errors
- Protocol-level errors for HTTP, AMQP, and MQTT transport: detached links or expired sessions
- Application-level errors that result from either local mistakes: invalid credentials or service behavior (for example, exceeding the quota or throttling)

The device SDKs detect errors at all three levels. OS-related errors and hardware errors are not detected and handled by the device SDKs. The SDK design is based on [The Transient Fault Handling Guidance](#) from the Azure

Retry patterns

The following steps describe the retry process when connection errors are detected:

1. The SDK detects the error and the associated error in the network, protocol, or application.
2. The SDK uses the error filter to determine the error type and decide if a retry is needed.
3. If the SDK identifies an **unrecoverable error**, operations like connection, send, and receive are stopped. The SDK notifies the user. Examples of unrecoverable errors include an authentication error and a bad endpoint error.
4. If the SDK identifies a **recoverable error**, it retries according to the specified retry policy until the defined timeout elapses. Note that the SDK uses **Exponential back-off with jitter** retry policy by default.
5. When the defined timeout expires, the SDK stops trying to connect or send. It notifies the user.
6. The SDK allows the user to attach a callback to receive connection status changes.

The SDKs provide three retry policies:

- **Exponential back-off with jitter:** This default retry policy tends to be aggressive at the start and slow down over time until it reaches a maximum delay. The design is based on [Retry guidance from Azure Architecture Center](#).
- **Custom retry:** For some SDK languages, you can design a custom retry policy that is better suited for your scenario and then inject it into the `RetryPolicy`. Custom retry isn't available on the C SDK.
- **No retry:** You can set retry policy to "no retry," which disables the retry logic. The SDK tries to connect once and send a message once, assuming the connection is established. This policy is typically used in scenarios with bandwidth or cost concerns. If you choose this option, messages that fail to send are lost and can't be recovered.

Retry policy APIs

SDK	SETRETRYPOLICY METHOD	POLICY IMPLEMENTATIONS	IMPLEMENTATION GUIDANCE
C/Python/iOS	<code>IOTHUB_CLIENT_RESULT</code> <code>IoTHubClient_SetRetryPolicy</code>	Default: <code>IOTHUB_CLIENT_RETRY_EXPONENTIAL_BACKOFF</code> Custom: use available <code>retryPolicy</code> No retry: <code>IOTHUB_CLIENT_RETRY_NONE</code>	C/Python/iOS implementation
Java	<code>SetRetryPolicy</code>	Default: <code>ExponentialBackoffWithJitter class</code> Custom: implement <code>RetryPolicy interface</code> No retry: <code>NoRetry class</code>	Java implementation
.NET	<code>DeviceClient.SetRetryPolicy</code>	Default: <code>ExponentialBackoff class</code> Custom: implement <code>IRetryPolicy interface</code> No retry: <code>NoRetry class</code>	C# implementation

SDK	SETRETRYPOLICY METHOD	POLICY IMPLEMENTATIONS	IMPLEMENTATION GUIDANCE
Node	setRetryPolicy	Default: ExponentialBackoffWithJitter class Custom: implement RetryPolicy interface No retry: NoRetry class	Node implementation

The following code samples illustrate this flow:

.NET implementation guidance

The following code sample shows how to define and set the default retry policy:

```
# define/set default retry policy
RetryPolicy retryPolicy = new ExponentialBackoff(int.MaxValue, TimeSpan.FromMilliseconds(100),
TimeSpan.FromSeconds(10), TimeSpan.FromMilliseconds(100));
SetRetryPolicy(retryPolicy);
```

To avoid high CPU usage, the retries are throttled if the code fails immediately. For example, when there's no network or route to the destination. The minimum time to execute the next retry is 1 second.

If the service responds with a throttling error, the retry policy is different and can't be changed via public API:

```
# throttled retry policy
RetryPolicy retryPolicy = new ExponentialBackoff(RetryCount, TimeSpan.FromSeconds(10),
TimeSpan.FromSeconds(60), TimeSpan.FromSeconds(5));
SetRetryPolicy(retryPolicy);
```

The retry mechanism stops after `DefaultOperationTimeoutInMilliseconds`, which is currently set at 4 minutes.

Other languages implementation guidance

For code samples in other languages, review the following implementation documents. The repository contains samples that demonstrate the use of retry policy APIs.

- [C/Python/iOS SDK](#)
- [.NET SDK](#)
- [Java SDK](#)
- [Node SDK](#)

Next steps

- [Use device and service SDKs](#)
- [Use the IoT device SDK for C](#)
- [Develop for constrained devices](#)
- [Develop for mobile devices](#)
- [Troubleshoot device disconnects](#)

Develop for Android Things platform using Azure IoT SDKs

2/22/2019 • 6 minutes to read

Azure IoT Hub [SDKs](#) provide first tier support for popular platforms such as Windows, Linux, OSX, MBED, and mobile platforms like Android and iOS. As part of our commitment to enable greater choice and flexibility in IoT deployments, the Java SDK also supports [Android Things](#) platform. Developers can leverage the benefits of Android Things operating system on the device side, while using [Azure IoT Hub](#) as the central message hub that scales to millions of simultaneously connected devices.

This tutorial outlines the steps to build a device side application on Android Things using the Azure IoT Java SDK.

Prerequisites

- An Android Things supported hardware with Android Things OS running. You can follow [Android Things documentation](#) on how to flash Android Things OS. Make sure your Android Things device is connected to the internet with essential peripherals such as keyboard, display, and mouse attached. This tutorial uses Raspberry Pi 3.
- Latest version of [Android Studio](#)
- Latest version of [Git](#)

Open Azure Cloud Shell

Azure Cloud Shell is a free, interactive shell that you can use to run the steps in this article. Common Azure tools are preinstalled and configured in Cloud Shell for you to use with your account. Just select the **Copy** button to copy the code, paste it in Cloud Shell, and then press Enter to run it. There are a few ways to open Cloud Shell:

Select Try It in the upper-right corner of a code block.	
Open Cloud Shell in your browser.	
Select the Cloud Shell button on the menu in the upper-right corner of the Azure portal .	

Create an IoT hub

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose +**Create a resource**, then choose **Internet of Things**.
3. Click **Iot Hub** from the list on the right. You see the first screen for creating an IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Review + create

Create an IoT Hub to help you connect, monitor, and manage billions of your IoT assets. [Learn More](#)

PROJECT DETAILS

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

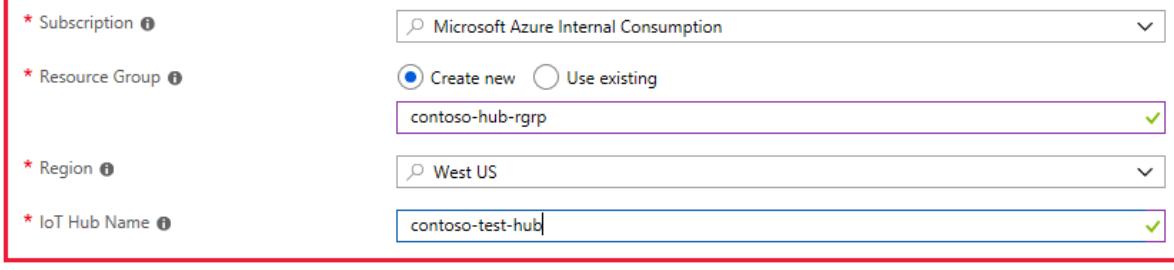
* Subscription [?](#) Microsoft Azure Internal Consumption

* Resource Group [?](#) Create new Use existing
contoso-hub-rgrp

* Region [?](#) West US

* IoT Hub Name [?](#) contoso-test-hub

Review + create **Next: Size and scale »** Automation options



Fill in the fields.

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics **Size and scale** Review + create

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS

* Pricing and scale tier [?](#) S1: Standard tier [▼](#) [Learn how to choose the right IoT Hub tier for your solution](#)

Number of S1 IoT Hub units [?](#) [1](#) This determines your IoT Hub scale capability and can be changed as your need increases.

Device-to-cloud-messages [?](#) Enabled

Message routing [?](#) Enabled

Cloud-to-device commands [?](#) Enabled

IoT Edge [?](#) Enabled

Device management [?](#) Enabled

Advanced Settings

Device-to-cloud partitions [?](#) [4](#)

Review + create [« Previous: Basics](#) Automation options

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: The number of messages allowed per unit per day depends on your hub's pricing tier. For example, if you want the IoT hub to support ingress of 700,000 messages, you choose two S1 tier units.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

Advanced / Device-to-cloud partitions: This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most IoT hubs only need four partitions.

5. Click **Review + create** to review your choices. You see something similar to this screen.

The screenshot shows the 'Review + create' step of the Azure IoT hub creation wizard. It displays the following configuration:

Setting	Value
Subscription	Microsoft Azure Internal Consumption
Resource Group	contoso-hub-rgrp
Region	West US
IoT Hub Name	contoso-test-hub
Pricing and scale tier	S1
Number of S1 IoT Hub units	1
Messages per day	400,000
Cost per month	25.00 USD

At the bottom, there are three buttons: 'Create' (highlighted with a red box), '[« Previous: Size and scale](#)', and '[Automation options](#)'.

6. Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Register a device

A device must be registered with your IoT hub before it can connect. In this quickstart, you use the Azure Cloud Shell to register a simulated device.

1. Run the following commands in Azure Cloud Shell to add the IoT Hub CLI extension and to create the device identity.

YourIoTHubName : Replace this placeholder below with the name you choose for your IoT hub.

MyAndroidThingsDevice : This is the name given for the registered device. Use MyAndroidThingsDevice as shown. If you choose a different name for your device, you will also need to use that name throughout this article, and update the device name in the sample applications before you run them.

```
az extension add --name azure-cli-iot-ext
az iot hub device-identity create --hub-name YourIoTHubName --device-id MyAndroidThingsDevice
```

2. Run the following commands in Azure Cloud Shell to get the *device connection string* for the device you just registered: **YourIoTHubName** : Replace this placeholder below with the name you choose for your IoT hub.

```
az iot hub device-identity show-connection-string --hub-name YourIoTHubName --device-id
MyAndroidThingsDevice --output table
```

Make a note of the device connection string, which looks like:

```
HostName={YourIoTHubName}.azure-devices.net;DeviceId=MyAndroidThingsDevice;SharedAccessKey=
{YourSharedAccessKey}
```

You use this value later in the quickstart.

Building an Android Things application

1. The first step to building an Android Things application is connecting to your Android Things devices. Connect your Android Things device to a display and connect it to the internet. Android Things provide [documentation](#) on how to connect to WiFi. After you have connected to the internet, take a note of the IP address listed under Networks.
2. Use the `adb` tool to connect to your Android Things device with the IP address noted above. Double check the connection by using this command from your terminal. You should see your devices listed as "connected"
`adb devices`
3. Download our sample for Android/Android Things from this [repository](#) or use Git.
`git clone https://github.com/Azure-Samples/azure-iot-samples-java.git`
4. In Android Studio, open the Android Project located in "`\azure-iot-samples-java\iot-hub\Samples\device\AndroidSample`".
5. Open `gradle.properties` file, and replace "`Device_connection_string`" with your device connection string noted earlier.
6. Click on Run - Debug and select your device to deploy this code to your Android Things devices.
7. When the application is started successfully, you can see an application running on your Android Things device. This sample application sends randomly generated temperature readings.

Read the telemetry from your hub

You can view the data through your IoT hub as it is received. The IoT Hub CLI extension can connect to the service-side **Events** endpoint on your IoT Hub. The extension receives the device-to-cloud messages sent from your simulated device. An IoT Hub back-end application typically runs in the cloud to receive and process device-to-cloud messages.

Run the following commands in Azure Cloud Shell, replacing `YourIoTHubName` with the name of your IoT hub:

```
azurecli-interactive
az iot hub monitor-events --device-id MyAndroidThingsDevice --hub-name YourIoTHubName
```

Clean up resources

If you will be continuing to the next recommended article, you can keep the resources you've already created and reuse them.

Otherwise, you can delete the Azure resources created in this article to avoid charges.

IMPORTANT

Deleting a resource group is irreversible. The resource group and all the resources contained in it are permanently deleted. Make sure that you do not accidentally delete the wrong resource group or resources. If you created the IoT Hub inside an existing resource group that contains resources you want to keep, only delete the IoT Hub resource itself instead of deleting the resource group.

To delete a resource group by name:

1. Sign in to the [Azure portal](#) and click **Resource groups**.
2. In the **Filter by name...** textbox, type the name of the resource group containing your IoT Hub.
3. To the right of your resource group in the result list, click ... then **Delete resource group**.

The screenshot shows the Microsoft Azure portal interface. On the left, the navigation bar includes 'Create a resource', 'All services', 'FAVORITES' (with 'Resource groups' highlighted by a red box), 'Dashboard', 'All resources', and 'Recent'. The main content area is titled 'Resource groups' and shows a table with one item selected. The table has columns: NAME, SUBSCRIPTION, and LOCATION. The single row contains 'TestResources', 'Prototype3', and 'LOCATION'. A red box highlights the 'TestResources' cell. To the right of the table, there is a context menu with options: 'Delete resource group' (highlighted by a red box) and '...'. At the top of the main content area, there are buttons for '+ Add', 'Edit columns', 'Refresh', and 'Assign Tags', along with a search bar and notification icons.

4. You will be asked to confirm the deletion of the resource group. Type the name of your resource group again to confirm, and then click **Delete**. After a few moments, the resource group and all of its contained resources are deleted.

Next steps

- Learn about [how to manage connectivity and reliable messaging](#) using the IoT Hub SDKs.
- Learn about how to [develop for mobile platforms](#) such as iOS and Android.
- [Azure IoT SDK platform support](#)

Query Avro data by using Azure Data Lake Analytics

1/2/2019 • 4 minutes to read

This article discusses how to query Avro data to efficiently route messages from Azure IoT Hub to Azure services. [Message Routing](#) allows you to filter data using rich queries based on message properties, message body, device twin tags, and device twin properties. To learn more about the querying capabilities in Message Routing, see the article about message routing query syntax.

The challenge has been that when Azure IoT Hub routes messages to Azure Blob storage, IoT Hub writes the content in Avro format, which has both a message body property and a message property. IoT Hub supports writing data to Blob storage only in the Avro data format, and this format is not used for any other endpoints. For more information, see an article about using Azure Storage containers. Although the Avro format is great for data and message preservation, it's a challenge to use it to query data. In comparison, JSON or CSV format is much easier for querying data.

To address non-relational big-data needs and formats and overcome this challenge, you can use many of the big-data patterns for both transforming and scaling data. One of the patterns, "pay per query", is Azure Data Lake Analytics, which is the focus of this article. Although you can easily execute the query in Hadoop or other solutions, Data Lake Analytics is often better suited for this "pay per query" approach.

There is an "extractor" for Avro in U-SQL. For more information, see [U-SQL Avro example](#).

Query and export Avro data to a CSV file

In this section, you query Avro data and export it to a CSV file in Azure Blob storage, although you could easily place the data in other repositories or data stores.

1. Set up Azure IoT Hub to route data to an Azure Blob storage endpoint by using a property in the message body to select messages.

The screenshot shows the 'Custom endpoints' tab selected in the IoT Hub blade. It lists various Azure services as potential endpoints. A 'Blob storage' entry is expanded, showing a table for configuring the storage endpoint. The table includes columns for NAME, CONTAINER NAME, BATCH FREQUENCY(SEC...), FILENAME FORMAT, and STATUS. One row is present with the values: hotTubContainer, hottubcontainer, 100, {iothub}/{partition}/{...}, and Unknown.

NAME	CONTAINER NAME	BATCH FREQUENCY(SEC...)	FILENAME FORMAT	STATUS
hotTubContainer	hottubcontainer	100	{iothub}/{partition}/{...}	Unknown

Send data from your devices to endpoints that you choose.

Routes Custom endpoints

Create an endpoint, and then add a route (you can add up to 100 from each IoT hub). Messages that don't match a query are automatically sent to messages/events if you've enabled the fallback route.

[Disable fallback route](#)

[Add](#) [Test all routes](#) [Delete](#)

<input type="checkbox"/>	NAME	DATA SOURCE	ROUTING QUERY	ENDPOINT	ENABLED
	hottubtostorage	DeviceMessages	\$body.event = 'TempUpdate' OR \$body.event = 'DoorUpdate'	hotTubContainer	true

For more information on settings up routes and custom endpoints, see [Message Routing for an IoT hub](#).

2. Ensure that your device has the encoding, content type, and needed data in either the properties or the message body, as referenced in the product documentation. When you view these attributes in Device Explorer, as shown here, you can verify that they are set correctly.

Device Explorer Twin

Configuration Management Data [Messages To Device](#) [Call Method on Device](#)

Monitoring

Event Hub:

Device ID:

Start Time:

Event Hub Data

```
Receiving events...
2/7/2018 4:54:43 PM> Device: [HomeGateway]. Data:[{
  "message": "TempUpdate:UpStairs:60.44 (56.93)",
  "event": "TempUpdate",
  "object": "UpStairs",
  "status": "60.44 (56.93)",
  "host": "UpStairs.internal.saye.org"
}]Properties:
'route': 'yes'
SYSTEM>iotHub-connection-device-id=HomeGateway
SYSTEM>iotHub-connection-auth-method={"scope":"device","type":"sas","issuer":"iothub","acceptingIpFilterRule":null}
SYSTEM>iotHub-connection-auth-generation-id=636510460778278549
SYSTEM>iotHub-enqueuedTime=2/8/2018 12:54:42 AM
SYSTEM>iotHub-message-source=Telemetry
SYSTEM>x-opt-sequence-number=49394
SYSTEM>x-opt-offset=27932800
SYSTEM>x-opt-enqueued-time=2/8/2018 12:54:43 AM
SYSTEM>EnqueuedTimeUtc=2/8/2018 12:54:43 AM
SYSTEM>SequenceNumber=49394
SYSTEM>Offset=27932800
SYSTEM>content-type=application/json
SYSTEM>content-encoding=utf-8
```

3. Set up an Azure Data Lake Store instance and a Data Lake Analytics instance. Azure IoT Hub does not route to a Data Lake Store instance, but a Data Lake Analytics instance requires one.

Subscription (change)
MSFT FTE kevinsay

Subscription ID
dc6f773e-4b13-4f8b-8d76-f34469246722

Deployments
2 Succeeded

Filter by name... All types All locations No grouping

2 items

NAME	TYPE	LOCATION
kevinsaydemo	Data Lake Analytics	East US 2
kevinsaydemo	Data Lake Store	East US 2

- In Data Lake Analytics, configure Azure Blob storage as an additional store, the same Blob storage that Azure IoT Hub routes data to.

kevinsaydemo - Data sources

Search (Ctrl+ /)

- Overview
- Activity log
- Access control (IAM)
- Tags
- Diagnose and solve problems
- SETTINGS
 - Firewall
 - Data sources**

Add data source

NAME	TYPE
kevinsaydemo (default)	Azure Data Lake Store
kevinsayazstorage	Azure Storage

- As discussed in the [U-SQL Avro example](#), you need four DLL files. Upload these files to a location in your Data Lake Store instance.

kevinsaydemo - Data explorer

kevinsaydemo Data Lake Store

Search (Ctrl+/)

- Overview
- Activity log
- Access control (IAM)
- Tags
- Diagnose and solve problems
- SETTINGS
 - Encryption
 - Firewall
 - Pricing tier
 - Properties
 - Locks
 - Automation script
- DATA LAKE STORE
 - Quick start
 - Data explorer**

kevinsaydemo

kevinsaydemo Assemblies Avro

Filter New folder Upload Access Rename folder Folder properties

NAME	SIZE
Avro.dll	124 KB
log4net.dll	270 KB
Microsoft.Analytics.Samples.Formats.dll	28.2 KB
Newtonsoft.Json.dll	654 KB

- In Visual Studio, create a U-SQL project.

!Create a U-SQL project](./media/iot-hub-query-avro-data/query-avro-data-6.png)

7. Paste the content of the following script into the newly created file. Modify the three highlighted sections: your Data Lake Analytics account, the associated DLL file paths, and the correct path for your storage account.

The screenshot shows a 'Script.usql*' window with the following configuration:

- ADLA Account:** kevinsaydemo
- Database:** Avro
- Schema:** dbo
- AU:** 5 / 100
- More Options**
- Submit**

The U-SQL script is as follows, with sections highlighted by red boxes:

```

DROP ASSEMBLY IF EXISTS [Avro];
CREATE ASSEMBLY [Avro] FROM @"/Assemblies/Avro/Avro.dll";
DROP ASSEMBLY IF EXISTS [Microsoft.Analytics.Samples.Formats];
CREATE ASSEMBLY [Microsoft.Analytics.Samples.Formats] FROM @"/Assemblies/Avro/Microsoft.Analytics.Samples.Formats.dll";
DROP ASSEMBLY IF EXISTS [Newtonsoft.Json];
CREATE ASSEMBLY [Newtonsoft.Json] FROM @"/Assemblies/Avro/Newtonsoft.Json.dll";
DROP ASSEMBLY IF EXISTS [log4net];
CREATE ASSEMBLY [log4net] FROM @"/Assemblies/Avro/log4net.dll";

REFERENCE ASSEMBLY [Newtonsoft.Json];
REFERENCE ASSEMBLY [log4net];
REFERENCE ASSEMBLY [Avro];
REFERENCE ASSEMBLY [Microsoft.Analytics.Samples.Formats];

// Blob container storage account filenames, with any path
DECLARE @input_file string = @"wasb://hottubrawdata@kevinsayzstorage/kevinsayIoT/{*}/{*}/{*}/04/23/{*}";
DECLARE @output_file string = @"/output/output.csv";

@rs =
    EXTRACT
        EnqueuedTimeUtc      string,
        Body                 byte[]
    FROM @input_file

USING new Microsoft.Analytics.Samples.Formats.ApacheAvro.AvroExtractor@"
{

```

The actual U-SQL script for simple output to a CSV file:

```

DROP ASSEMBLY IF EXISTS [Avro];
CREATE ASSEMBLY [Avro] FROM @"/Assemblies/Avro/Avro.dll";
DROP ASSEMBLY IF EXISTS [Microsoft.Analytics.Samples.Formats];
CREATE ASSEMBLY [Microsoft.Analytics.Samples.Formats] FROM
@"/Assemblies/Avro/Microsoft.Analytics.Samples.Formats.dll";
DROP ASSEMBLY IF EXISTS [Newtonsoft.Json];
CREATE ASSEMBLY [Newtonsoft.Json] FROM @"/Assemblies/Avro/Newtonsoft.Json.dll";
DROP ASSEMBLY IF EXISTS [log4net];
CREATE ASSEMBLY [log4net] FROM @"/Assemblies/Avro/log4net.dll";

REFERENCE ASSEMBLY [Newtonsoft.Json];
REFERENCE ASSEMBLY [log4net];
REFERENCE ASSEMBLY [Avro];
REFERENCE ASSEMBLY [Microsoft.Analytics.Samples.Formats];

// Blob container storage account filenames, with any path
DECLARE @input_file string =
@"wasb://hottubrawdata@kevinsayazstorage/kevinsayIoT/{*}/{*}/{*}/{*}/{*}/{*}";
DECLARE @output_file string = @"/output/output.csv";

@rs =
EXTRACT
EnqueuedTimeUtc string,
Body byte[]
FROM @input_file

USING new Microsoft.Analytics.Samples.Formats.ApacheAvro.AvroExtractor(@"
{
    ""type"":""record"",
    ""name"":""Message"",
    ""namespace"":""Microsoft.Azure.Devices"",
    ""fields"":
    [
        {
            ""name"":""EnqueuedTimeUtc"",
            ""type"":""string""
        },
        {
            ""name"":""Properties"",
            ""type"":
            {
                ""type"":""map"",
                ""values"":""string"""
            }
        },
        {
            ""name"":""SystemProperties"",
            ""type"":
            {
                ""type"":""map"",
                ""values"":""string"""
            }
        },
        {
            ""name"":""Body"",
            ""type"":[""null"",""bytes""]
        }
    ]
}
);

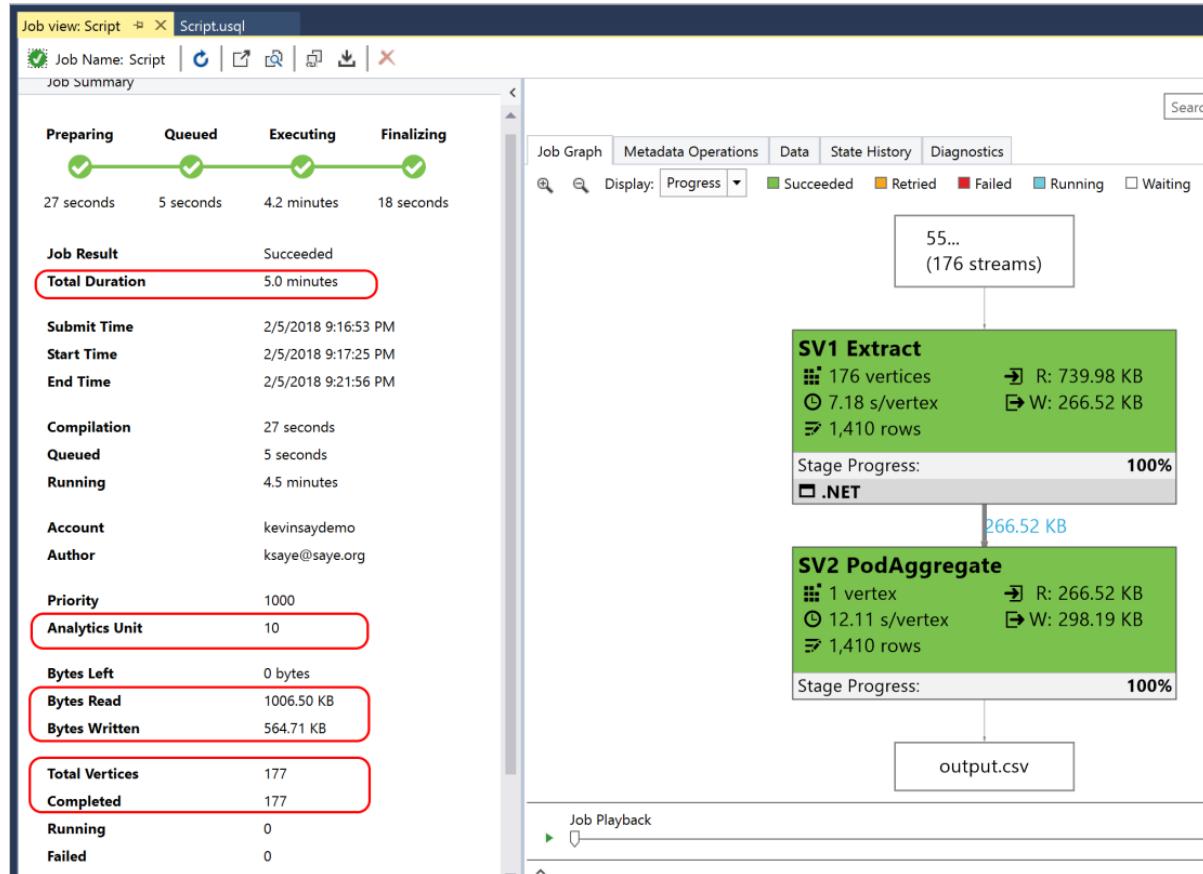
@cnt =
SELECT EnqueuedTimeUtc AS time, Encoding.UTF8.GetString(Body) AS jsonmessage
FROM @rs;

OUTPUT @cnt TO @output_file USING Outputters.Text();

```

It took Data Lake Analytics five minutes to run the following script, which was limited to 10 analytic units

and processed 177 files. The result is shown in the CSV-file output that's displayed in the following image:



File preview	
output.csv	
0	1
2018-02-04T20:35:30.8440000Z	{ "message": "TempUpdate:HotTub Air:62.78", "event": "TempUpdate", "object": "HotTub Air", "status": "62.78", "host": "HotTub.inte
2018-02-05T07:42:58.3050000Z	{ "message": "TempUpdate:UpStairs:57.20 (53.74)", "event": "TempUpdate", "object": "UpStairs", "status": "57.20 (53.74)", "host": "Up
2018-02-05T10:43:22.1210000Z	{ "message": "TempUpdate:UpStairs:57.56 (54.02)", "event": "TempUpdate", "object": "UpStairs", "status": "57.56 (54.02)", "host": "Up
2018-02-04T18:21:20.8590000Z	{ "message": "TempUpdate:UpStairs:62.60 (59.19)", "event": "TempUpdate", "object": "UpStairs", "status": "62.60 (59.19)", "host": "Up
2018-02-06T01:54:27.2300000Z	{ "message": "TempUpdate:HotTub Air:50.90", "event": "TempUpdate", "object": "HotTub Air", "status": "50.90", "host": "HotTub.inte

To parse the JSON, continue to step 8.

8. Most IoT messages are in JSON file format. By adding the following lines, you can parse the message into a JSON file, which lets you add the WHERE clauses and output only the needed data.

```

@jsonify =
    SELECT
Microsoft.Analytics.Samples.Formats.Json.JsonFunctions.JsonTuple(Encoding.UTF8.GetString(Body))
    AS message FROM @rs;

/*
@cnt =
    SELECT EnqueuedTimeUtc AS time, Encoding.UTF8.GetString(Body) AS jsonmessage
    FROM @rs;

OUTPUT @cnt TO @output_file USING Outputters.Text();

*/
@cnt =
    SELECT message["message"] AS iotmessage,
           message["event"] AS msgevent,
           message["object"] AS msgobject,
           message["status"] AS msgstatus,
           message["host"] AS msghost
    FROM @jsonify;

OUTPUT @cnt TO @output_file USING Outputters.Text();

```

The output displays a column for each item in the `SELECT` command.

File preview output.csv				
	Format	Download	Rename file	Access
0	1	2	3	4
TempUpdate:HotTub Water:52.70	TempUpdate	HotTub Water	52.70	HotTub.internal.saye.org
TempUpdate:HotTub Air:55.76	TempUpdate	HotTub Air	55.76	HotTub.internal.saye.org
TempUpdate:HotTub Water:52.81	TempUpdate	HotTub Water	52.81	HotTub.internal.saye.org
TempUpdate:Garage:64.76	TempUpdate	Garage	64.76	garageDoor.internal.saye.org

Next steps

In this tutorial, you learned how to query Avro data to efficiently route messages from Azure IoT Hub to Azure services.

For examples of complete end-to-end solutions that use IoT Hub, see the [Azure IoT Solution Accelerators Documentation](#).

To learn more about developing solutions with IoT Hub, see the [IoT Hub developer guide](#).

To learn more about message routing in IoT Hub, see [Send and receive messages with IoT Hub](#).

Order device connection events from Azure IoT Hub using Azure Cosmos DB

3/6/2019 • 8 minutes to read

Azure Event Grid helps you build event-based applications and easily integrate IoT events in your business solutions. This article walks you through a setup which can be used to track and store the latest device connection state in Cosmos DB. We will use the sequence number available in the Device Connected and Device Disconnected events and store the latest state in Cosmos DB. We are going to use a stored procedure, which is an application logic that is executed against a collection in Cosmos DB.

The sequence number is a string representation of a hexadecimal number. You can use string compare to identify the larger number. If you are converting the string to hex, then the number will be a 256-bit number. The sequence number is strictly increasing, and the latest event will have a higher number than other events. This is useful if you have frequent device connects and disconnects, and want to ensure only the latest event is used to trigger a downstream action, as Azure Event Grid doesn't support ordering of events.

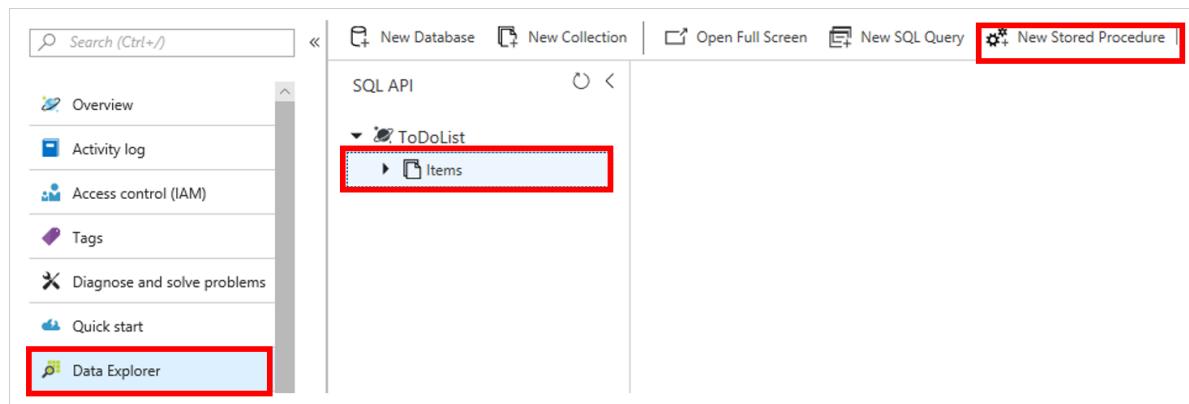
Prerequisites

- An active Azure account. If you don't have one, you can [create a free account](#).
- An active Azure Cosmos DB SQL API account. If you haven't created one yet, see [Create a database account](#) for a walkthrough.
- A collection in your database. See [Add a collection](#) for a walkthrough.
- An IoT Hub in Azure. If you haven't created one yet, see [Get started with IoT Hub](#) for a walkthrough.

Create a stored procedure

First, create a stored procedure and set it up to run a logic that compares sequence numbers of incoming events and records the latest event per device in the database.

1. In your Cosmos DB SQL API, select **Data Explorer** > **Items** > **New Stored Procedure**.



2. Enter a stored procedure ID and paste the following in the "Stored Procedure body". Note that this code should replace any existing code in the stored procedure body. This code maintains one row per device ID and records the latest connection state of that device ID by identifying the highest sequence number.

```
// SAMPLE STORED PROCEDURE
function UpdateDevice(deviceId, moduleId, hubName, connectionState, connectionStateUpdatedTime,
```

```

sequenceNumber) {
  var collection = getContext().getCollection();
  var response = {};

  var docLink = getDocumentLink(deviceId, moduleId);

  var isAccepted = collection.readDocument(docLink, function(err, doc) {
    if (err) {
      console.log('Cannot find device ' + docLink + ' - ');
      createDocument();
    } else {
      console.log('Document Found - ');
      replaceDocument(doc);
    }
  });
}

function replaceDocument(document) {
  console.log(
    'Old Seq : ' +
    document.sequenceNumber +
    ' New Seq: ' +
    sequenceNumber +
    ' - '
  );
  if (sequenceNumber > document.sequenceNumber) {
    document.connectionState = connectionState;
    document.connectionStateUpdatedTime = connectionStateUpdatedTime;
    document.sequenceNumber = sequenceNumber;

    console.log('replace doc - ');

    isAccepted = collection.replaceDocument(docLink, document, function(
      err,
      updated
    ) {
      if (err) {
        getContext()
          .getResponse()
          .setBody(err);
      } else {
        getContext()
          .getResponse()
          .setBody(updated);
      }
    });
  } else {
    getContext()
      .getResponse()
      .setBody('Old Event - current: ' + document.sequenceNumber + ' Incoming: ' + sequenceNumber);
  }
}

function createDocument() {
  document = {
    id: deviceId + '-' + moduleId,
    deviceId: deviceId,
    moduleId: moduleId,
    hubName: hubName,
    connectionState: connectionState,
    connectionStateUpdatedTime: connectionStateUpdatedTime,
    sequenceNumber: sequenceNumber
  };
  console.log('Add new device - ' + collection.getAltLink());
  isAccepted = collection.createDocument(
    collection.getAltLink(),
    document,
    function(err, doc) {
      if (err) {
        getContext()
          .getResponse()

```

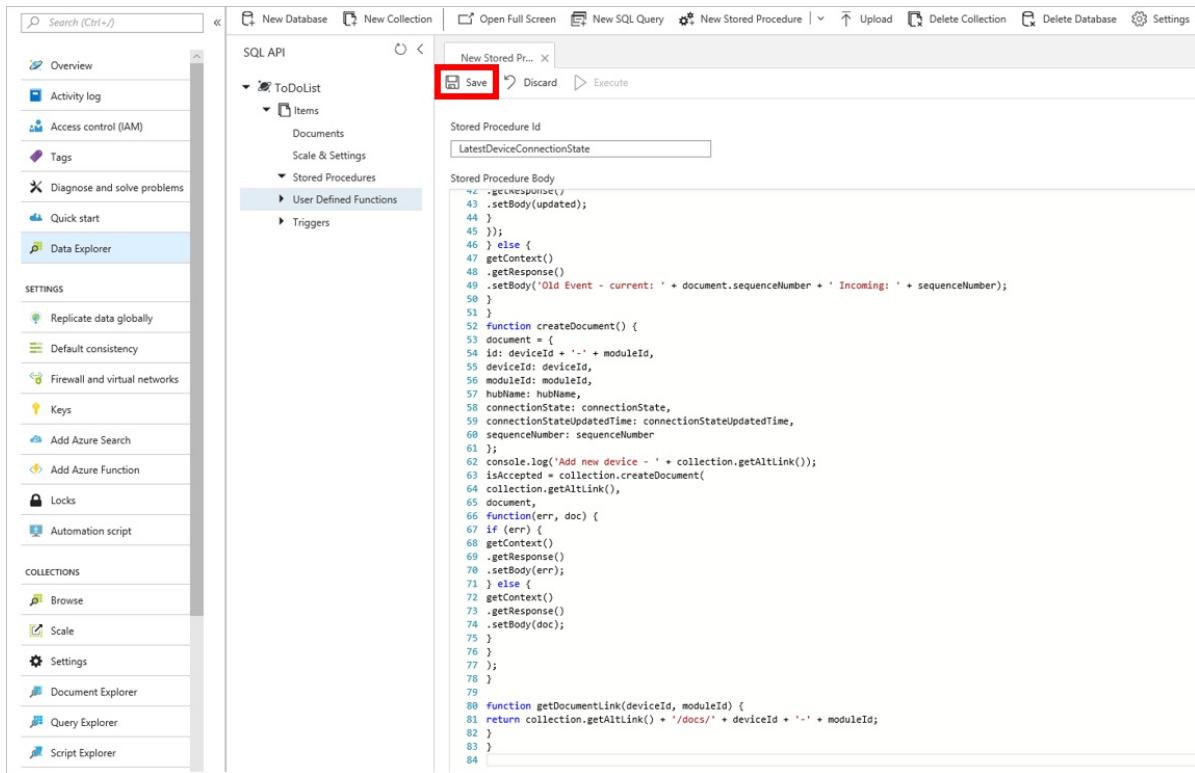
```

        .setBody(err);
    } else {
        getContext()
        .getResponse()
        .setBody(doc);
    }
}

function getDocumentLink(deviceId, moduleId) {
    return collection.getAltLink() + '/docs/' + deviceId + '-' + moduleId;
}
}

```

3. Save the stored procedure:

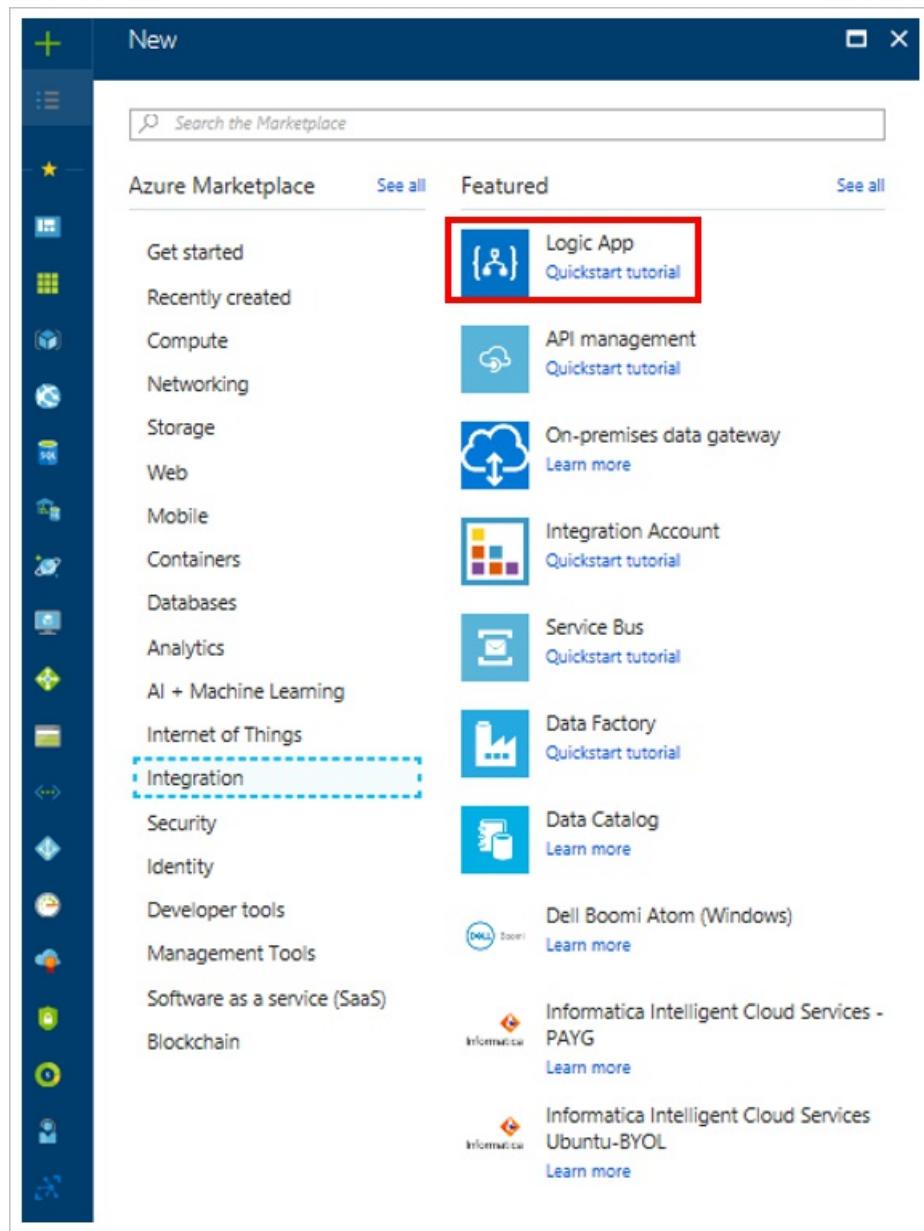


Create a logic app

First, create a logic app and add an Event grid trigger that monitors the resource group for your virtual machine.

Create a logic app resource

- In the [Azure portal](#), select **New > Integration > Logic App**.



2. Give your logic app a name that's unique in your subscription, then select the same subscription, resource group, and location as your IoT hub.
3. Select **Pin to dashboard**, then choose **Create**.

You've now created an Azure resource for your logic app. After Azure deploys your logic app, the Logic Apps Designer shows you templates for common patterns so you can get started faster.

NOTE

When you select **Pin to dashboard**, your logic app automatically opens in the Logic Apps Designer. Otherwise, you can manually find and open your logic app.

4. In the Logic App Designer under **Templates**, choose **Blank Logic App** so that you can build your logic app from scratch.

Select a trigger

A trigger is a specific event that starts your logic app. For this tutorial, the trigger that sets off the workflow is receiving a request over HTTP.

1. In the connectors and triggers search bar, type **HTTP**.

2. Select **Request - When an HTTP request is received** as the trigger.

The screenshot shows the Microsoft Power Automate search interface. A search bar at the top contains the text 'http'. Below it, under the heading 'Connectors', there are several icons and names: Request (blue globe icon), Azure Kusto (purple cube icon), Content Moderator (blue exclamation mark icon), HTTP (green globe icon), HTTP with Azure AD (blue globe icon with a gear), Instagram (blue camera icon), and Nexmo (blue phone icon). Below this, two more connectors are listed: Pitney Bowes Data (blue people icon) and Plumsail Documents (blue gear icon). At the bottom of the connector section is a 'See more' link. Under the heading 'Triggers (4)', there are four items: 'HTTP' (green globe icon), 'HTTP + Swagger' (green curly braces icon), 'HTTP Webhook' (green gear icon), and 'Request' (blue globe icon). The 'Request' item is highlighted with a red rectangle. To its right is a 'See more' link. Below the triggers is a section titled 'TELL US WHAT YOU NEED' with a 'Help us decide which connectors and triggers to add next with UserVoice' button featuring a smiley face icon.

3. Select **Use sample payload to generate schema**.

The screenshot shows the configuration for the 'When a HTTP request is received' trigger. At the top, the trigger name is displayed. Below it, the 'HTTP POST URL' field contains the placeholder 'URL will be generated after save'. To the right of this field is a small '...' button and a copy icon. Underneath is the 'Request Body JSON Schema' section, which is currently empty. At the bottom of the configuration screen is a red-bordered button labeled 'Use sample payload to generate schema'. Below this button is another button labeled 'Show advanced options' with a dropdown arrow.

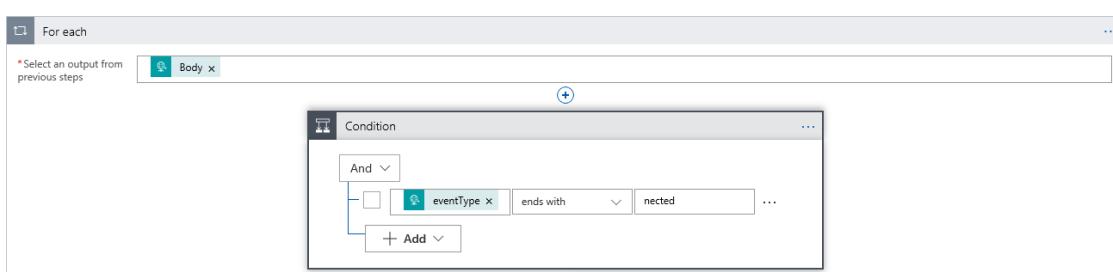
4. Paste the following sample JSON code into the text box, then select **Done**:

5. You may receive a pop-up notification that says, **Remember to include a Content-Type header set to application/json in your request.** You can safely ignore this suggestion, and move on to the next section.

Create a condition

In your logic app workflow, conditions help run specific actions after passing that specific condition. Once the condition is met, a desired action can be defined. For this tutorial, the condition is to check whether eventType is device connected or device disconnected. The action will be to execute the stored procedure in your database.

1. Select **New step** then **Built-ins** and **Condition**.
 2. Fill the condition as shown below to only execute this for Device Connected and Device Disconnected events:
 - Choose a value: **eventType**
 - Change "is equal to" to **ends with**
 - Choose a value: **nected**



3. If the condition is true, click on **Add an action**.



4. Search for Cosmos DB and click on **Azure Cosmos DB - Execute stored procedure**

The screenshot shows the Logic Apps Designer interface with the 'Cosmos DB' connector selected. Under the 'Actions (11)' tab, the 'Execute stored procedure' action is highlighted with a red box. Other actions listed include Create or update document, Create stored procedure, Delete a document, Delete stored procedure, Get a document, Get all documents, and Get stored procedures.

5. Populate the form for Execute stored procure by selecting values from your database. Enter the partition key value and parameters as shown below.

The screenshot shows the 'Execute stored procedure (Preview)' configuration form. It includes fields for Database ID (selected as 'ToDoList'), Collection ID (selected as 'Items'), and Sproc ID (selected as 'LatestDeviceConnectionState'). The 'Partition key value' field contains '[data.deviceId, data.moduleId]'. The 'Parameters for the stored procedure' field contains '["data.deviceId", "data.moduleId", "data.hubName", "eventType", "eventTime", "data.deviceCon..."]'.

6. Save your logic app.

Copy the HTTP URL

Before you leave the Logic Apps Designer, copy the URL that your logic app is listening to for a trigger. You use this URL to configure Event Grid.

1. Expand the **When a HTTP request is received** trigger configuration box by clicking on it.
2. Copy the value of **HTTP POST URL** by selecting the copy button next to it.

The screenshot shows the Logic App designer interface. A trigger named "When a HTTP request is received" is selected. The "HTTP POST URL" field contains the copied URL from the previous step. The "Request Body JSON Schema" pane shows a partial schema definition:

```
{"type": "array", "items": {}}
```

3. Save this URL so that you can refer to it in the next section.

Configure subscription for IoT Hub events

In this section, you configure your IoT Hub to publish events as they occur.

1. In the Azure portal, navigate to your IoT hub.

2. Select **Events**.

The screenshot shows the IoT Hub navigation menu. The "Events" option is highlighted with a red box.

3. Select **Event subscription**.

The screenshot shows the "Event subscription" blade. The "Event Subscription" button is highlighted with a red box.

4. Create the event subscription with the following values:

- **Event Type:** Uncheck Subscribe to all event types and select **Device Connected** and **Device Disconnected** from the menu.
- **Endpoint Details:** Select Endpoint Type as **Web Hook** and click on select endpoint and paste the URL that you copied from your logic app and confirm selection.

The screenshot shows the "ENDPOINT DETAILS" blade. The "Endpoint Type" is set to "Web Hook (change)". The "Endpoint" field has a placeholder "Select an endpoint" with a red box around it.

- **Event Subscription Details:** Provide a descriptive name and select **Event Grid Schema**. The form looks similar to the following example:

 Create Event Subscription

Create Advanced Editor

Event Subscriptions listen for events emitted by the topic resource and send them to the endpoint resource. [Learn more](#)

TOPIC DETAILS

Pick a topic resource for which events should be generated and pushed. [Learn more](#)

Topic Type  IoT Hub
Topic Resource [myIoTHub](#)

EVENT TYPES

Pick which event types get pushed to your destination. [Learn more](#)

Subscribe to all event types

Defined Event Types [2 selected](#)

ENDPOINT DETAILS

Pick an event handler to receive your events. [Learn more](#)

Endpoint Type  Web Hook ([change](#))
Endpoint [Select an endpoint](#)

EVENT SUBSCRIPTION DETAILS

Name ✓
Event Schema [Event Grid Schema](#)

FILTERS

Subject Begins With
Subject Ends With
Labels  Add Label

Create

5. Select **Create** to save the event subscription.

Observe events

Now that your event subscription is set up, let's test by connecting a device.

Register a device in IoT Hub

1. From your IoT hub, select **IoT Devices**.
2. Select **Add**.
3. For **Device ID**, enter .
4. Select **Save**.
5. You can add multiple devices with different device IDs.

6. Copy the **Connection string -- primary key** for later use.

Device Details
Demo-Device-1

Device Id: Demo-Device-1

Primary key: O18aPNmLOZtoM3Hh3G6/4pxlnm8XL60NugEFHhNPiy8=

Secondary key: DeziZ3FluhkrnwYnb6C29vPZECpN1dARdIKoMQJLdl=

Connection string—primary key: HostName=myIoTHub.azure-devices.net;DeviceId=Demo-Device-1;SharedAccessKey=O18aPNmLOZtoM3Hh3G6/4pxlnm8...

Connection string—secondary key: HostName=myIoTHub.azure-devices.net;DeviceId=Demo-Device-1;SharedAccessKey=DeziZ3FluhkrnwYnb6C29vPZECpN1...

Start Raspberry Pi simulator

1. Let's use the Raspberry Pi web simulator to simulate device connection.

[Start Raspberry Pi simulator](#)

Run a sample application on the Raspberry Pi web simulator

This will trigger a device connected event.

1. In the coding area, replace the placeholder in Line 15 with your Azure IoT Hub device connection string.

```

14
15 const connectionString = '[Your IoT hub device connection string]';
16 const LEDPin = 4;
17

```

2. Run the application by clicking on **Run**.

You should see the following output that shows the sensor data and the messages that are sent to your IoT hub.

Run **Reset**

Click 'Run' button to run the sample code(When sample is running, code is read-only).
Click 'Stop' button to stop the sample code running.
Click 'Reset' to reset the code.We keep your changes to the editor even you refresh the page.

>
Sending message: {"messageId":1,"deviceId":"Raspberry Pi Web Client","temperature":25.813850468676115,"humidity":76.82965549699682}
>
Message sent to Azure IoT Hub
>
Sending message: {"messageId":2,"deviceId":"Raspberry Pi Web Client","temperature":23.690479538498596,"humidity":72.04969988562655}
>
Message sent to Azure IoT Hub

Click **Stop** to stop the simulator and trigger a **Device Disconnected** event.

You have now run a sample application to collect sensor data and send it to your IoT hub.

Observe events in Cosmos DB

You can see results of the executed stored procedure in your Cosmos DB document. Here's what it looks like. Each row contains the latest device connection state per device.

Use the Azure CLI

Instead of using the [Azure portal](#), you can accomplish the IoT Hub steps using the Azure CLI. For details, see the Azure CLI pages for [creating an event subscription](#) and [creating an IoT device](#).

Clean up resources

This tutorial used resources that incur charges on your Azure subscription. When you're done trying out the tutorial and testing your results, disable or delete resources that you don't want to keep.

If you don't want to lose the work on your logic app, disable it instead of deleting it.

1. Navigate to your logic app.
 2. On the **Overview** blade, select **Delete** or **Disable**.

Each subscription can have one free IoT hub. If you created a free hub for this tutorial, then you don't need to delete it to prevent charges.

1. Navigate to your IoT hub.
 2. On the **Overview** blade, select **Delete**.

Even if you keep your IoT hub, you may want to delete the event subscription that you created.

1. In your IoT hub, select **Event Grid**.
2. Select the event subscription that you want to remove.
3. Select **Delete**.

To remove an Azure Cosmos DB account from the Azure portal, right-click the account name and click **Delete account**. See detailed instructions for [deleting an Azure Cosmos DB account](#).

Next steps

- Learn more about [Reacting to IoT Hub events by using Event Grid to trigger actions](#)
- [Try the IoT Hub events tutorial](#)
- Learn about what else you can do with [Event Grid](#)

Send messages from the cloud to your device with IoT Hub (.NET)

2/28/2019 • 6 minutes to read

Introduction

Azure IoT Hub is a fully managed service that helps enable reliable and secure bi-directional communications between millions of devices and a solution back end. [Send telemetry from a device to an IoT hub...](#) shows how to create an IoT hub, provision a device identity in it, and code a device app that sends device-to-cloud messages.

NOTE

The features described in this article are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

This tutorial builds on the quickstart [Send telemetry from a device to an IoT hub...](#). It shows you how to do the following steps:

- From your solution back end, send cloud-to-device messages to a single device through IoT Hub.
- Receive cloud-to-device messages on a device.
- From your solution back end, request delivery acknowledgement (*feedback*) for messages sent to a device from IoT Hub.

You can find more information on cloud-to-device messages in [D2C and C2D Messaging with IoT Hub](#).

At the end of this tutorial, you run two .NET console apps:

- **SimulatedDevice**, a modified version of the app created in [Send telemetry from a device to an IoT hub...](#), which connects to your IoT hub and receives cloud-to-device messages.
- **SendCloudToDevice**, which sends a cloud-to-device message to the device app through IoT Hub, and then receives its delivery acknowledgement.

NOTE

IoT Hub has SDK support for many device platforms and languages (including C, Java, and Javascript) through [Azure IoT device SDKs](#). For step-by-step instructions on how to connect your device to this tutorial's code, and generally to Azure IoT Hub, see the [IoT Hub developer guide](#).

To complete this tutorial, you need the following:

- Visual Studio 2017
- An active Azure account. (If you don't have an account, you can create a [free account](#) in just a couple of minutes.)

Receive messages in the device app

In this section, you'll modify the device app you created in [Send telemetry from a device to an IoT hub...](#) to receive cloud-to-device messages from the IoT hub.

1. In Visual Studio, in the **SimulatedDevice** project, add the following method to the **Program** class.

```
private static async void ReceiveC2dAsync()
{
    Console.WriteLine("\nReceiving cloud to device messages from service");
    while (true)
    {
        Message receivedMessage = await deviceClient.ReceiveAsync();
        if (receivedMessage == null) continue;

        Console.ForegroundColor = ConsoleColor.Yellow;
        Console.WriteLine("Received message: {0}",
            Encoding.ASCII.GetString(receivedMessage.GetBytes()));
        Console.ResetColor();

        await deviceClient.CompleteAsync(receivedMessage);
    }
}
```

The `ReceiveAsync` method asynchronously returns the received message at the time that it is received by the device. It returns `null` after a specifiable timeout period (in this case, the default of one minute is used). When the app receives a `null`, it should continue to wait for new messages. This requirement is the reason for the `if (receivedMessage == null) continue` line.

The call to `completeAsync()` notifies IoT Hub that the message has been successfully processed. The message can be safely removed from the device queue. If something happened that prevented the device app from completing the processing of the message, IoT Hub delivers it again. It is then important that message processing logic in the device app is *idempotent*, so that receiving the same message multiple times produces the same result.

An application can also temporarily abandon a message, which results in IoT hub retaining the message in the queue for future consumption. Or the application can reject a message, which permanently removes the message from the queue. For more information about the cloud-to-device message lifecycle, see [D2C and C2D messaging with IoT Hub](#).

NOTE

When using HTTPS instead of MQTT or AMQP as a transport, the `ReceiveAsync` method returns immediately. The supported pattern for cloud-to-device messages with HTTPS is intermittently connected devices that check for messages infrequently (less than every 25 minutes). Issuing more HTTPS receives results in IoT Hub throttling the requests. For more information about the differences between MQTT, AMQP and HTTPS support, and IoT Hub throttling, see [D2C and C2D messaging with IoT Hub](#).

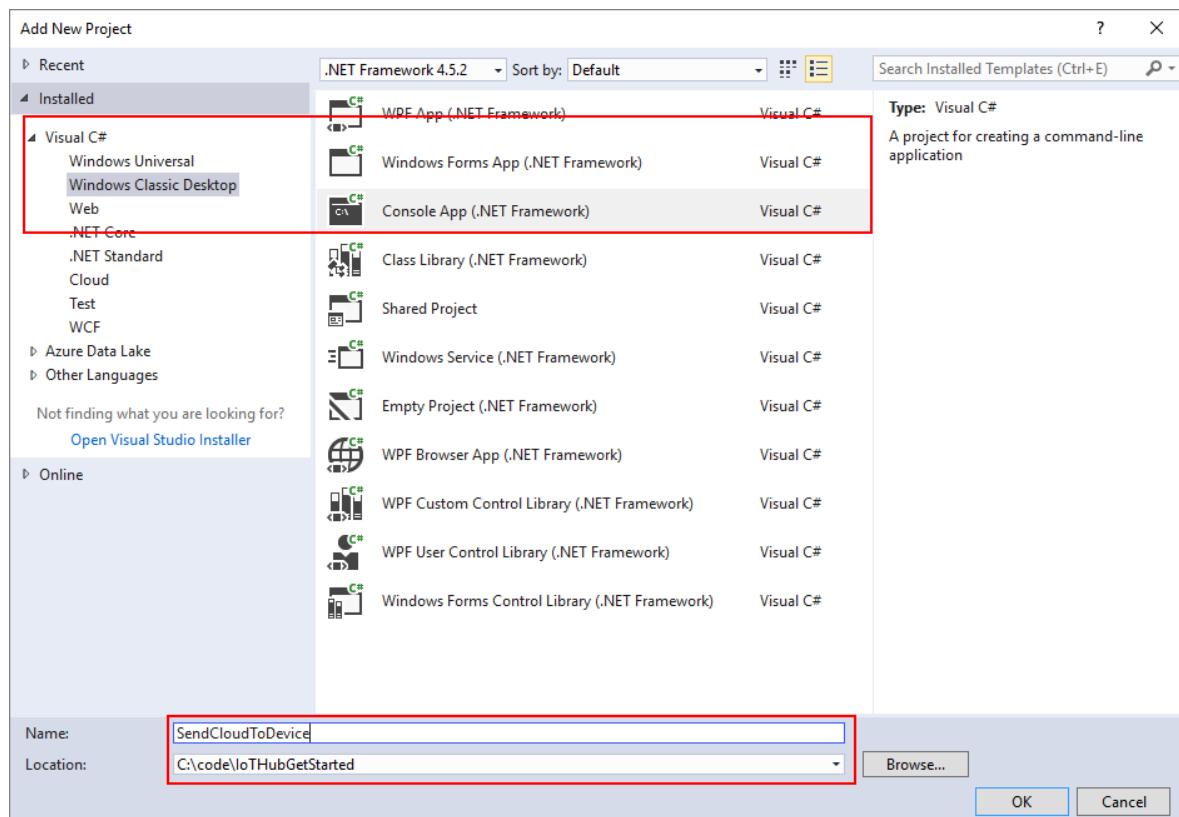
2. Add the following method in the **Main** method, right before the `Console.ReadLine()` line:

```
ReceiveC2dAsync();
```

Send a cloud-to-device message

In this section, you write a .NET console app that sends cloud-to-device messages to the device app.

1. In the current Visual Studio solution, create a Visual C# Desktop App project by using the **Console Application** project template. Name the project **SendCloudToDevice**.



2. In Solution Explorer, right-click the solution, and then click **Manage NuGet Packages for Solution...**.

This action opens the **Manage NuGet Packages** window.

3. Search for **Microsoft.Azure.Devices**, click **Install**, and accept the terms of use.

This downloads, installs, and adds a reference to the [Azure IoT service SDK NuGet package](#).

4. Add the following `using` statement at the top of the **Program.cs** file:

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

5. Add the following fields to the **Program** class. Substitute the placeholder value with the IoT hub connection string from [Send telemetry from a device to an IoT hub...](#):

```
static ServiceClient serviceClient;
static string connectionString = "{iot hub connection string}";
```

6. Add the following method to the **Program** class:

```
private async static Task SendCloudToDeviceMessageAsync()
{
    var commandMessage = new
        Message(Encoding.ASCII.GetBytes("Cloud to device message."));
    await serviceClient.SendAsync("myFirstDevice", commandMessage);
}
```

This method sends a new cloud-to-device message to the device with the ID, `myFirstDevice`. Change this parameter only if you modified it from the one used in [Send telemetry from a device to an IoT hub...](#).

7. Finally, add the following lines to the **Main** method:

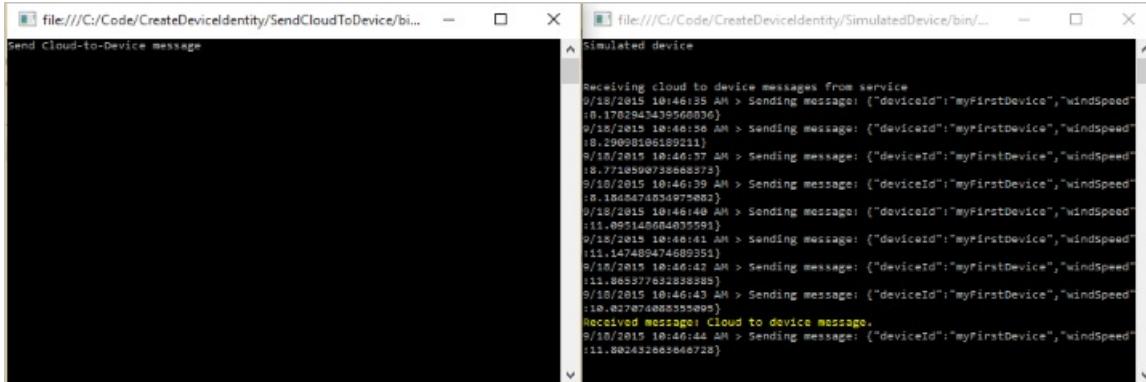
```

Console.WriteLine("Send Cloud-to-Device message\n");
serviceClient = ServiceClient.CreateFromConnectionString(connectionString);

Console.WriteLine("Press any key to send a C2D message.");
Console.ReadLine();
SendCloudToDeviceMessageAsync().Wait();
Console.ReadLine();

```

8. From within Visual Studio, right-click your solution, and select **Set StartUp projects....** Select **Multiple startup projects**, then select the **Start** action for **ReadDeviceToCloudMessages**, **SimulatedDevice**, and **SendCloudToDevice**.
9. Press **F5**. All three applications should start. Select the **SendCloudToDevice** windows, and press **Enter**. You should see the message being received by the device app.



Receive delivery feedback

It is possible to request delivery (or expiration) acknowledgements from IoT Hub for each cloud-to-device message. This option enables the solution back end to easily inform retry or compensation logic. For more information about cloud-to-device feedback, see [D2C and C2D Messaging with IoT Hub](#).

In this section, you modify the **SendCloudToDevice** app to request feedback, and receive it from the IoT hub.

1. In Visual Studio, in the **SendCloudToDevice** project, add the following method to the **Program** class.

```

private async static void ReceiveFeedbackAsync()
{
    var feedbackReceiver = serviceClient.GetFeedbackReceiver();

    Console.WriteLine("\nReceiving c2d feedback from service");
    while (true)
    {
        var feedbackBatch = await feedbackReceiver.ReceiveAsync();
        if (feedbackBatch == null) continue;

        Console.ForegroundColor = ConsoleColor.Yellow;
        Console.WriteLine("Received feedback: {0}",
            string.Join(", ", feedbackBatch.Records.Select(f => f.StatusCode)));
        Console.ResetColor();

        await feedbackReceiver.CompleteAsync(feedbackBatch);
    }
}

```

Note this receive pattern is the same one used to receive cloud-to-device messages from the device app.

2. Add the following method in the **Main** method, right after the

```
serviceClient = ServiceClient.CreateFromConnectionString(connectionString)
```

line:

```
ReceiveFeedbackAsync();
```

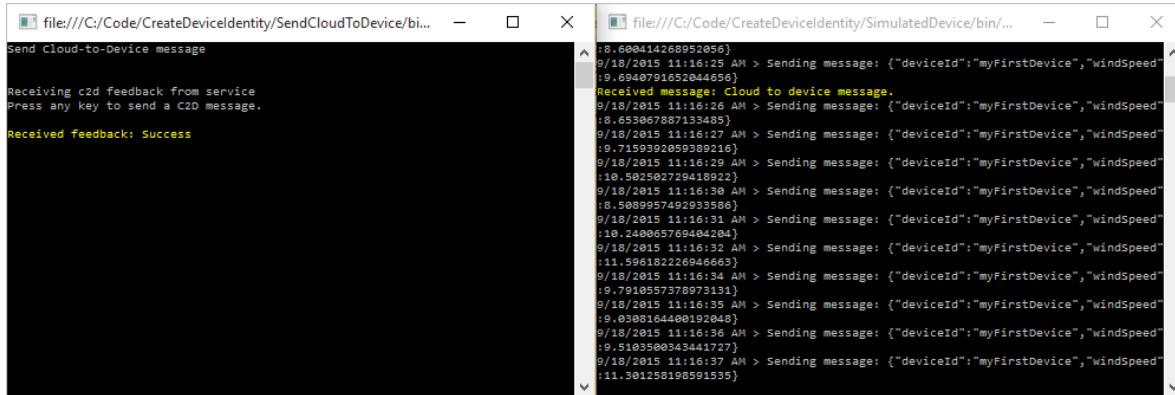
3. To request feedback for the delivery of your cloud-to-device message, you have to specify a property in the **SendCloudToDeviceMessageAsync** method. Add the following line, right after the

```
var commandMessage = new Message(...);
```

line:

```
commandMessage.Ack = DeliveryAcknowledgement.Full;
```

4. Run the apps by pressing **F5**. You should see all three applications start. Select the **SendCloudToDevice** windows, and press **Enter**. You should see the message being received by the device app, and after a few seconds, the feedback message being received by your **SendCloudToDevice** application.



NOTE

For simplicity's sake, this tutorial does not implement any retry policy. In production code, you should implement retry policies (such as exponential backoff), as suggested in the article, [Transient Fault Handling](#).

Next steps

In this how-to, you learned how to send and receive cloud-to-device messages.

To see examples of complete end-to-end solutions that use IoT Hub, see [Azure IoT Remote Monitoring solution accelerator](#).

To learn more about developing solutions with IoT Hub, see the [IoT Hub developer guide](#).

Send cloud-to-device messages with IoT Hub (Java)

3/6/2019 • 5 minutes to read

Azure IoT Hub is a fully managed service that helps enable reliable and secure bi-directional communications between millions of devices and a solution back end. The [Send telemetry from a device to a Hub \(Java\)](#) tutorial shows how to create an IoT hub, provision a device identity in it, and code a simulated device app that sends device-to-cloud messages.

NOTE

The features described in this article are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

This tutorial builds on [Send telemetry from a device to an IoT Hub \(Java\)](#). It shows you how to do the following:

- From your solution back end, send cloud-to-device messages to a single device through IoT Hub.
- Receive cloud-to-device messages on a device.
- From your solution back end, request delivery acknowledgement (*feedback*) for messages sent to a device from IoT Hub.

You can find more information on [cloud-to-device messages in the IoT Hub developer guide](#).

At the end of this tutorial, you run two Java console apps:

- **simulated-device**, a modified version of the app created in [Send telemetry from a device to a Hub \(Java\)](#), which connects to your IoT hub and receives cloud-to-device messages.
- **send-c2d-messages**, which sends a cloud-to-device message to the simulated device app through IoT Hub, and then receives its delivery acknowledgement.

NOTE

IoT Hub has SDK support for many device platforms and languages (including C, Java, and Javascript) through Azure IoT device SDKs. For step-by-step instructions on how to connect your device to this tutorial's code, and generally to Azure IoT Hub, see the [Azure IoT Developer Center](#).

To complete this tutorial, you need the following:

- A complete working version of the [Send telemetry from a device to a Hub \(Java\)](#) or the [Configure message routing with IoT Hub](#) tutorial.
- The latest [Java SE Development Kit 8](#)
- [Maven 3](#)
- An active Azure account. If you don't have an account, you can create a [free account](#) in just a couple of minutes.

Receive messages in the simulated device app

In this section, you modify the simulated device app you created in [Send telemetry from a device to a Hub \(Java\)](#)

to receive cloud-to-device messages from the IoT hub.

1. Using a text editor, open the simulated-device\src\main\java\com\mycompany\app\App.java file.
2. Add the following **MessageCallback** class as a nested class inside the **App** class. The **execute** method is invoked when the device receives a message from IoT Hub. In this example, the device always notifies the IoT hub that it has completed the message:

```
private static class AppMessageCallback implements MessageCallback {  
    public IoTHubMessageResult execute(Message msg, Object context) {  
        System.out.println("Received message from hub:  
            + new String(msg.getBytes(), Message.DEFAULT_IOTHUB_MESSAGE_CHARSET));  
  
        return IoTHubMessageResult.COMPLETE;  
    }  
}
```

3. Modify the **main** method to create an **AppMessageCallback** instance and call the **setMessageCallback** method before it opens the client as follows:

```
client = new DeviceClient(connString, protocol);  
  
MessageCallback callback = new AppMessageCallback();  
client.setMessageCallback(callback, null);  
client.open();
```

NOTE

If you use HTTPS instead of MQTT or AMQP as the transport, the **DeviceClient** instance checks for messages from IoT Hub infrequently (less than every 25 minutes). For more information about the differences between MQTT, AMQP and HTTPS support, and IoT Hub throttling, see the [messaging section of the IoT Hub developer guide](#).

4. To build the **simulated-device** app using Maven, execute the following command at the command prompt in the simulated-device folder:

```
mvn clean package -DskipTests
```

Send a cloud-to-device message

In this section, you create a Java console app that sends cloud-to-device messages to the simulated device app. You need the device ID of the device you added in the [Send telemetry from a device to a Hub \(Java\)](#) quickstart. You also need the IoT Hub connection string for your hub that you can find in the [Azure portal](#).

1. Create a Maven project called **send-c2d-messages** using the following command at your command prompt. Note this command is a single, long command:

```
mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=send-c2d-messages -  
DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

2. At your command prompt, navigate to the new send-c2d-messages folder.
3. Using a text editor, open the pom.xml file in the send-c2d-messages folder and add the following dependency to the **dependencies** node. Adding the dependency enables you to use the **iothub-java-service-client** package in your application to communicate with your IoT hub service:

```
<dependency>
<groupId>com.microsoft.azure.sdk.iot</groupId>
<artifactId>iot-service-client</artifactId>
<version>1.7.23</version>
</dependency>
```

NOTE

You can check for the latest version of **iot-service-client** using [Maven search](#).

4. Save and close the pom.xml file.
5. Using a text editor, open the send-c2d-messages\src\main\java\com\mycompany\app\App.java file.
6. Add the following **import** statements to the file:

```
import com.microsoft.azure.sdk.iot.service.*;
import java.io.IOException;
import java.net.URISyntaxException;
```

7. Add the following class-level variables to the **App** class, replacing **{yourhubconnectionstring}** and **{yourdeviceid}** with the values you noted earlier:

```
private static final String connectionString = "{yourhubconnectionstring}";
private static final String deviceId = "{yourdeviceid}";
private static final IoTHubServiceClientProtocol protocol =
    IoTHubServiceClientProtocol.AMQPS;
```

8. Replace the **main** method with the following code. This code connects to your IoT hub, sends a message to your device, and then waits for an acknowledgment that the device received and processed the message:

```
public static void main(String[] args) throws IOException,
    URISyntaxException, Exception {
    ServiceClient serviceClient = ServiceClient.createFromConnectionString(
        connectionString, protocol);

    if (serviceClient != null) {
        serviceClient.open();
        FeedbackReceiver feedbackReceiver = serviceClient
            .getFeedbackReceiver();
        if (feedbackReceiver != null) feedbackReceiver.open();

        Message messageToSend = new Message("Cloud to device message.");
        messageToSend.setDeliveryAcknowledgement(DeliveryAcknowledgement.Full);

        serviceClient.send(deviceId, messageToSend);
        System.out.println("Message sent to device");

        FeedbackBatch feedbackBatch = feedbackReceiver.receive(10000);
        if (feedbackBatch != null) {
            System.out.println("Message feedback received, feedback time: "
                + feedbackBatch.getEnqueuedTimeUtc().toString());
        }

        if (feedbackReceiver != null) feedbackReceiver.close();
        serviceClient.close();
    }
}
```

NOTE

For simplicity's sake, this tutorial does not implement any retry policy. In production code, you should implement retry policies (such as exponential backoff), as suggested in the article, [Transient Fault Handling](#).

9. To build the **simulated-device** app using Maven, execute the following command at the command prompt in the simulated-device folder:

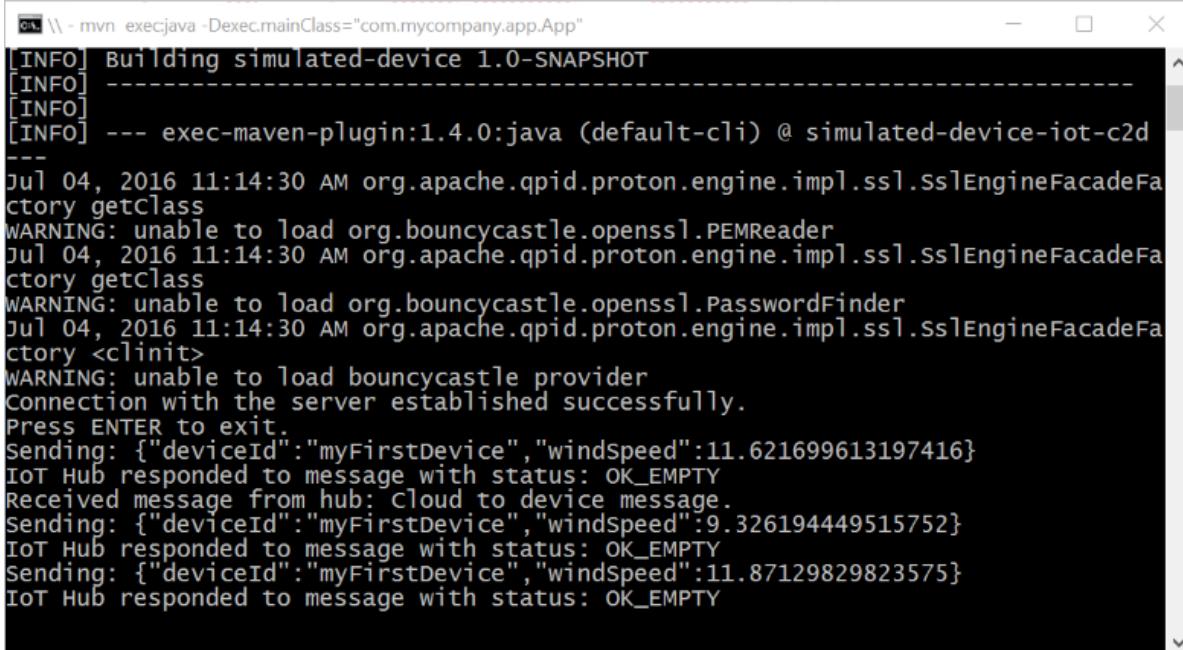
```
mvn clean package -DskipTests
```

Run the applications

You are now ready to run the applications.

1. At a command prompt in the simulated-device folder, run the following command to begin sending telemetry to your IoT hub and to listen for cloud-to-device messages sent from your hub:

```
mvn exec:java -Dexec.mainClass="com.mycompany.app.App"
```



```
mvn exec:java -Dexec.mainClass="com.mycompany.app.App"
[INFO] Building simulated-device 1.0-SNAPSHOT
[INFO] -----
[INFO] --- exec-maven-plugin:1.4.0:java (default-cli) @ simulated-device-iot-c2d
---
Jul 04, 2016 11:14:30 AM org.apache.qpid.proton.engine.impl.ssl.SslEngineFacadeFactory getClass
WARNING: unable to load org.bouncycastle.openssl.PEMReader
Jul 04, 2016 11:14:30 AM org.apache.qpid.proton.engine.impl.ssl.SslEngineFacadeFactory getClass
WARNING: unable to load org.bouncycastle.openssl.PasswordFinder
Jul 04, 2016 11:14:30 AM org.apache.qpid.proton.engine.impl.ssl.SslEngineFacadeFactory <clinit>
WARNING: unable to load bouncycastle provider
Connection with the server established successfully.
Press ENTER to exit.
Sending: {"deviceId": "myFirstDevice", "windspeed": 11.621699613197416}
IoT Hub responded to message with status: OK_EMPTY
Received message from hub: Cloud to device message.
Sending: {"deviceId": "myFirstDevice", "windSpeed": 9.326194449515752}
IoT Hub responded to message with status: OK_EMPTY
Sending: {"deviceId": "myFirstDevice", "windSpeed": 11.87129829823575}
IoT Hub responded to message with status: OK_EMPTY
```

2. At a command prompt in the send-c2d-messages folder, run the following command to send a cloud-to-device message and wait for a feedback acknowledgment:

```
mvn exec:java -Dexec.mainClass="com.mycompany.app.App"
```

```
c:\\\n[INFO] [INFO] -----n[INFO] Building send-c2d-messages 1.0-SNAPSHOTn[INFO] -----n[INFO] --- exec-maven-plugin:1.4.0:java (default-cli) @ send-c2d-messages ---nJul 04, 2016 11:14:40 AM org.apache.qpid.proton.engine.impl.ssl.SslEngineFactory getClassnWARNING: unable to load org.bouncycastle.openssl.PEMReadernJul 04, 2016 11:14:40 AM org.apache.qpid.proton.engine.impl.ssl.SslEngineFactory getClassnWARNING: unable to load org.bouncycastle.openssl.PasswordFindernJul 04, 2016 11:14:40 AM org.apache.qpid.proton.engine.impl.ssl.SslEngineFactory <clinit>nWARNING: unable to load bouncycastle providernMessage sent to devicenMessage feedback received, feedback time: 2016-07-04T10:14:41.958093400Zn[INFO] -----n[INFO] BUILD SUCCESSn[INFO] -----n[INFO] Total time: 3.732 sn[INFO] Finished at: 2016-07-04T11:14:42+01:00n[INFO] Final Memory: 13M/346Mn[INFO] -----n:c:\repos\samples\JavaIoTC2D\send-c2d-messages>
```

Next steps

In this tutorial, you learned how to send and receive cloud-to-device messages.

To see examples of complete end-to-end solutions that use IoT Hub, see [Azure IoT Solution Accelerators](#).

To learn more about developing solutions with IoT Hub, see the [IoT Hub developer guide](#).

Send cloud-to-device messages with IoT Hub (Node)

3/6/2019 • 5 minutes to read

Introduction

Azure IoT Hub is a fully managed service that helps enable reliable and secure bi-directional communications between millions of devices and a solution back end. The [Get started with IoT Hub](#) tutorial shows how to create an IoT hub, provision a device identity in it, and code a simulated device app that sends device-to-cloud messages.

NOTE

The features described in this article are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

This tutorial builds on [Get started with IoT Hub](#). It shows you how to:

- From your solution back end, send cloud-to-device messages to a single device through IoT Hub.
- Receive cloud-to-device messages on a device.
- From your solution back end, request delivery acknowledgement (*feedback*) for messages sent to a device from IoT Hub.

You can find more information on cloud-to-device messages in the [IoT Hub developer guide](#).

At the end of this tutorial, you run two Node.js console apps:

- **SimulatedDevice**, a modified version of the app created in [Get started with IoT Hub](#), which connects to your IoT hub and receives cloud-to-device messages.
- **SendCloudToDeviceMessage**, which sends a cloud-to-device message to the simulated device app through IoT Hub, and then receives its delivery acknowledgement.

NOTE

IoT Hub has SDK support for many device platforms and languages (including C, Java, and Javascript) through Azure IoT device SDKs. For step-by-step instructions on how to connect your device to this tutorial's code, and generally to Azure IoT Hub, see the [Azure IoT Developer Center](#).

To complete this tutorial, you need the following:

- Node.js version 4.0.x or later.
- An active Azure account. (If you don't have an account, you can create a [free account](#) in just a couple of minutes.)

Receive messages in the simulated device app

In this section, you modify the simulated device app you created in [Get started with IoT Hub](#) to receive cloud-to-device messages from the IoT hub.

1. Using a text editor, open the `SimulatedDevice.js` file.
2. Modify the **connectCallback** function to handle messages sent from IoT Hub. In this example, the device always invokes the **complete** function to notify IoT Hub that it has processed the message. Your new version of the **connectCallback** function looks like the following snippet:

```

var connectCallback = function (err) {
    if (err) {
        console.log('Could not connect: ' + err);
    } else {
        console.log('Client connected');
        client.on('message', function (msg) {
            console.log('Id: ' + msg.messageId + ' Body: ' + msg.data);
            client.complete(msg, printResultFor('completed'));
        });
        // Create a message and send it to the IoT Hub every second
        setInterval(function(){
            var temperature = 20 + (Math.random() * 15);
            var humidity = 60 + (Math.random() * 20);
            var data = JSON.stringify({ deviceId: 'myFirstNodeDevice', temperature: temperature, humidity: humidity });
            var message = new Message(data);
            message.properties.add('temperatureAlert', (temperature > 30) ? 'true' : 'false');
            console.log("Sending message: " + message.getData());
            client.sendEvent(message, printResultFor('send'));
        }, 1000);
    }
};

```

NOTE

If you use HTTPS instead of MQTT or AMQP as the transport, the **DeviceClient** instance checks for messages from IoT Hub infrequently (less than every 25 minutes). For more information about the differences between MQTT, AMQP and HTTPS support, and IoT Hub throttling, see the [IoT Hub developer guide](#).

Send a cloud-to-device message

In this section, you create a Node.js console app that sends cloud-to-device messages to the simulated device app. You need the device ID of the device you added in the [Get started with IoT Hub](#) tutorial. You also need the IoT Hub connection string for your hub that you can find in the [Azure portal](#).

1. Create an empty folder called **sendcloudtodevicemessage**. In the **sendcloudtodevicemessage** folder, create a package.json file using the following command at your command prompt. Accept all the defaults:

```
npm init
```

2. At your command prompt in the **sendcloudtodevicemessage** folder, run the following command to install the **azure-iothub** package:

```
npm install azure-iothub --save
```

3. Using a text editor, create a **SendCloudToDeviceMessage.js** file in the **sendcloudtodevicemessage** folder.
4. Add the following `require` statements at the start of the **SendCloudToDeviceMessage.js** file:

```
'use strict';

var Client = require('azure-iothub').Client;
var Message = require('azure-iot-common').Message;
```

5. Add the following code to **SendCloudToDeviceMessage.js** file. Replace the "{iot hub connection string}" placeholder value with the IoT Hub connection string for the hub you created in the [Get started with IoT](#)

Hub tutorial. Replace the "{device id}" placeholder with the device ID of the device you added in the [Get started with IoT Hub](#) tutorial:

```
var connectionString = '{iot hub connection string}';  
var targetDevice = '{device id}';  
  
var serviceClient = Client.fromConnectionString(connectionString);
```

6. Add the following function to print operation results to the console:

```
function printResultFor(op) {  
    return function printResult(err, res) {  
        if (err) console.log(op + ' error: ' + err.toString());  
        if (res) console.log(op + ' status: ' + res.constructor.name);  
    };  
}
```

7. Add the following function to print delivery feedback messages to the console:

```
function receiveFeedback(err, receiver){  
    receiver.on('message', function (msg) {  
        console.log('Feedback message: ')  
        console.log(msg.getData().toString('utf-8'));  
    });  
}
```

8. Add the following code to send a message to your device and handle the feedback message when the device acknowledges the cloud-to-device message:

```
serviceClient.open(function (err) {  
    if (err) {  
        console.error('Could not connect: ' + err.message);  
    } else {  
        console.log('Service client connected');  
        serviceClient.getFeedbackReceiver(receiveFeedback);  
        var message = new Message('Cloud to device message.');//  
        message.ack = 'full';  
        message.messageId = "My Message ID";  
        console.log('Sending message: ' + message.getData());  
        serviceClient.send(targetDevice, message, printResultFor('send'));  
    }  
});
```

9. Save and close **SendCloudToDeviceMessage.js** file.

Run the applications

You are now ready to run the applications.

1. At the command prompt in the **simulateddevice** folder, run the following command to send telemetry to IoT Hub and to listen for cloud-to-device messages:

```
node SimulatedDevice.js
```

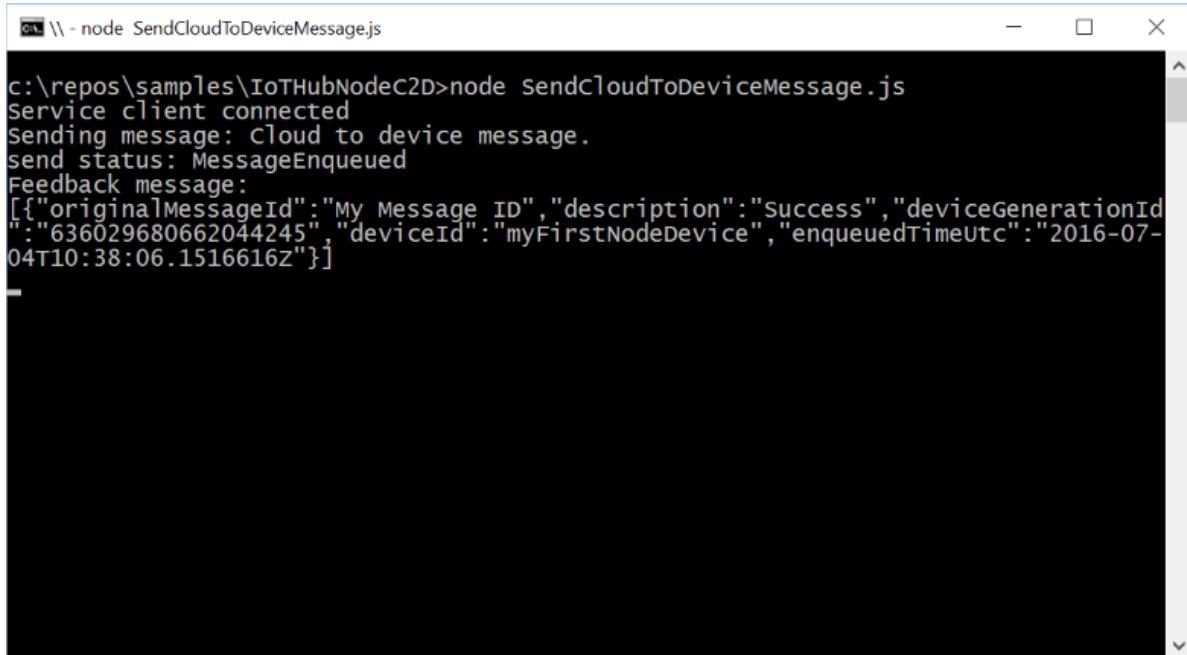


```
c:\repos\samples\IoTHubNodeC2D>node simulatedDevice.js
Client connected
Sending message: {"deviceId":"myFirstNodeDevice","windspeed":13.22213007429614}
send status: MessageEnqueued
Id: My Message ID Body: Cloud to device message.
completed status: MessageCompleted
Sending message: {"deviceId":"myFirstNodeDevice","windspeed":12.356118474293469}

send status: MessageEnqueued
Sending message: {"deviceId":"myFirstNodeDevice","windspeed":11.274137642401836}
send status: MessageEnqueued
```

- At a command prompt in the **sendcloudtodevicemessage** folder, run the following command to send a cloud-to-device message and wait for the acknowledgment feedback:

```
node SendCloudToDeviceMessage.js
```



```
c:\repos\samples\IoTHubNodeC2D>node sendCloudToDeviceMessage.js
Service client connected
Sending message: Cloud to device message.
send status: MessageEnqueued
Feedback message:
[{"originalMessageId": "My Message ID", "description": "Success", "deviceGenerationId": "636029680662044245", "deviceId": "myFirstNodeDevice", "enqueuedTimeUtc": "2016-07-04T10:38:06.1516616Z"}]
```

NOTE

For simplicity's sake, this tutorial does not implement any retry policy. In production code, you should implement retry policies (such as exponential backoff), as suggested in the article, [Transient Fault Handling](#).

Next steps

In this tutorial, you learned how to send and receive cloud-to-device messages.

To see examples of complete end-to-end solutions that use IoT Hub, see [Azure IoT Remote Monitoring solution accelerator](#).

To learn more about developing solutions with IoT Hub, see the [IoT Hub developer guide](#).

Send cloud-to-device messages with IoT Hub (Python)

3/6/2019 • 6 minutes to read

Introduction

Azure IoT Hub is a fully managed service that helps enable reliable and secure bi-directional communications between millions of devices and a solution back end. The [Get started with IoT Hub](#) tutorial shows how to create an IoT hub, provision a device identity in it, and code a simulated device app that sends device-to-cloud messages.

NOTE

The features described in this article are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

This tutorial builds on [Get started with IoT Hub](#). It shows you how to:

- From your solution back end, send cloud-to-device messages to a single device through IoT Hub.
- Receive cloud-to-device messages on a device.
- From your solution back end, request delivery acknowledgement (*feedback*) for messages sent to a device from IoT Hub.

You can find more information on cloud-to-device messages in the [IoT Hub developer guide](#).

At the end of this tutorial, you run two Python console apps:

- **SimulatedDevice.py**, a modified version of the app created in [Get started with IoT Hub](#), which connects to your IoT hub and receives cloud-to-device messages.
- **SendCloudToDeviceMessage.py**, which sends a cloud-to-device message to the simulated device app through IoT Hub, and then receives its delivery acknowledgement.

NOTE

IoT Hub has SDK support for many device platforms and languages (including C, Java, and Javascript) through Azure IoT device SDKs. For step-by-step instructions on how to connect your device to this tutorial's code, and generally to Azure IoT Hub, see the [Azure IoT Developer Center](#).

To complete this tutorial, you need the following:

- [Python 2.x or 3.x](#). Make sure to use the 32-bit or 64-bit installation as required by your setup. When prompted during the installation, make sure to add Python to your platform-specific environment variable. If you are using Python 2.x, you may need to [install or upgrade pip, the Python package management system](#).
- If you are using Windows OS, then [Visual C++ redistributable package](#) to allow the use of native DLLs from Python.
- An active Azure account. (If you don't have an account, you can create a [free account](#) in just a couple of minutes.)

NOTE

The `pip` packages for `azure-iothub-service-client` and `azure-iothub-device-client` are currently available only for Windows OS. For Linux/Mac OS, please refer to the Linux and Mac OS-specific sections on the [Prepare your development environment for Python](#) post.

Receive messages in the simulated device app

In this section, you create a Python console app to simulate the device and receive cloud-to-device messages from the IoT hub.

1. Using a text editor, create a **SimulatedDevice.py** file.
2. Add the following `import` statements and variables at the start of the **SimulatedDevice.py** file:

```
import time
import sys
import iothub_client
from iothub_client import IoTHubClient, IoTHubClientError, IoTHubTransportProvider, IoTHubClientResult
from iothub_client import IoTHubMessage, IoTHubMessageDispositionResult, IoTHubError

RECEIVE_CONTEXT = 0
WAIT_COUNT = 10
RECEIVED_COUNT = 0
RECEIVE_CALLBACKS = 0
```

3. Add the following code to **SimulatedDevice.py** file. Replace the "`{deviceConnectionString}`" placeholder value with the device connection string for the device you created in the [Get started with IoT Hub](#) tutorial:

```
# choose AMQP or AMQP_WS as transport protocol
PROTOCOL = IoTHubTransportProvider.AMQP
CONNECTION_STRING = "{deviceConnectionString}"
```

4. Add the following function to print received messages to the console:

```

def receive_message_callback(message, counter):
    global RECEIVE_CALLBACKS
    message_buffer = message.get_bytearray()
    size = len(message_buffer)
    print ( "Received Message [%d]: " % counter )
    print ( "    Data: <<%s>> & Size=%d" % (message_buffer[:size].decode('utf-8'), size) )
    map_properties = message.properties()
    key_value_pair = map_properties.get_internals()
    print ( "    Properties: %s" % key_value_pair )
    counter += 1
    RECEIVE_CALLBACKS += 1
    print ( "    Total calls received: %d" % RECEIVE_CALLBACKS )
    return IoTHubMessageDispositionResult.ACCEPTED

def iothub_client_init():
    client = IoTHubClient(CONNECTION_STRING, PROTOCOL)

    client.set_message_callback(receive_message_callback, RECEIVE_CONTEXT)

    return client

def print_last_message_time(client):
    try:
        last_message = client.get_last_message_receive_time()
        print ( "Last Message: %s" % time.asctime(time.localtime(last_message)) )
        print ( "Actual time : %s" % time.asctime() )
    except IoTHubClientError as iothub_client_error:
        if iothub_client_error.args[0].result == IoTHubClientResult.INDEFINITE_TIME:
            print ( "No message received" )
        else:
            print ( iothub_client_error )

```

5. Add the following code to initialize the client and wait to receive the cloud-to-device message:

```

def iothub_client_init():
    client = IoTHubClient(CONNECTION_STRING, PROTOCOL)

    client.set_message_callback(receive_message_callback, RECEIVE_CONTEXT)

    return client

def iothub_client_sample_run():
    try:
        client = iothub_client_init()

        while True:
            print ( "IoTHubClient waiting for commands, press Ctrl-C to exit" )

            status_counter = 0
            while status_counter <= WAIT_COUNT:
                status = client.get_send_status()
                print ( "Send status: %s" % status )
                time.sleep(10)
                status_counter += 1

    except IoTHubError as iothub_error:
        print ( "Unexpected error %s from IoTHub" % iothub_error )
        return
    except KeyboardInterrupt:
        print ( "IoTHubClient sample stopped" )

    print_last_message_time(client)

```

6. Add the following main function:

```

if __name__ == '__main__':
    print ( "Starting the IoT Hub Python sample..." )
    print ( "    Protocol %s" % PROTOCOL )
    print ( "    Connection string=%s" % CONNECTION_STRING )

    iothub_client_sample_run()

```

7. Save and close **SimulatedDevice.py** file.

Send a cloud-to-device message

In this section, you create a Python console app that sends cloud-to-device messages to the simulated device app. You need the device ID of the device you added in the [Get started with IoT Hub](#) tutorial. You also need the IoT Hub connection string for your hub that you can find in the [Azure portal](#).

1. Using a text editor, create a **SendCloudToDeviceMessage.py** file.
2. Add the following `import` statements and variables at the start of the **SendCloudToDeviceMessage.py** file:

```

import random
import sys
import iothub_service_client
from iothub_service_client import IoTHubMessaging, IoTHubMessage, IoTHubError

OPEN_CONTEXT = 0
FEEDBACK_CONTEXT = 1
MESSAGE_COUNT = 1
AVG_WIND_SPEED = 10.0
MSG_TXT = "{\"service client sent a message\": %.2f}"

```

3. Add the following code to **SendCloudToDeviceMessage.py** file. Replace the "{IoHubConnectionString}" placeholder value with the IoT Hub connection string for the hub you created in the [Get started with IoT Hub](#) tutorial. Replace the "{deviceId}" placeholder with the device ID of the device you added in the [Get started with IoT Hub](#) tutorial:

```

CONNECTION_STRING = "{IoHubConnectionString}"
DEVICE_ID = "{deviceId}"

```

4. Add the following function to print feedback messages to the console:

```

def open_complete_callback(context):
    print ( 'open_complete_callback called with context: {0}'.format(context) )

def send_complete_callback(context, messaging_result):
    context = 0
    print ( 'send_complete_callback called with context : {0}'.format(context) )
    print ( 'messagingResult : {0}'.format(messaging_result) )

```

5. Add the following code to send a message to your device and handle the feedback message when the device acknowledges the cloud-to-device message:

```

def iothub.messaging_sample_run():
    try:
        iothub.messaging = IoTHubMessaging(CONNECTION_STRING)

        iothub.messaging.open(open_complete_callback, OPEN_CONTEXT)

        for i in range(0, MESSAGE_COUNT):
            print ('Sending message: {0}'.format(i))
            msg_txt_formatted = MSG_TXT % (AVG_WIND_SPEED + (random.random() * 4 + 2))
            message = IoTHubMessage(bytearray(msg_txt_formatted, 'utf8'))

            # optional: assign ids
            message.message_id = "message_%d" % i
            message.correlation_id = "correlation_%d" % i
            # optional: assign properties
            prop_map = message.properties()
            prop_text = "PropMsg_%d" % i
            prop_map.add("Property", prop_text)

            iothub.messaging.send_async(DEVICE_ID, message, send_complete_callback, i)

        try:
            # Try Python 2.xx first
            raw_input("Press Enter to continue...\n")
        except:
            pass
            # Use Python 3.xx in the case of exception
            input("Press Enter to continue...\n")

        iothub.messaging.close()

    except IoTHubError as iothub_error:
        print ("Unexpected error {0}" % iothub_error)
        return
    except KeyboardInterrupt:
        print ("IoTHubMessaging sample stopped")

```

6. Add the following main function:

```

if __name__ == '__main__':
    print ( "Starting the IoT Hub Service Client Messaging Python sample..." )
    print ( "    Connection string = {0}".format(CONNECTION_STRING) )
    print ( "    Device ID       = {0}".format(DEVICE_ID) )

    iothub.messaging_sample_run()

```

7. Save and close **SendCloudToDeviceMessage.py** file.

Run the applications

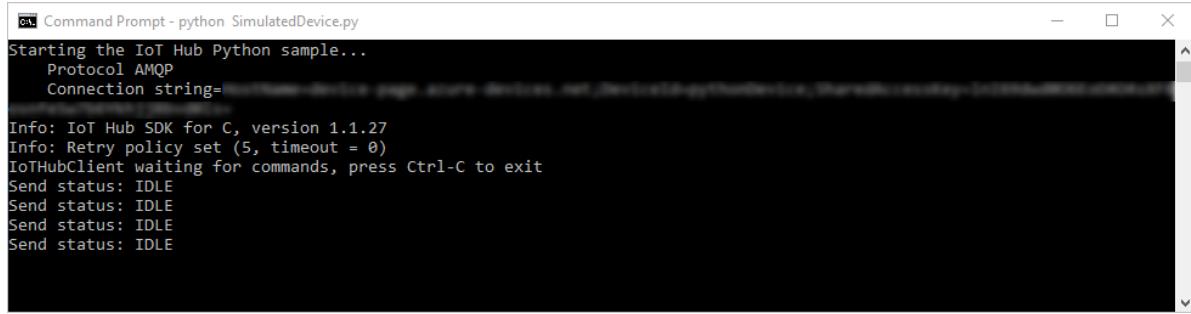
You are now ready to run the applications.

1. Open a command prompt and install the **Azure IoT Hub Device SDK for Python**.

```
pip install azure-iothub-device-client
```

2. At the command prompt, run the following command to listen for cloud-to-device messages:

```
python SimulatedDevice.py
```



```
Command Prompt - python SimulatedDevice.py
Starting the IoT Hub Python sample...
Protocol AMQP
Connection string=REDACTED

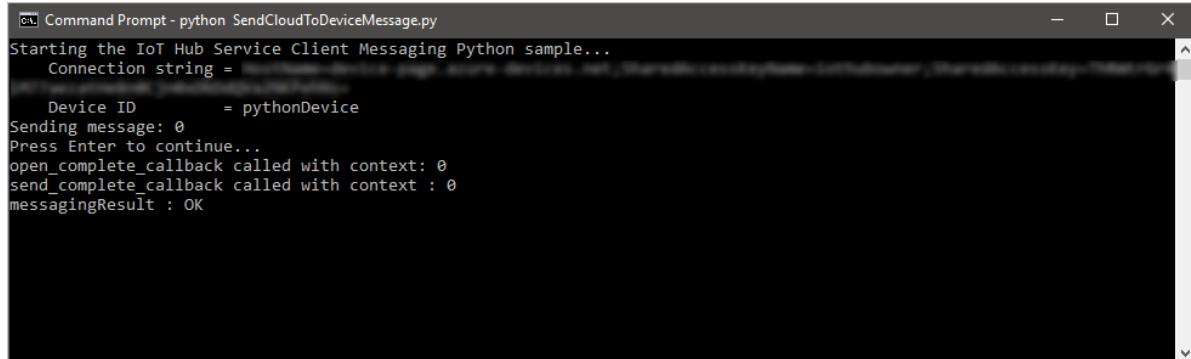
Info: IoT Hub SDK for C, version 1.1.27
Info: Retry policy set (5, timeout = 0)
IoTHubClient waiting for commands, press Ctrl-C to exit
Send status: IDLE
Send status: IDLE
Send status: IDLE
Send status: IDLE
```

3. Open a new command prompt and install the **Azure IoT Hub Service SDK for Python**.

```
pip install azure-iothub-service-client
```

4. At a command prompt, run the following command to send a cloud-to-device message and wait for the message feedback:

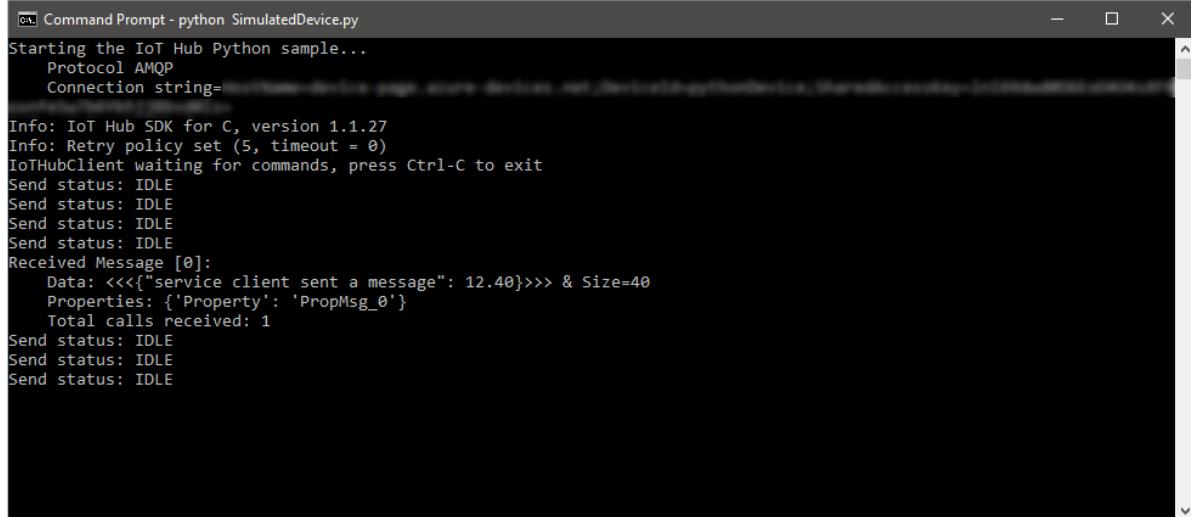
```
python SendCloudToDeviceMessage.py
```



```
Command Prompt - python SendCloudToDeviceMessage.py
Starting the IoT Hub Service Client Messaging Python sample...
Connection string = REDACTED

Device ID      = pythonDevice
Sending message: 0
Press Enter to continue...
open_complete_callback called with context: 0
send_complete_callback called with context : 0
messagingResult : OK
```

5. Note the message received by the device.



```
Command Prompt - python SimulatedDevice.py
Starting the IoT Hub Python sample...
Protocol AMQP
Connection string=REDACTED

Info: IoT Hub SDK for C, version 1.1.27
Info: Retry policy set (5, timeout = 0)
IoTHubClient waiting for commands, press Ctrl-C to exit
Send status: IDLE
Send status: IDLE
Send status: IDLE
Send status: IDLE
Received Message [0]:
  Data: <<<{"service client sent a message": 12.40}>>> & Size=40
  Properties: {'Property': 'PropMsg_0'}
  Total calls received: 1
Send status: IDLE
Send status: IDLE
Send status: IDLE
```

Next steps

In this tutorial, you learned how to send and receive cloud-to-device messages.

To see examples of complete end-to-end solutions that use IoT Hub, see [Azure IoT Remote Monitoring solution accelerator](#).

To learn more about developing solutions with IoT Hub, see the [IoT Hub developer guide](#).

Send cloud-to-device messages with IoT Hub (iOS)

3/6/2019 • 5 minutes to read

Azure IoT Hub is a fully managed service that helps enable reliable and secure bi-directional communications between millions of devices and a solution back end. The [Send telemetry from a device to an IoT hub](#) article shows how to create an IoT hub, provision a device identity in it, and code a simulated device app that sends device-to-cloud messages.

This article shows you how to:

- From your solution back end, send cloud-to-device messages to a single device through IoT Hub.
- Receive cloud-to-device messages on a device.
- From your solution back end, request delivery acknowledgement (*feedback*) for messages sent to a device from IoT Hub.

You can find more information on cloud-to-device messages in the [messaging section of the IoT Hub developer guide](#).

At the end of this article, you run two Swift iOS projects:

- **sample-device**, the same app created in [Send telemetry from a device to an IoT hub](#), which connects to your IoT hub and receives cloud-to-device messages.
- **sample-service**, which sends a cloud-to-device message to the simulated device app through IoT Hub, and then receives its delivery acknowledgement.

NOTE

IoT Hub has SDK support for many device platforms and languages (including C, Java, and Javascript) through Azure IoT device SDKs. For step-by-step instructions on how to connect your device to this tutorial's code, and generally to Azure IoT Hub, see the [Azure IoT Developer Center](#).

To complete this tutorial, you need the following:

- An active Azure account. (If you don't have an account, you can create a [free account](#) in just a couple of minutes.)
- An active IoT hub in Azure.
- The code sample from [Azure samples](#) .
- The latest version of [XCode](#), running the latest version of the iOS SDK. This quickstart was tested with XCode 9.3 and iOS 11.3.
- The latest version of [CocoaPods](#).

Simulate an IoT device

In this section, you simulate an iOS device running a Swift application to receive cloud-to-device messages from the IoT hub.

This is the sample device that you create in the article [Send telemetry from a device to an IoT hub](#). If you already have that running, you can skip this section.

Install CocoaPods

CocoaPods manage dependencies for iOS projects that use third-party libraries.

In a terminal window, navigate to the Azure-IoT-Samples-iOS folder that you downloaded in the prerequisites. Then, navigate to the sample project:

```
cd quickstart/sample-device
```

Make sure that XCode is closed, then run the following command to install the CocoaPods that are declared in the **podfile** file:

```
pod install
```

Along with installing the pods required for your project, the installation command also created an XCode workspace file that is already configured to use the pods for dependencies.

Run the sample device application

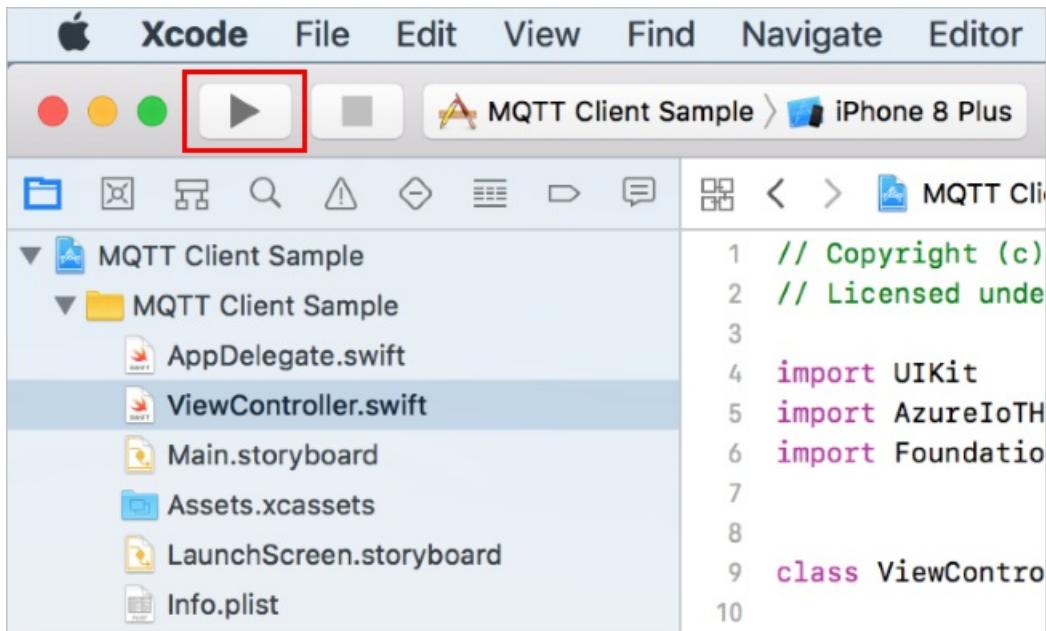
1. Retrieve the connection string for your device. You can copy this string from the [Azure portal](#) in the device details blade, or retrieve it with the following CLI command:

```
az iot hub device-identity show-connection-string --hub-name {YourIoTHubName} --device-id {YourDeviceID} --output table
```

2. Open the sample workspace in XCode.

```
open "MQTT Client Sample.xcworkspace"
```

3. Expand the **MQTT Client Sample** project and then folder of the same name.
4. Open **ViewController.swift** for editing in XCode.
5. Search for the **connectionString** variable and update the value with the device connection string that you copied in the first step.
6. Save your changes.
7. Run the project in the device emulator with the **Build and run** button or the key combo **command + r**.



Simulate a service device

In this section, you simulate a second iOS device with a Swift app that sends cloud-to-device messages through the IoT hub. This configuration is useful for IoT scenarios where there is one iPhone or iPad functioning as a controller for other iOS devices connected to an IoT hub.

Install CocoaPods

CocoaPods manage dependencies for iOS projects that use third-party libraries.

Navigate to the Azure IoT iOS Samples folder that you downloaded in the prerequisites. Then, navigate to the sample service project:

```
cd quickstart/sample-service
```

Make sure that XCode is closed, then run the following command to install the CocoaPods that are declared in the **podfile** file:

```
pod install
```

Along with installing the pods required for your project, the installation command also created an XCode workspace file that is already configured to use the pods for dependencies.

Run the sample service application

1. Retrieve the service connection string for your IoT hub. You can copy this string from the [Azure portal](#) from the **iothubowner** policy in the **Shared access policies** blade, or retrieve it with the following CLI command:

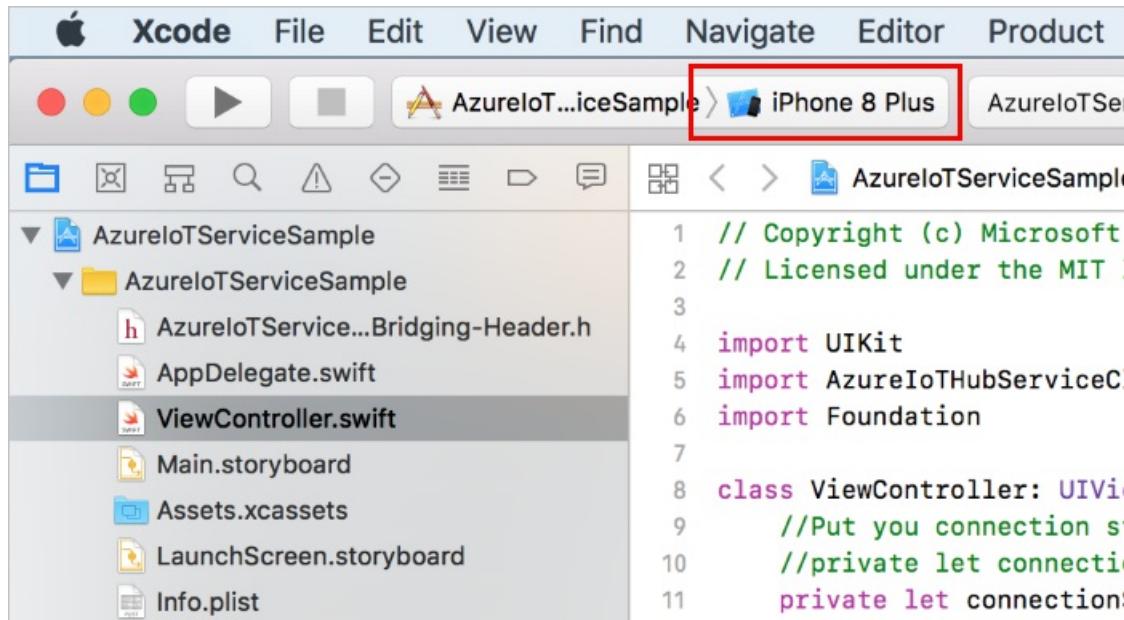
```
az iot hub show-connection-string --hub-name {YourIoTHubName} --output table
```

2. Open the sample workspace in XCode.

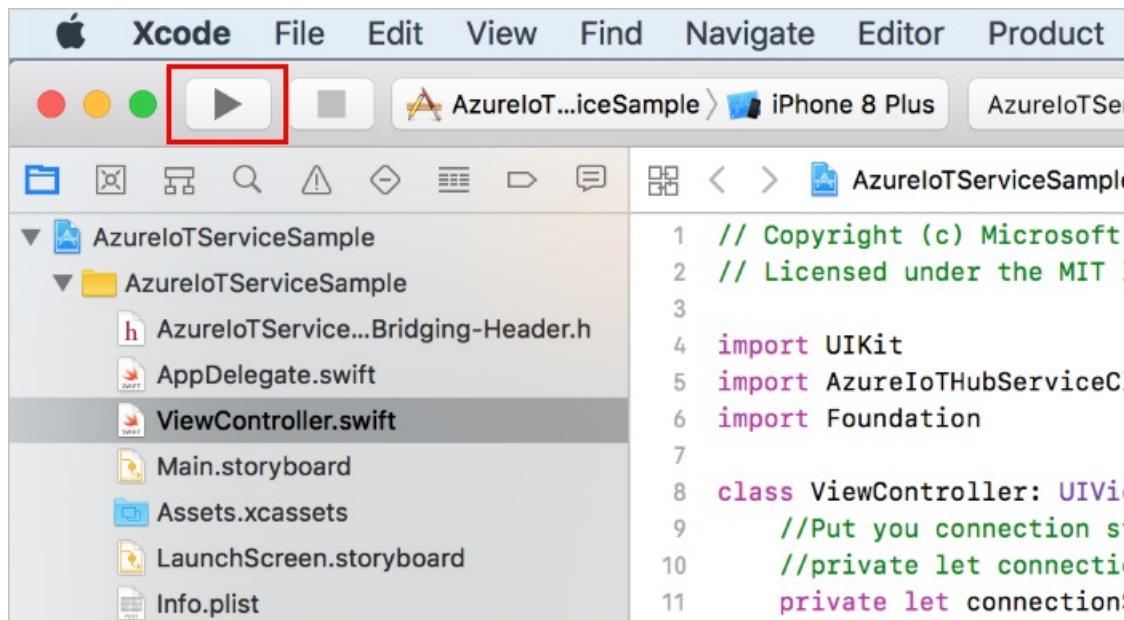
```
open AzureIoTServiceSample.xcworkspace
```

3. Expand the **AzureIoTServiceSample** project and then expand the folder of the same name.

4. Open **ViewController.swift** for editing in XCode.
5. Search for the **connectionString** variable and update the value with the service connection string that you copied previously.
6. Save your changes.
7. In Xcode, change the emulator settings to a different iOS device than you used to run the IoT device. XCode cannot run multiple emulators of the same type.



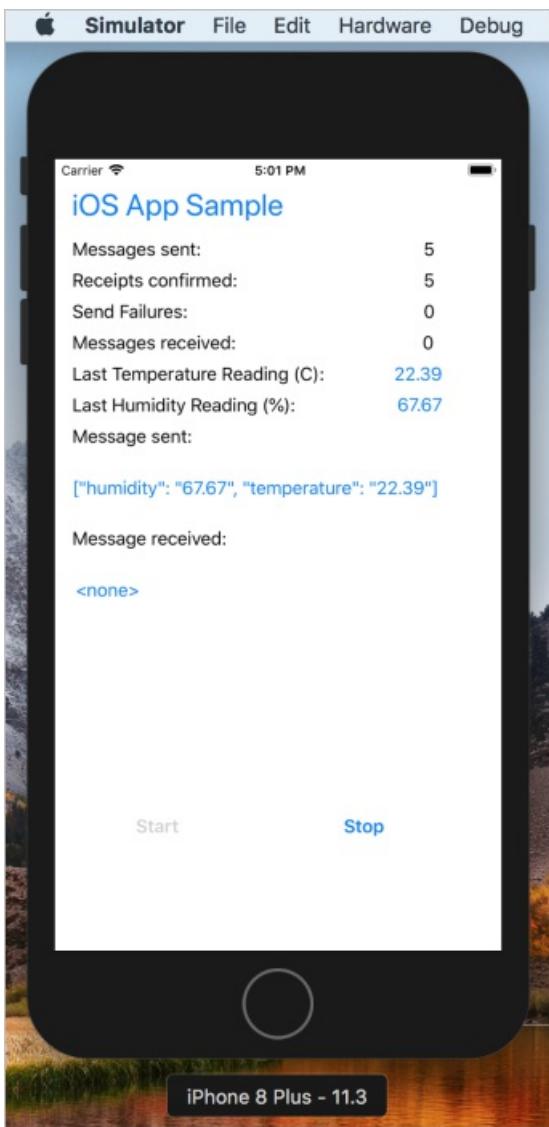
8. Run the project in the device emulator with the **Build and run** button or the key combo **Command + r**.



Send a cloud-to-device message

You are now ready to use the two applications to send and receive cloud-to-device messages.

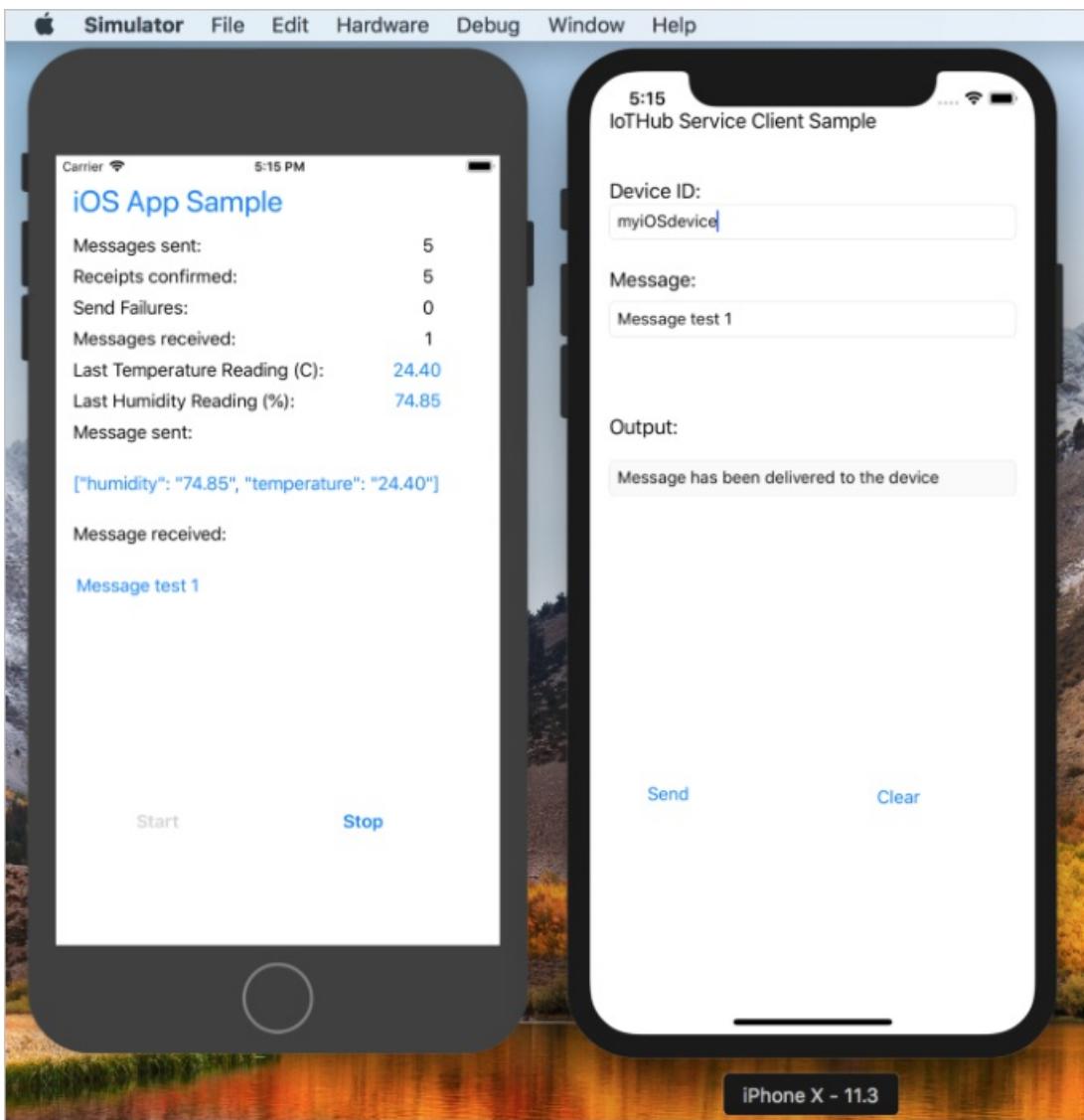
1. In the **iOS App Sample** app running on the simulated IoT device, click **Start**. The application starts sending device-to-cloud messages, but also starts listening for cloud-to-device messages.



2. In the **IoTHub Service Client Sample** app running on the simulated service device, enter the ID for the IoT device that you want to send a message to.
3. Write a plaintext message, then click **Send**.

Several actions happen as soon as you click send. The service sample sends the message to your IoT hub, which the app has access to because of the service connection string that you provided. Your IoT hub checks the device ID, sends the message to the destination device, and sends a confirmation receipt to the source device. The app running on your simulated IoT device checks for messages from IoT Hub and prints the text from the most recent one on the screen.

Your output should look like the following example:



Next steps

In this tutorial, you learned how to send and receive cloud-to-device messages.

To see examples of complete end-to-end solutions that use IoT Hub, see the [Azure IoT Solution Accelerators](#) documentation.

To learn more about developing solutions with IoT Hub, see the [IoT Hub developer guide](#).

Upload files from your device to the cloud with IoT Hub using .NET

2/28/2019 • 5 minutes to read

This tutorial builds on the code in the [Send Cloud-to-Device messages with IoT Hub](#) tutorial to show you how to use the file upload capabilities of IoT Hub. It shows you how to:

- Securely provide a device with an Azure blob URI for uploading a file.
- Use the IoT Hub file upload notifications to trigger processing the file in your app back end.

The [Send telemetry from a device to an IoT hub](#) and [Send Cloud-to-Device messages with IoT Hub](#) articles show the basic device-to-cloud and cloud-to-device messaging functionality of IoT Hub. The [Configure Message Routing with IoT Hub](#) tutorial describes a way to reliably store device-to-cloud messages in Azure blob storage. However, in some scenarios you cannot easily map the data your devices send into the relatively small device-to-cloud messages that IoT Hub accepts. For example:

- Large files that contain images
- Videos
- Vibration data sampled at high frequency
- Some form of preprocessed data

These files are typically batch processed in the cloud using tools such as [Azure Data Factory](#) or the [Hadoop](#) stack. When you need to upload files from a device, you can still use the security and reliability of IoT Hub.

At the end of this tutorial you run two .NET console apps:

- **SimulatedDevice**, a modified version of the app created in the [Send Cloud-to-Device messages with IoT Hub](#) tutorial. This app uploads a file to storage using a SAS URI provided by your IoT hub.
- **ReadFileUploadNotification**, which receives file upload notifications from your IoT hub.

NOTE

IoT Hub supports many device platforms and languages (including C, Java, and Javascript) through Azure IoT device SDKs. Refer to the [Azure IoT Developer Center](#) for step-by-step instructions on how to connect your device to Azure IoT Hub.

To complete this tutorial, you need the following:

- Visual Studio 2017
- An active Azure account. (If you don't have an account, you can create a [free account](#) in just a couple of minutes.)

Associate an Azure Storage account to IoT Hub

Because the simulated device app uploads a file to a blob, you must have an [Azure Storage](#) account associated with your IoT hub. When you associate an Azure Storage account with an IoT hub, the IoT hub generates a SAS URI. A device can use this SAS URI to securely upload a file to a blob container. The IoT Hub service and the device SDKs coordinate the process that generates the SAS URI and makes it available to a device to use to upload a file.

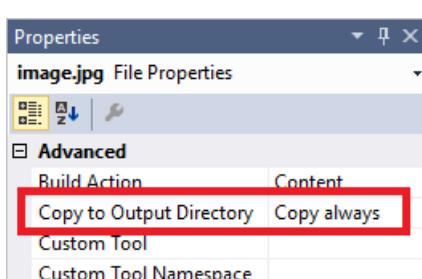
Follow the instructions in [Configure file uploads using the Azure portal](#). Make sure that a blob container is associated with your IoT hub and that file notifications are enabled.

The screenshot shows the 'File upload' settings for an IoT hub named 'iothubd'. On the left, there's a sidebar with links like Overview, Activity log, Access control (IAM), and various settings sections. The 'File upload' link is highlighted in blue. The main area has a 'Save' and 'Discard' button at the top. A red box highlights the 'Storage container' section, which lists 'iotdblob' and a toggle switch for 'Receive notifications for uploaded files' set to 'On'. Below this are sliders for 'SAS TTL' (set to 1 hr), 'Default TTL' (set to 1 hr), and 'Maximum delivery count' (set to 10).

Upload a file from a device app

In this section, you modify the device app you created in [Send Cloud-to-Device messages with IoT Hub](#) to receive cloud-to-device messages from the IoT hub.

1. In Visual Studio, right-click the **SimulatedDevice** project, click **Add**, and then click **Existing Item**. Navigate to an image file and include it in your project. This tutorial assumes the image is named `image.jpg`.
2. Right-click on the image, and then click **Properties**. Make sure that **Copy to Output Directory** is set to **Copy always**.



3. In the **Program.cs** file, add the following statements at the top of the file:

```
using System.IO;
```

4. Add the following method to the **Program** class:

```
private static async void SendToBlobAsync()
{
    string fileName = "image.jpg";
    Console.WriteLine("Uploading file: {0}", fileName);
    var watch = System.Diagnostics.Stopwatch.StartNew();

    using (var sourceData = new FileStream(@"image.jpg", FileMode.Open))
    {
        await deviceClient.UploadToBlobAsync(fileName, sourceData);
    }

    watch.Stop();
    Console.WriteLine("Time to upload file: {0}ms\n", watch.ElapsedMilliseconds);
}
```

The `UploadToBlobAsync` method takes in the file name and stream source of the file to be uploaded and handles the upload to storage. The console app displays the time it takes to upload the file.

5. Add the following method in the **Main** method, right before the `Console.ReadLine()` line:

```
SendToBlobAsync();
```

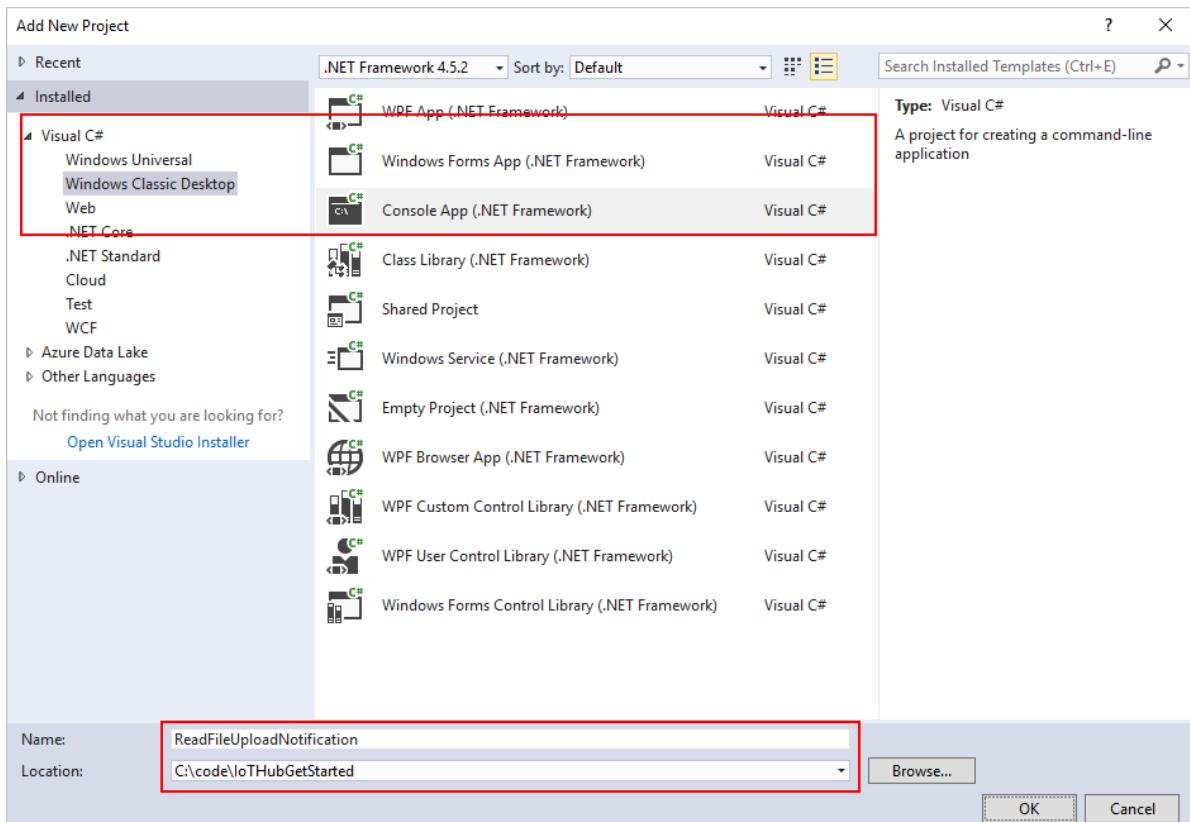
NOTE

For simplicity's sake, this tutorial does not implement any retry policy. In production code, you should implement retry policies (such as exponential backoff), as suggested in the article, [Transient Fault Handling](#).

Receive a file upload notification

In this section, you write a .NET console app that receives file upload notification messages from IoT Hub.

1. In the current Visual Studio solution, create a Visual C# Windows project by using the **Console Application** project template. Name the project **ReadFileUploadNotification**.



2. In Solution Explorer, right-click the **ReadFileUploadNotification** project, and then click **Manage NuGet Packages....**
3. In the **NuGet Package Manager** window, search for **Microsoft.Azure.Devices**, click **Install**, and accept the terms of use.

This action downloads, installs, and adds a reference to the [Azure IoT service SDK NuGet package](#) in the **ReadFileUploadNotification** project.

4. In the **Program.cs** file, add the following statements at the top of the file:

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

5. Add the following fields to the **Program** class. Substitute the placeholder value with the IoT hub connection string from [Send telemetry from a device to an IoT hub](#):

```
static ServiceClient serviceClient;  
static string connectionString = "{iot hub connection string}";
```

6. Add the following method to the **Program** class:

```

private async static void ReceiveFileUploadNotificationAsync()
{
    var notificationReceiver = serviceClient.GetFileNotificationReceiver();

    Console.WriteLine("\nReceiving file upload notification from service");
    while (true)
    {
        var fileUploadNotification = await notificationReceiver.ReceiveAsync();
        if (fileUploadNotification == null) continue;

        Console.ForegroundColor = ConsoleColor.Yellow;
        Console.WriteLine("Received file upload notification: {0}",
            string.Join(", ", fileUploadNotification.BlobName));
        Console.ResetColor();

        await notificationReceiver.CompleteAsync(fileUploadNotification);
    }
}

```

Note this receive pattern is the same one used to receive cloud-to-device messages from the device app.

- Finally, add the following lines to the **Main** method:

```

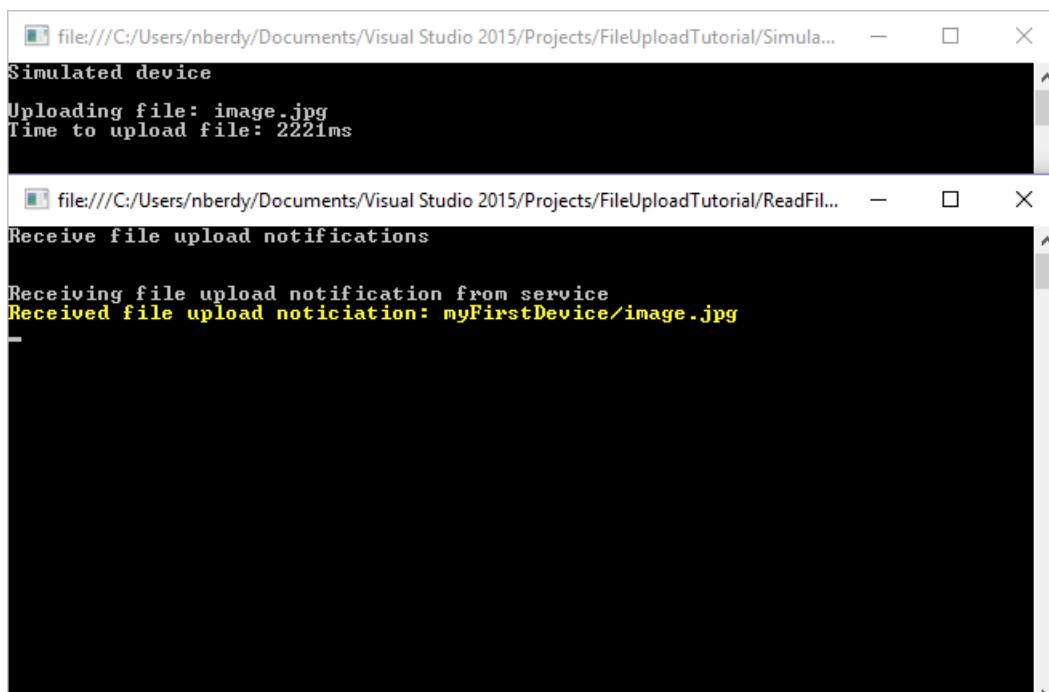
Console.WriteLine("Receive file upload notifications\n");
serviceClient = ServiceClient.CreateFromConnectionString(connectionString);
ReceiveFileUploadNotificationAsync();
Console.WriteLine("Press Enter to exit\n");
Console.ReadLine();

```

Run the applications

Now you are ready to run the applications.

- In Visual Studio, right-click your solution, and select **Set StartUp projects**. Select **Multiple startup projects**, then select the **Start** action for **ReadFileUploadNotification** and **SimulatedDevice**.
- Press **F5**. Both applications should start. You should see the upload completed in one console app and the upload notification message received by the other console app. You can use the [Azure portal](#) or Visual Studio Server Explorer to check for the presence of the uploaded file in your Azure Storage account.



Next steps

In this tutorial, you learned how to use the file upload capabilities of IoT Hub to simplify file uploads from devices. You can continue to explore IoT hub features and scenarios with the following articles:

- [Create an IoT hub programmatically](#)
- [Introduction to C SDK](#)
- [Azure IoT SDKs](#)

To further explore the capabilities of IoT Hub, see:

- [Deploying AI to edge devices with Azure IoT Edge](#)

Upload files from your device to the cloud with IoT Hub

3/6/2019 • 6 minutes to read

This tutorial builds on the code in the [Send Cloud-to-Device messages with IoT Hub](#) tutorial to show you how to use the [file upload capabilities of IoT Hub](#) to upload a file to [Azure blob storage](#). The tutorial shows you how to:

- Securely provide a device with an Azure blob URI for uploading a file.
- Use the IoT Hub file upload notifications to trigger processing the file in your app back end.

The [Send telemetry to IoT Hub \(Java\)](#) and [Send Cloud-to-Device messages with IoT Hub \(Java\)](#) tutorials show the basic device-to-cloud and cloud-to-device messaging functionality of IoT Hub. The [Configure message routing with IoT Hub](#) tutorial describes a way to reliably store device-to-cloud messages in Azure blob storage. However, in some scenarios you cannot easily map the data your devices send into the relatively small device-to-cloud messages that IoT Hub accepts. For example:

- Large files that contain images
- Videos
- Vibration data sampled at high frequency
- Some form of preprocessed data.

These files are typically batch processed in the cloud using tools such as [Azure Data Factory](#) or the [Hadoop](#) stack. When you need to upload files from a device, you can still use the security and reliability of IoT Hub.

At the end of this tutorial you run two Java console apps:

- **simulated-device**, a modified version of the app created in the [Send Cloud-to-Device messages with IoT Hub] tutorial. This app uploads a file to storage using a SAS URI provided by your IoT hub.
- **read-file-upload-notification**, which receives file upload notifications from your IoT hub.

NOTE

IoT Hub supports many device platforms and languages (including C, .NET, and Javascript) through Azure IoT device SDKs. Refer to the [Azure IoT Developer Center](#) for step-by-step instructions on how to connect your device to Azure IoT Hub.

To complete this tutorial, you need the following:

- The latest [Java SE Development Kit 8](#)
- [Maven 3](#)
- An active Azure account. (If you don't have an account, you can create a [free account](#) in just a couple of minutes.)

Associate an Azure Storage account to IoT Hub

Because the simulated device app uploads a file to a blob, you must have an [Azure Storage](#) account associated with your IoT hub. When you associate an Azure Storage account with an IoT hub, the IoT hub generates a SAS URI. A device can use this SAS URI to securely upload a file to a blob container. The IoT Hub service and the device SDKs coordinate the process that generates the SAS URI and makes it available to a device to use to

upload a file.

Follow the instructions in [Configure file uploads using the Azure portal](#). Make sure that a blob container is associated with your IoT hub and that file notifications are enabled.

The screenshot shows the 'File upload' settings page for an IoT hub named 'iothubd'. On the left, there's a sidebar with links like Overview, Activity log, Access control (IAM), and various settings sections. The 'MESSAGING' section has 'File upload' selected. The main area shows a 'Storage container' set to 'iotdblob'. A red box highlights the 'Receive notifications for uploaded files' section, which has a switch set to 'On'. Below it are sliders for 'SAS TTL' (set to 1 hr), 'Default TTL' (set to 1 hr), and 'Maximum delivery count' (set to 10).

Upload a file from a device app

In this section, you modify the device app you created in [Send Cloud-to-Device messages with IoT Hub](#) to upload a file to IoT hub.

1. Copy an image file to the `simulated-device` folder and rename it `myimage.png`.
2. Using a text editor, open the `simulated-device\src\main\java\com\mycompany\app\App.java` file.
3. Add the variable declaration to the **App** class:

```
private static String fileName = "myimage.png";
```

4. To process file upload status callback messages, add the following nested class to the **App** class:

```
// Define a callback method to print status codes from IoT Hub.
protected static class FileUploadStatusCallBack implements IoTHubEventCallback {
    public void execute(IotHubStatusCode status, Object context) {
        System.out.println("IoT Hub responded to file upload for " + fileName
            + " operation with status " + status.name());
    }
}
```

5. To upload images to IoT Hub, add the following method to the **App** class to upload images to IoT Hub:

```

// Use IoT Hub to upload a file asynchronously to Azure blob storage.
private static void uploadFile(String fullFileName) throws FileNotFoundException, IOException
{
    File file = new File(fullFileName);
    InputStream inputStream = new FileInputStream(file);
    long streamLength = file.length();

    client.uploadToBlobAsync(fileName, inputStream, streamLength, new FileUploadStatusCallBack(), null);
}

```

6. Modify the **main** method to call the **uploadFile** method as shown in the following snippet:

```

client.open();

try
{
    // Get the filename and start the upload.
    String fullFileName = System.getProperty("user.dir") + File.separator + fileName;
    uploadFile(fullFileName);
    System.out.println("File upload started with success");
}
catch (Exception e)
{
    System.out.println("Exception uploading file: " + e.getCause() + " \nERROR: " + e.getMessage());
}

MessageSender sender = new MessageSender();

```

7. Use the following command to build the **simulated-device** app and check for errors:

```
mvn clean package -DskipTests
```

Receive a file upload notification

In this section, you create a Java console app that receives file upload notification messages from IoT Hub.

You need the **iothubowner** connection string for your IoT Hub to complete this section. You can find the connection string in the [Azure portal](#) on the **Shared access policy** blade.

1. Create a Maven project called **read-file-upload-notification** using the following command at your command prompt. Note this command is a single, long command:

```
mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=read-file-upload-notification -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

2. At your command prompt, navigate to the new `read-file-upload-notification` folder.
3. Using a text editor, open the `pom.xml` file in the `read-file-upload-notification` folder and add the following dependency to the **dependencies** node. Adding the dependency enables you to use the **iothub-java-service-client** package in your application to communicate with your IoT hub service:

```

<dependency>
    <groupId>com.microsoft.azure.sdk.iot</groupId>
    <artifactId>iot-service-client</artifactId>
    <version>1.7.23</version>
</dependency>

```

NOTE

You can check for the latest version of **iot-service-client** using [Maven search](#).

4. Save and close the `pom.xml` file.
5. Using a text editor, open the `read-file-upload-notification\src\main\java\com\mycompany\app\App.java` file.
6. Add the following **import** statements to the file:

```
import com.microsoft.azure.sdk.iot.service.*;
import java.io.IOException;
import java.net.URISyntaxException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
```

7. Add the following class-level variables to the **App** class:

```
private static final String connectionString = "{Your IoT Hub connection string}";
private static final IoTHubServiceClientProtocol protocol = IoTHubServiceClientProtocol.AMQPS;
private static FileUploadNotificationReceiver fileUploadNotificationReceiver = null;
```

8. To print information about the file upload to the console, add the following nested class to the **App** class:

```
// Create a thread to receive file upload notifications.
private static class ShowFileUploadNotifications implements Runnable {
    public void run() {
        try {
            while (true) {
                System.out.println("Receive file upload notifications...");
                FileUploadNotification fileUploadNotification = fileUploadNotificationReceiver.receive();
                if (fileUploadNotification != null) {
                    System.out.println("File Upload notification received");
                    System.out.println("Device Id : " + fileUploadNotification.getDeviceId());
                    System.out.println("Blob Uri: " + fileUploadNotification.getBlobUri());
                    System.out.println("Blob Name: " + fileUploadNotification.getBlobName());
                    System.out.println("Last Updated : " + fileUploadNotification.getLastUpdatedTimeDate());
                    System.out.println("Blob Size (Bytes): " + fileUploadNotification.getBlobSizeInBytes());
                    System.out.println("Enqueued Time: " + fileUploadNotification.getEnqueuedTimeUtcDate());
                }
            }
        } catch (Exception ex) {
            System.out.println("Exception reading reported properties: " + ex.getMessage());
        }
    }
}
```

9. To start the thread that listens for file upload notifications, add the following code to the **main** method:

```

public static void main(String[] args) throws IOException, URISyntaxException, Exception {
    ServiceClient serviceClient = ServiceClient.createFromConnectionString(connectionString, protocol);

    if (serviceClient != null) {
        serviceClient.open();

        // Get a file upload notification receiver from the ServiceClient.
        fileUploadNotificationReceiver = serviceClient.getFileUploadNotificationReceiver();
        fileUploadNotificationReceiver.open();

        // Start the thread to receive file upload notifications.
        ShowFileUploadNotifications showFileUploadNotifications = new ShowFileUploadNotifications();
        ExecutorService executor = Executors.newFixedThreadPool(1);
        executor.execute(showFileUploadNotifications);

        System.out.println("Press ENTER to exit.");
        System.in.read();
        executor.shutdownNow();
        System.out.println("Shutting down sample...");
        fileUploadNotificationReceiver.close();
        serviceClient.close();
    }
}

```

10. Save and close the `read-file-upload-notification\src\main\java\com\mycompany\app\App.java` file.

11. Use the following command to build the **read-file-upload-notification** app and check for errors:

```
mvn clean package -DskipTests
```

Run the applications

Now you are ready to run the applications.

At a command prompt in the `read-file-upload-notification` folder, run the following command:

```
mvn exec:java -Dexec.mainClass="com.mycompany.app.App"
```

At a command prompt in the `simulated-device` folder, run the following command:

```
mvn exec:java -Dexec.mainClass="com.mycompany.app.App"
```

The following screenshot shows the output from the **simulated-device** app:

```
simulated-device
c:\repos\samples\JavaIoTFileupload>run-simulated-device.bat
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building simulated-device 1.0-SNAPSHOT
[INFO] -----
[INFO] --- exec-maven-plugin:1.4.0:java (default-cli) @ simulated-device ---
log4j:WARN No appenders could be found for logger (com.microsoft.azure.sdk.iot.device.IotHubConnectionString).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
CWD: c:\repos\samples\JavaIoTFileupload\simulated-device
File upload started with success
Waiting for file upload callback with the status...
Press ENTER to exit.
Sending: {"deviceId": "myFirstDeviceAgain", "windspeed": 10.5406552256608}
IoT Hub responded to message with status: OK_EMPTY
IoT Hub responded to file upload for myimage.png operation with status OK_EMPTY
Sending: {"deviceId": "myFirstDeviceAgain", "windspeed": 9.37813739407624}
IoT Hub responded to message with status: OK_EMPTY
```

The following screenshot shows the output from the **read-file-upload-notification** app:

```
read-file-upload-notification
c:\repos\samples\JavaIoTFileupload>run-read-file-upload-notification.bat
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building read-file-upload-notification 1.0-SNAPSHOT
[INFO] -----
[INFO] --- exec-maven-plugin:1.4.0:java (default-cli) @ read-file-upload-notification ---
Press ENTER to exit.
Recieve file upload notifications...
File Upload notification received
Device Id : myFirstDeviceAgain
Blob Uri: https://myiothubfileupload.blob.core.windows.net/uploadcontainer/myFirstDeviceAgain/myimage.png
Blob Name: myFirstDeviceAgain/myimage.png
Last Updated : Tue Jun 27 15:22:38 BST 2017
Blob Size (Bytes): 1493
Enqueued Time: Tue Jun 27 17:17:20 BST 2017
Recieve file upload notifications...
```

You can use the portal to view the uploaded file in the storage container you configured:

The screenshot shows the Azure Storage Blob service interface. On the left, under 'Container' 'myiothubfileupload', there's a list of containers: 'another', 'fileupload', 'qtcontainer', and 'uploadcontainer'. The 'uploadcontainer' row is highlighted with a blue background. On the right, under 'Folder' 'myFirstDeviceAgain', it shows a single file named 'myimage.png'. The file details are listed as follows:

NAME	MODIFIED
[-]	27/06/2017 03:22
myimage.png	27/06/2017 03:22

Next steps

In this tutorial, you learned how to use the file upload capabilities of IoT Hub to simplify file uploads from devices. You can continue to explore IoT hub features and scenarios with the following articles:

- [Create an IoT hub programmatically](#)
- [Introduction to C SDK](#)
- [Azure IoT SDKs](#)

To further explore the capabilities of IoT Hub, see:

- [Simulating a device with IoT Edge](#)

Upload files from your device to the cloud with IoT Hub

3/6/2019 • 5 minutes to read

This tutorial builds on the code in the [Send Cloud-to-Device messages with IoT Hub](#) tutorial to show you how to use the [file upload capabilities of IoT Hub](#) to upload a file to [Azure blob storage](#). The tutorial shows you how to:

- Securely provide a device with an Azure blob URI for uploading a file.
- Use the IoT Hub file upload notifications to trigger processing the file in your app back end.

The [Get started with IoT Hub](#) tutorial demonstrates the basic device-to-cloud messaging functionality of IoT Hub. However, in some scenarios you cannot easily map the data your devices send into the relatively small device-to-cloud messages that IoT Hub accepts. For example:

- Large files that contain images
- Videos
- Vibration data sampled at high frequency
- Some form of preprocessed data.

These files are typically batch processed in the cloud using tools such as [Azure Data Factory](#) or the [Hadoop](#) stack. When you need to upload files from a device, you can still use the security and reliability of IoT Hub.

At the end of this tutorial you run two Node.js console apps:

- **SimulatedDevice.js**, which uploads a file to storage using a SAS URI provided by your IoT hub.
- **ReadFileUploadNotification.js**, which receives file upload notifications from your IoT hub.

NOTE

IoT Hub supports many device platforms and languages (including C, .NET, Javascript, Python, and Java) through Azure IoT device SDKs. Refer to the [Azure IoT Developer Center](#) for step-by-step instructions on how to connect your device to Azure IoT Hub.

To complete this tutorial, you need the following:

- Node.js version 4.0.x or later.
- An active Azure account. (If you don't have an account, you can create a [free account](#) in just a couple of minutes.)

Associate an Azure Storage account to IoT Hub

Because the simulated device app uploads a file to a blob, you must have an [Azure Storage](#) account associated with your IoT hub. When you associate an Azure Storage account with an IoT hub, the IoT hub generates a SAS URI. A device can use this SAS URI to securely upload a file to a blob container. The IoT Hub service and the device SDKs coordinate the process that generates the SAS URI and makes it available to a device to use to upload a file.

Follow the instructions in [Configure file uploads using the Azure portal](#). Make sure that a blob container is associated with your IoT hub and that file notifications are enabled.

Upload a file from a device app

In this section, you create the device app to upload a file to IoT hub.

1. Create an empty folder called `simulateddevice`. In the `simulateddevice` folder, create a `package.json` file using the following command at your command prompt. Accept all the defaults:

```
npm init
```

2. At your command prompt in the `simulateddevice` folder, run the following command to install the **azure-iot-device** Device SDK package and **azure-iot-device-mqtt** package:

```
npm install azure-iot-device azure-iot-device-mqtt --save
```

3. Using a text editor, create a **SimulatedDevice.js** file in the `simulateddevice` folder.

4. Add the following `require` statements at the start of the **SimulatedDevice.js** file:

```
'use strict';

var fs = require('fs');
var mqtt = require('azure-iot-device-mqtt').Mqtt;
var clientFromConnectionString = require('azure-iot-device-mqtt').clientFromConnectionString;
```

5. Add a `deviceconnectionstring` variable and use it to create a **Client** instance. Replace `{deviceconnectionstring}` with the name of the device you created in the *Create an IoT Hub* section:

```
var connectionString = '{deviceconnectionstring}';  
var filename = 'myimage.png';
```

NOTE

For the sake of simplicity the connection string is included in the code: this is not a recommended practice and depending on your use-case and architecture you may want to consider more secure ways of storing this secret.

6. Add the following code to connect the client:

```
var client = clientFromConnectionString(connectionString);  
console.log('Client connected');
```

7. Create a callback and use the **uploadToBlob** function to upload the file.

```
fs.stat(filename, function (err, stats) {  
    const rr = fs.createReadStream(filename);  
  
    client.uploadToBlob(filename, rr, stats.size, function (err) {  
        if (err) {  
            console.error('Error uploading file: ' + err.toString());  
        } else {  
            console.log('File uploaded');  
        }  
    });  
});
```

8. Save and close the **SimulatedDevice.js** file.

9. Copy an image file to the `simulateddevice` folder and rename it `myimage.png`.

Receive a file upload notification

In this section, you create a Node.js console app that receives file upload notification messages from IoT Hub.

You can use the **iothubowner** connection string from your IoT Hub to complete this section. You will find the connection string in the [Azure portal](#) on the **Shared access policy** blade.

1. Create an empty folder called `fileuploadnotification`. In the `fileuploadnotification` folder, create a `package.json` file using the following command at your command prompt. Accept all the defaults:

```
npm init
```

2. At your command prompt in the `fileuploadnotification` folder, run the following command to install the **azure-iothub** SDK package:

```
npm install azure-iothub --save
```

3. Using a text editor, create a **FileUploadNotification.js** file in the `fileuploadnotification` folder.

4. Add the following `require` statements at the start of the **FileUploadNotification.js** file:

```
'use strict';

var Client = require('azure-iothub').Client;
```

5. Add a `iothubconnectionstring` variable and use it to create a **Client** instance. Replace `{iothubconnectionstring}` with the connection string to the IoT hub you created in the *Create an IoT Hub* section:

```
var connectionString = '{iothubconnectionstring}';
```

NOTE

For the sake of simplicity the connection string is included in the code: this is not a recommended practice and depending on your use-case and architecture you may want to consider more secure ways of storing this secret.

6. Add the following code to connect the client:

```
var serviceClient = Client.fromConnectionString(connectionString);
```

7. Open the client and use the **getFileNotificationReceiver** function to receive status updates.

```
serviceClient.open(function (err) {
  if (err) {
    console.error('Could not connect: ' + err.message);
  } else {
    console.log('Service client connected');
    serviceClient.getFileNotificationReceiver(function receiveFileUploadNotification(err, receiver){
      if (err) {
        console.error('error getting the file notification receiver: ' + err.toString());
      } else {
        receiver.on('message', function (msg) {
          console.log('File upload from device:')
          console.log(msg.getData().toString('utf-8'));
        });
      }
    });
  }
});
```

8. Save and close the **FileUploadNotification.js** file.

Run the applications

Now you are ready to run the applications.

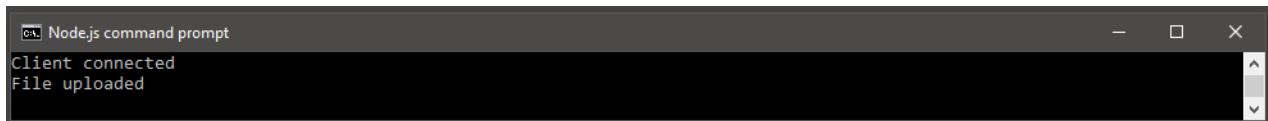
At a command prompt in the `fileuploadnotification` folder, run the following command:

```
node FileUploadNotification.js
```

At a command prompt in the `simulateddevice` folder, run the following command:

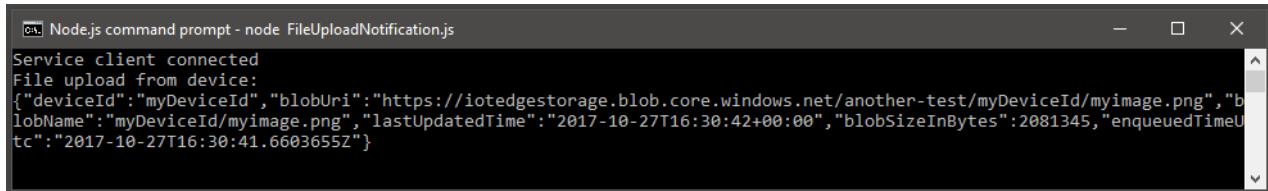
```
node SimulatedDevice.js
```

The following screenshot shows the output from the **SimulatedDevice** app:



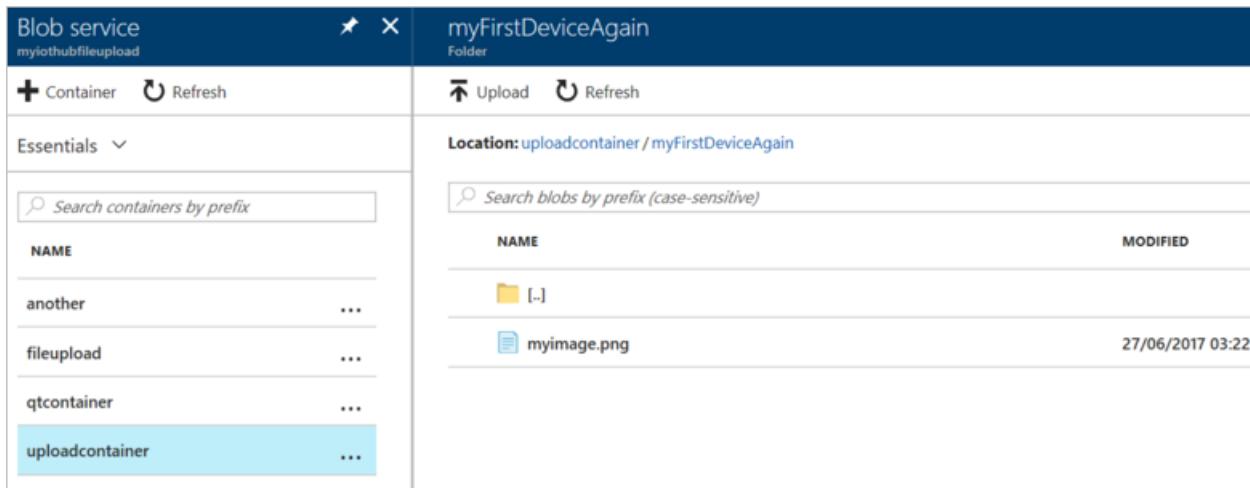
```
Node.js command prompt
Client connected
File uploaded
```

The following screenshot shows the output from the **FileUploadNotification** app:



```
Node.js command prompt - node FileUploadNotification.js
Service client connected
File upload from device:
{"deviceId": "myDeviceId", "blobUri": "https://iotedgestorage.blob.core.windows.net/another-test/myDeviceId/myimage.png", "blobName": "myDeviceId/myimage.png", "lastUpdatedTime": "2017-10-27T16:30:42+00:00", "blobSizeInBytes": 2081345, "enqueuedTimeUtc": "2017-10-27T16:30:41.6603655Z"}
```

You can use the portal to view the uploaded file in the storage container you configured:



The screenshot shows the Azure Storage Blob service interface. On the left, a list of containers is shown, with 'uploadcontainer' selected. On the right, a detailed view of the 'myFirstDeviceAgain' folder within 'uploadcontainer' is displayed, showing a single file named 'myimage.png'.

NAME	MODIFIED
[..]	
myimage.png	27/06/2017 03:22:

Next steps

In this tutorial, you learned how to use the file upload capabilities of IoT Hub to simplify file uploads from devices. You can continue to explore IoT hub features and scenarios with the following articles:

- [Create an IoT hub programmatically](#)
- [Introduction to C SDK](#)
- [Azure IoT SDKs](#)

Upload files from your device to the cloud with IoT Hub

3/6/2019 • 4 minutes to read

This article shows how to use the [file upload capabilities of IoT Hub](#) to upload a file to [Azure blob storage](#). The tutorial shows you how to:

- Securely provide a storage container for uploading a file.
- Use the Python client to upload a file through your IoT hub.

The [Send telemetry to IoT Hub](#) quickstart demonstrates the basic device-to-cloud messaging functionality of IoT Hub. However, in some scenarios you cannot easily map the data your devices send into the relatively small device-to-cloud messages that IoT Hub accepts. When you need to upload files from a device, you can still use the security and reliability of IoT Hub.

NOTE

IoT Hub Python SDK currently only supports uploading character-based files such as **.txt** files.

At the end of this tutorial you run the Python console app:

- **FileUpload.py**, which uploads a file to storage using the Python Device SDK.

NOTE

IoT Hub supports many device platforms and languages (including C, .NET, Javascript, Python, and Java) through Azure IoT device SDKs. Refer to the [Azure IoT Developer Center](#) for step-by-step instructions on how to connect your device to Azure IoT Hub.

To complete this tutorial, you need the following:

- [Python 2.x or 3.x](#). Make sure to use the 32-bit or 64-bit installation as required by your setup. When prompted during the installation, make sure to add Python to your platform-specific environment variable. If you are using Python 2.x, you may need to [install or upgrade pip, the Python package management system](#).
- If you are using Windows OS, then [Visual C++ redistributable package](#) to allow the use of native DLLs from Python.
- An active Azure account. If you don't have an account, you can create a [free account](#) in just a couple of minutes.
- An IoT hub in your Azure account, with a device identity for testing the file upload feature.

Associate an Azure Storage account to IoT Hub

Because the simulated device app uploads a file to a blob, you must have an [Azure Storage](#) account associated with your IoT hub. When you associate an Azure Storage account with an IoT hub, the IoT hub generates a SAS URI. A device can use this SAS URI to securely upload a file to a blob container. The IoT Hub service and the device SDKs coordinate the process that generates the SAS URI and makes it available to a device to use to upload a file.

Follow the instructions in [Configure file uploads using the Azure portal](#). Make sure that a blob container is associated with your IoT hub and that file notifications are enabled.

The screenshot shows the Azure IoT Hub - File upload settings page. On the left, there's a sidebar with various navigation options. The 'File upload' option is highlighted with a blue box. The main area is titled 'File notification settings' and contains three sliders: 'SAS TTL' (set to 1 hr), 'Default TTL' (set to 1 hr), and 'Maximum delivery count' (set to 10). A red box highlights the 'Storage container' section, which shows 'iotdblob' and a toggle switch for 'Receive notifications for uploaded files' which is set to 'On'.

Upload a file from a device app

In this section, you create the device app to upload a file to IoT hub.

1. At your command prompt, run the following command to install the **azure-iothub-device-client** package:

```
pip install azure-iothub-device-client
```

2. Using a text editor, create a test file that you will upload to blob storage.

NOTE

IoT Hub Python SDK currently only supports uploading character-based files such as **.txt** files.

3. Using a text editor, create a **FileUpload.py** file in your working folder.

4. Add the following `import` statements and variables at the start of the **FileUpload.py** file.

```

import time
import sys
import iothub_client
import os
from iothub_client import IoTHubClient, IoTHubClientError, IoTHubTransportProvider, IoTHubClientResult,
IoTHubError

CONNECTION_STRING = "[Device Connection String]"
PROTOCOL = IoTHubTransportProvider.HTTP

PATHTOFILE = "[Full path to file]"
FILENAME = "[File name for storage]"

```

5. In your file, replace `[Device Connection String]` with the connection string of your IoT hub device. Replace `[Full path to file]` with the path to the test file that you created, or any file on your device that you want to upload. Replace `[File name for storage]` with the name that you want to give to your file after it's uploaded to blob storage.

6. Create a callback for the **upload_blob** function:

```

def blob_upload_conf_callback(result, user_context):
    if str(result) == 'OK':
        print ( "...file uploaded successfully." )
    else:
        print ( "...file upload callback returned: " + str(result) )

```

7. Add the following code to connect the client and upload the file. Also include the `main` routine:

```

def iothub_file_upload_sample_run():
    try:
        print ( "IoT Hub file upload sample, press Ctrl-C to exit" )

        client = IoTHubClient(CONNECTION_STRING, PROTOCOL)

        f = open(PATHTOFILE, "r")
        content = f.read()

        client.upload_blob_async(FILENAME, content, len(content), blob_upload_conf_callback, 0)

        print ( "" )
        print ( "File upload initiated..." )

        while True:
            time.sleep(30)

    except IoTHubError as iothub_error:
        print ( "Unexpected error %s from IoTHub" % iothub_error )
        return
    except KeyboardInterrupt:
        print ( "IoTHubClient sample stopped" )
    except:
        print ( "generic error" )

if __name__ == '__main__':
    print ( "Simulating a file upload using the Azure IoT Hub Device SDK for Python" )
    print ( "    Protocol %s" % PROTOCOL )
    print ( "    Connection string=%s" % CONNECTION_STRING )

    iothub_file_upload_sample_run()

```

8. Save and close the **UploadFile.py** file.

Run the application

Now you are ready to run the application.

1. At a command prompt in your working folder, run the following command:

```
python FileUpload.py
```

2. The following screenshot shows the output from the **FileUpload** app:

```
Command Prompt
Simulating a file upload using the Azure IoT Hub Device SDK for Python
Protocol HTTP
Connection string=HostName=device-page.azure-devices.net;DeviceId=testDevice;SharedAccessKey=
IoT Hub file upload sample, press Ctrl-C to exit
File upload initiated...
...file uploaded successfully.
IoTHubClient sample stopped
```

3. You can use the portal to view the uploaded file in the storage container you configured:

The screenshot shows the Microsoft Azure portal interface for managing a blob service. On the left, there's a sidebar with various icons. The main area shows a 'Blob service' blade for a device named 'testDevice'. Under 'Container', there are two entries: 'device-storage' and 'iot-test'. The 'iot-test' container is currently selected. Inside 'iot-test', there are two blobs: a folder named '[-]' and a file named 'sample.txt'. The 'sample.txt' file is highlighted with a red box. The table below lists the blobs with columns for NAME, MODIFIED, BLOB TYPE, SIZE, and LEASE STATE.

NAME	MODIFIED	BLOB TYPE	SIZE	LEASE STATE
[-]	-	-	-	...
subdir	-	-	-	...
sample.txt	2/2/2018, 1:49:40 PM	Block blob	26 B	Available

Next steps

In this tutorial, you learned how to use the file upload capabilities of IoT Hub to simplify file uploads from devices. You can continue to explore IoT hub features and scenarios with the following articles:

- [Create an IoT hub programmatically](#)
- [Introduction to C SDK](#)
- [Azure IoT SDKs](#)

Get started with device twins (Node)

3/6/2019 • 10 minutes to read

Device twins are JSON documents that store device state information (metadata, configurations, and conditions). IoT Hub persists a device twin for each device that connects to it.

NOTE

The features described in this article are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

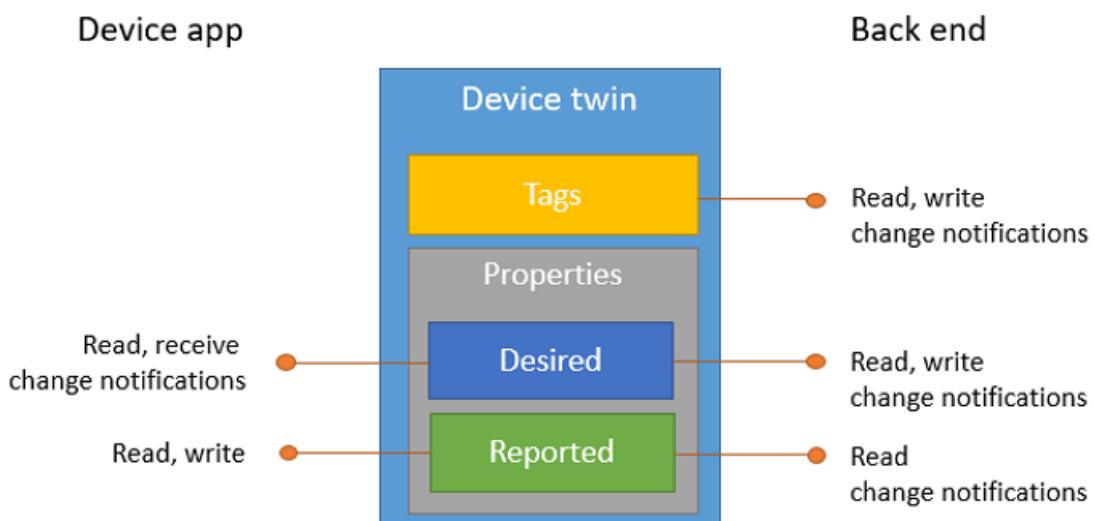
Use device twins to:

- Store device metadata from your solution back end.
- Report current state information such as available capabilities and conditions (for example, the connectivity method used) from your device app.
- Synchronize the state of long-running workflows (such as firmware and configuration updates) between a device app and a back-end app.
- Query your device metadata, configuration, or state.

Device twins are designed for synchronization and for querying device configurations and conditions. More information on when to use device twins can be found in [Understand device twins](#).

Device twins are stored in an IoT hub and contain:

- *tags*, device metadata accessible only by the solution back end;
- *desired properties*, JSON objects modifiable by the solution back end and observable by the device app; and
- *reported properties*, JSON objects modifiable by the device app and readable by the solution back end. Tags and properties cannot contain arrays, but objects can be nested.



Additionally, the solution back end can query device twins based on all the above data. Refer to [Understand device twins](#) for more information about device twins, and to the [IoT Hub query language](#) reference for querying.

This tutorial shows you how to:

- Create a back-end app that adds *tags* to a device twin, and a simulated device app that reports its connectivity channel as a *reported property* on the device twin.
- Query devices from your back-end app using filters on the tags and properties previously created.

At the end of this tutorial, you will have two Node.js console apps:

- **AddTagsAndQuery.js**, a Node.js back-end app, which adds tags and queries device twins.
- **TwinSimulatedDevice.js**, a Node.js app, which simulates a device that connects to your IoT hub with the device identity created earlier, and reports its connectivity condition.

NOTE

The article [Azure IoT SDKs](#) provides information about the Azure IoT SDKs that you can use to build both device and back-end apps.

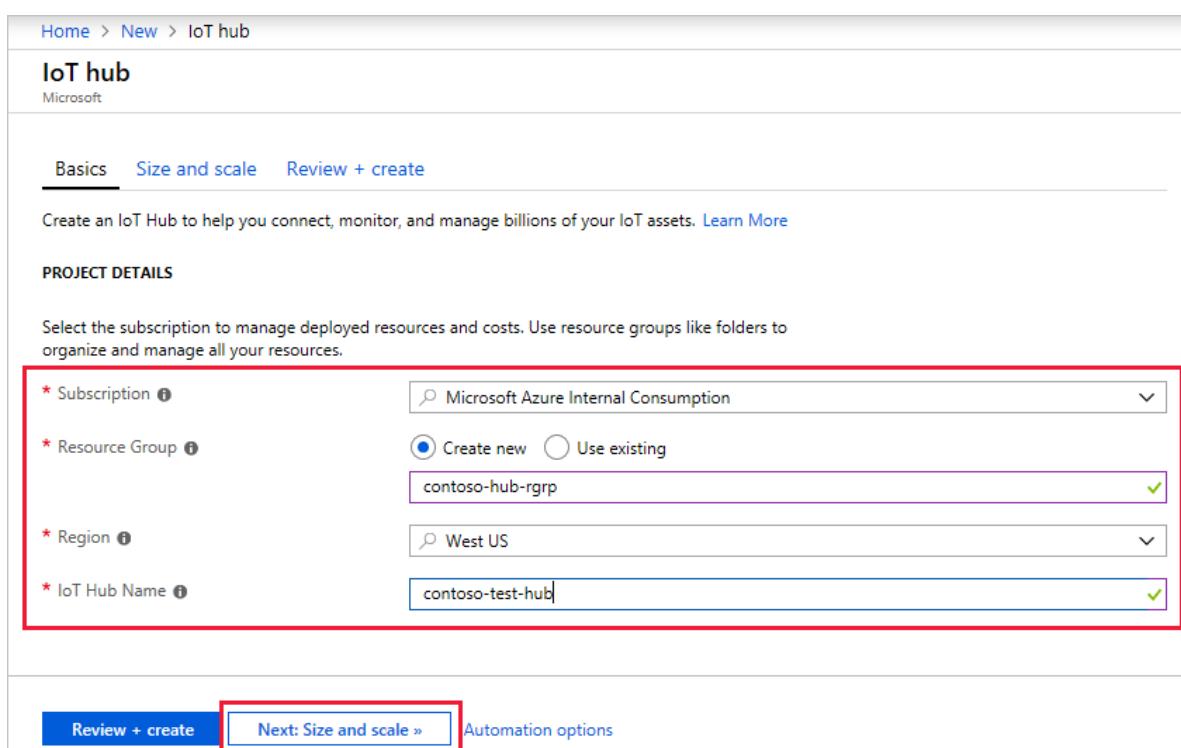
To complete this tutorial you need the following:

- Node.js version 4.0.x or later.
- An active Azure account. (If you don't have an account, you can create a [free account](#) in just a couple of minutes.)

Create an IoT hub

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose +**Create a resource**, then choose **Internet of Things**.
3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.



Fill in the fields.

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

The screenshot shows the 'Size and scale' configuration page for an IoT hub. At the top, there are tabs for 'Basics', 'Size and scale' (which is selected), and 'Review + create'. A note below the tabs states: 'Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)'.

SCALE TIER AND UNITS

* Pricing and scale tier: S1: Standard tier. A link to 'Learn how to choose the right IoT Hub tier for your solution' is provided.

Number of S1 IoT Hub units: A slider set to 1. A note below it says: 'This determines your IoT Hub scale capability and can be changed as your need increases.'

Enabled Features:

- Device-to-cloud-messages: Enabled
- Message routing: Enabled
- Cloud-to-device commands: Enabled
- IoT Edge: Enabled
- Device management: Enabled

Advanced Settings:

Device-to-cloud partitions: A slider set to 4.

At the bottom, there are buttons for 'Review + create', '< Previous: Basics', and 'Automation options'.

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free tier is intended for testing and

evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: The number of messages allowed per unit per day depends on your hub's pricing tier. For example, if you want the IoT hub to support ingress of 700,000 messages, you choose two S1 tier units.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

Advanced / Device-to-cloud partitions: This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most IoT hubs only need four partitions.

5. Click **Review + create** to review your choices. You see something similar to this screen.

The screenshot shows the 'Review + create' step of creating an IoT hub in the Azure portal. The 'Review + create' button is highlighted with a red box. At the bottom left, the 'Create' button is also highlighted with a red box. The page displays basic information like Subscription, Resource Group, Region, and IoT Hub Name, as well as size and scale details like Pricing tier, Number of units, Messages per day, and Cost per month.

6. Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Retrieve connection string for IoT hub

After your hub has been created, retrieve the connection string for the hub. This is used to connect devices and applications to your hub.

1. Click on your hub to see the IoT Hub pane with Settings, and so on. Click **Shared access policies**.
2. In **Shared access policies**, select the **iothubowner** policy.
3. Under **Shared access keys**, copy the **Connection string -- primary key** to be used later.

The screenshot shows the 'Shared access policies' section of the Azure IoT Hub 'ContosoHub' settings. A new policy named 'iothubowner' is being created. The policy includes permissions for Registry read, Registry write, Service connect, and Device connect. Under 'Shared access keys', the primary key connection string is highlighted with a red box: 'HostName=ContosoHub.azure-devices.net;SharedAccessKey...'. The secondary key connection string is also listed below it.

For more information, see [Access control](#) in the "IoT Hub developer guide."

Create a device identity

In this section, you use the Azure CLI to create a device identity for this tutorial. The Azure CLI is preinstalled in the [Azure Cloud Shell](#), or you can [install it locally](#). Device IDs are case sensitive.

1. Run the following command in the command-line environment where you are using the Azure CLI to install the IoT extension:

```
az extension add --name azure-cli-iot-ext
```

2. If you are running the Azure CLI locally, use the following command to sign in to your Azure account (if you are using the Cloud Shell, you are signed in automatically and you don't need to run this command):

```
az login
```

3. Finally, create a new device identity called `myDeviceId` and retrieve the device connection string with these commands:

```
az iot hub device-identity create --device-id myDeviceId --hub-name {Your IoT Hub name}  
az iot hub device-identity show-connection-string --device-id myDeviceId --hub-name {Your IoT Hub name} -o table
```

IMPORTANT

The device ID may be visible in the logs collected for customer support and troubleshooting, so make sure to avoid any sensitive information while naming it.

Make a note of the device connection string from the result. This device connection string is used by the device app to connect to your IoT Hub as a device.

Create the service app

In this section, you create a Node.js console app that adds location metadata to the device twin associated with **myDeviceId**. It then queries the device twins stored in the IoT hub selecting the devices located in the US, and then the ones that are reporting a cellular connection.

1. Create a new empty folder called **addtagsandqueryapp**. In the **addtagsandqueryapp** folder, create a new package.json file using the following command at your command prompt. Accept all the defaults:

```
npm init
```

2. At your command prompt in the **addtagsandqueryapp** folder, run the following command to install the **azure-iothub** package:

```
npm install azure-iothub --save
```

3. Using a text editor, create a new **AddTagsAndQuery.js** file in the **addtagsandqueryapp** folder.
4. Add the following code to the **AddTagsAndQuery.js** file, and substitute the **{iot hub connection string}** placeholder with the IoT Hub connection string you copied when you created your hub:

```
'use strict';
var iothub = require('azure-iothub');
var connectionString = '{iot hub connection string}';
var registry = iothub.Registry.fromConnectionString(connectionString);

registry.getTwin('myDeviceId', function(err, twin){
    if (err) {
        console.error(err.constructor.name + ': ' + err.message);
    } else {
        var patch = {
            tags: {
                location: {
                    region: 'US',
                    plant: 'Redmond43'
                }
            }
        };

        twin.update(patch, function(err) {
            if (err) {
                console.error('Could not update twin: ' + err.constructor.name + ': ' + err.message);
            } else {
                console.log(twin.deviceId + ' twin updated successfully');
                queryTwins();
            }
        });
    }
});
```

The **Registry** object exposes all the methods required to interact with device twins from the service. The previous code first initializes the **Registry** object, then retrieves the device twin for **myDeviceId**, and finally updates its tags with the desired location information.

After updating the tags it calls the **queryTwins** function.

5. Add the following code at the end of **AddTagsAndQuery.js** to implement the **queryTwins** function:

```

var queryTwins = function() {
    var query = registry.createQuery("SELECT * FROM devices WHERE tags.location.plant = 'Redmond43'", 100);
    query.nextAsTwin(function(err, results) {
        if (err) {
            console.error('Failed to fetch the results: ' + err.message);
        } else {
            console.log("Devices in Redmond43: " + results.map(function(twin) {return twin.deviceId}).join(',')));
        }
    });
};

query = registry.createQuery("SELECT * FROM devices WHERE tags.location.plant = 'Redmond43' AND properties.reported.connectivity.type = 'cellular'", 100);
query.nextAsTwin(function(err, results) {
    if (err) {
        console.error('Failed to fetch the results: ' + err.message);
    } else {
        console.log("Devices in Redmond43 using cellular network: " + results.map(function(twin) {return twin.deviceId}).join(',')));
    }
});

```

The previous code executes two queries: the first selects only the device twins of devices located in the **Redmond43** plant, and the second refines the query to select only the devices that are also connected through cellular network.

The previous code, when it creates the **query** object, specifies a maximum number of returned documents. The **query** object contains a **hasMoreResults** boolean property that you can use to invoke the **nextAsTwin** methods multiple times to retrieve all results. A method called **next** is available for results that are not device twins, for example, results of aggregation queries.

- Run the application with:

```
node AddTagsAndQuery.js
```

You should see one device in the results for the query asking for all devices located in **Redmond43** and none for the query that restricts the results to devices that use a cellular network.

```
$ node AddTagsAndQuery.js
myDeviceId twin updated successfully
Devices in Redmond43: myDeviceId
Devices in Redmond43 using cellular network:
```

In the next section, you create a device app that reports the connectivity information and changes the result of the query in the previous section.

Create the device app

In this section, you create a Node.js console app that connects to your hub as **myDeviceId**, and then updates its device twin's reported properties to contain the information that it is connected using a cellular network.

- Create a new empty folder called **reportconnectivity**. In the **reportconnectivity** folder, create a new package.json file using the following command at your command prompt. Accept all the defaults:

```
npm init
```

- At your command prompt in the **reportconnectivity** folder, run the following command to install the

azure-iot-device, and **azure-iot-device-mqtt** package:

```
npm install azure-iot-device azure-iot-device-mqtt --save
```

3. Using a text editor, create a new **ReportConnectivity.js** file in the **reportconnectivity** folder.
4. Add the following code to the **ReportConnectivity.js** file, and substitute the **{device connection string}** placeholder with the device connection string you copied when you created the **myDeviceId** device identity:

```
'use strict';
var Client = require('azure-iot-device').Client;
var Protocol = require('azure-iot-device-mqtt').Mqtt;

var connectionString = '{device connection string}';
var client = Client.fromConnectionString(connectionString, Protocol);

client.open(function(err) {
if (err) {
    console.error('could not open IoT Hub client');
} else {
    console.log('client opened');

    client.getTwin(function(err, twin) {
if (err) {
    console.error('could not get twin');
} else {
    var patch = {
        connectivity: {
            type: 'cellular'
        }
    };

    twin.properties.reported.update(patch, function(err) {
if (err) {
        console.error('could not update twin');
} else {
        console.log('twin state reported');
        process.exit();
    }
});
}
});
}
});
```

The **Client** object exposes all the methods you require to interact with device twins from the device. The previous code, after it initializes the **Client** object, retrieves the device twin for **myDeviceId** and updates its **reported** property with the connectivity information.

5. Run the device app

```
node ReportConnectivity.js
```

You should see the message `twin state reported`.

6. Now that the device reported its connectivity information, it should appear in both queries. Go back in the **addtagsandqueryapp** folder and run the queries again:

```
node AddTagsAndQuery.js
```

This time **myDeviceId** should appear in both query results.

```
$ node AddTagsAndQuery.js
myDeviceId twin updated successfully
Devices in Redmond43 using cellular network: myDeviceId
Devices in Redmond43: myDeviceId
```

Next steps

In this tutorial, you configured a new IoT hub in the Azure portal, and then created a device identity in the IoT hub's identity registry. You added device metadata as tags from a back-end app, and wrote a simulated device app to report device connectivity information in the device twin. You also learned how to query this information using the SQL-like IoT Hub query language.

Use the following resources to learn how to:

- send telemetry from devices with the [Get started with IoT Hub](#) tutorial,
- configure devices using device twin's desired properties with the [Use desired properties to configure devices](#) tutorial,
- control devices interactively (such as turning on a fan from a user-controlled app), with the [Use direct methods](#) tutorial.

Get started with device twins (.NET/.NET)

2/28/2019 • 11 minutes to read

Device twins are JSON documents that store device state information (metadata, configurations, and conditions). IoT Hub persists a device twin for each device that connects to it.

NOTE

The features described in this article are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

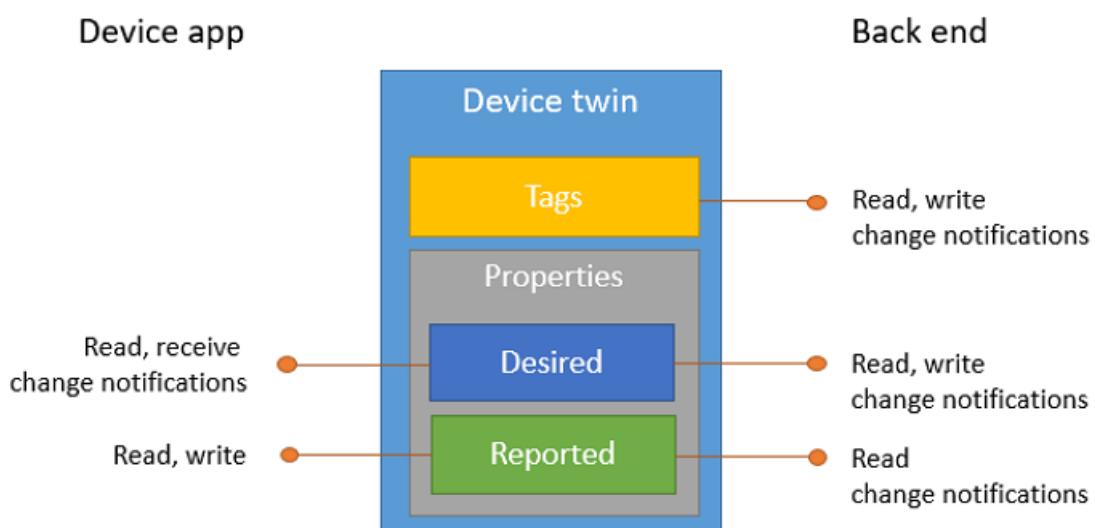
Use device twins to:

- Store device metadata from your solution back end.
- Report current state information such as available capabilities and conditions (for example, the connectivity method used) from your device app.
- Synchronize the state of long-running workflows (such as firmware and configuration updates) between a device app and a back-end app.
- Query your device metadata, configuration, or state.

Device twins are designed for synchronization and for querying device configurations and conditions. More information on when to use device twins can be found in [Understand device twins](#).

Device twins are stored in an IoT hub and contain:

- *tags*, device metadata accessible only by the solution back end;
- *desired properties*, JSON objects modifiable by the solution back end and observable by the device app; and
- *reported properties*, JSON objects modifiable by the device app and readable by the solution back end.
Tags and properties cannot contain arrays, but objects can be nested.



Additionally, the solution back end can query device twins based on all the above data. Refer to [Understand device twins](#) for more information about device twins, and to the [IoT Hub query language](#) reference for querying.

This tutorial shows you how to:

- Create a back-end app that adds *tags* to a device twin, and a simulated device app that reports its connectivity channel as a *reported property* on the device twin.
- Query devices from your back-end app using filters on the tags and properties previously created.

At the end of this tutorial, you will have these .NET console apps:

- **CreateDeviceIdentity**, a .NET app which creates a device identity and associated security key to connect your simulated device app.
- **AddTagsAndQuery**, a .NET back-end app which adds tags and queries device twins.
- **ReportConnectivity**, a .NET device app which simulates a device that connects to your IoT hub with the device identity created earlier, and reports its connectivity condition.

NOTE

The article [Azure IoT SDKs](#) provides information about the Azure IoT SDKs that you can use to build both device and back-end apps.

To complete this tutorial you need the following:

- Visual Studio 2017.
- An active Azure account. (If you don't have an account, you can create a [free account](#) in just a couple of minutes.)

Create an IoT hub

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose +**Create a resource**, then choose **Internet of Things**.
3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Review + create

Create an IoT Hub to help you connect, monitor, and manage billions of your IoT assets. [Learn More](#)

PROJECT DETAILS

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

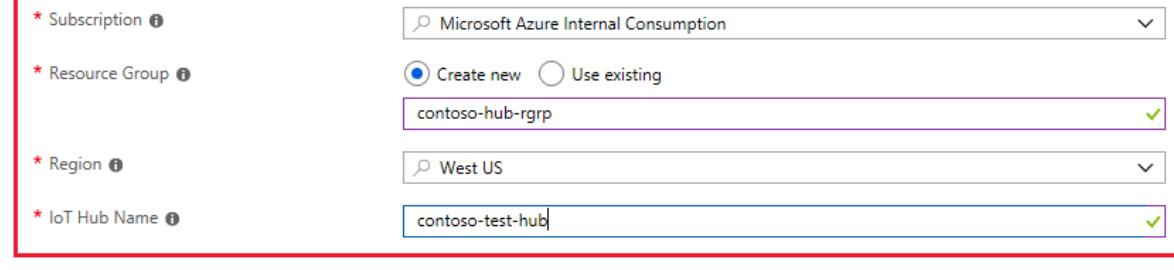
* Subscription

* Resource Group Create new Use existing

* Region

* IoT Hub Name

Review + create **Next: Size and scale »** Automation options



Fill in the fields.

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

IoT hub

Microsoft

[Basics](#) [Size and scale](#) [Review + create](#)

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS* Pricing and scale tier [?](#)

S1: Standard tier

[Learn how to choose the right IoT Hub tier for your solution](#)Number of S1 IoT Hub units [?](#)

This determines your IoT Hub scale capability and can be changed as your need increases.

Device-to-cloud-messages [?](#) EnabledMessage routing [?](#) EnabledCloud-to-device commands [?](#) EnabledIoT Edge [?](#) EnabledDevice management [?](#) Enabled[Advanced Settings](#)Device-to-cloud partitions [?](#)[Review + create](#)[« Previous: Basics](#)[Automation options](#)

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: The number of messages allowed per unit per day depends on your hub's pricing tier. For example, if you want the IoT hub to support ingress of 700,000 messages, you choose two S1 tier units.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

Advanced / Device-to-cloud partitions: This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most IoT hubs only need four partitions.

- Click **Review + create** to review your choices. You see something similar to this screen.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale **Review + create**

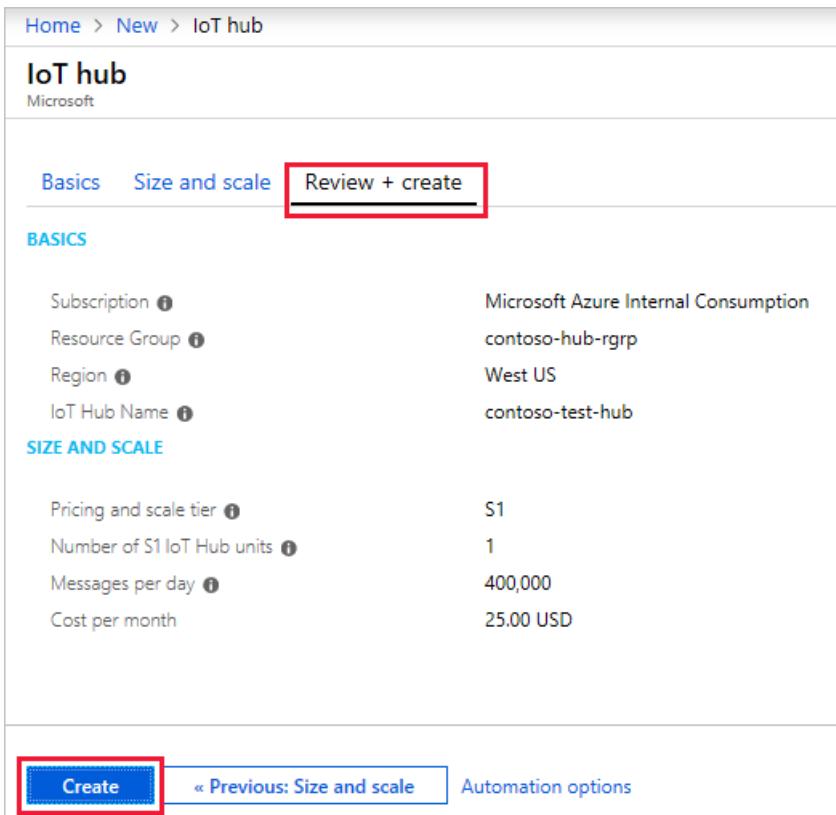
BASICS

Subscription	Microsoft Azure Internal Consumption
Resource Group	contoso-hub-rgrp
Region	West US
IoT Hub Name	contoso-test-hub

SIZE AND SCALE

Pricing and scale tier	S1
Number of S1 IoT Hub units	1
Messages per day	400,000
Cost per month	25.00 USD

Create [« Previous: Size and scale](#) [Automation options](#)



6. Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Retrieve connection string for IoT hub

After your hub has been created, retrieve the connection string for the hub. This is used to connect devices and applications to your hub.

1. Click on your hub to see the IoT Hub pane with Settings, and so on. Click **Shared access policies**.
2. In **Shared access policies**, select the **iothubowner** policy.
3. Under **Shared access keys**, copy the **Connection string -- primary key** to be used later.

Home > IoT Hub > ContosoHub - Shared access policies

ContosoHub - Shared access policies

iothubowner
ContosoHub

Add

IoT Hub uses permissions to grant access to each functionality.

Search to filter items...

POLICY

iothubowner
service
device
registryRead
registryReadWrite

Save **Discard** **Regen key** **Delete**

Access policy name: iothubowner

Permissions:

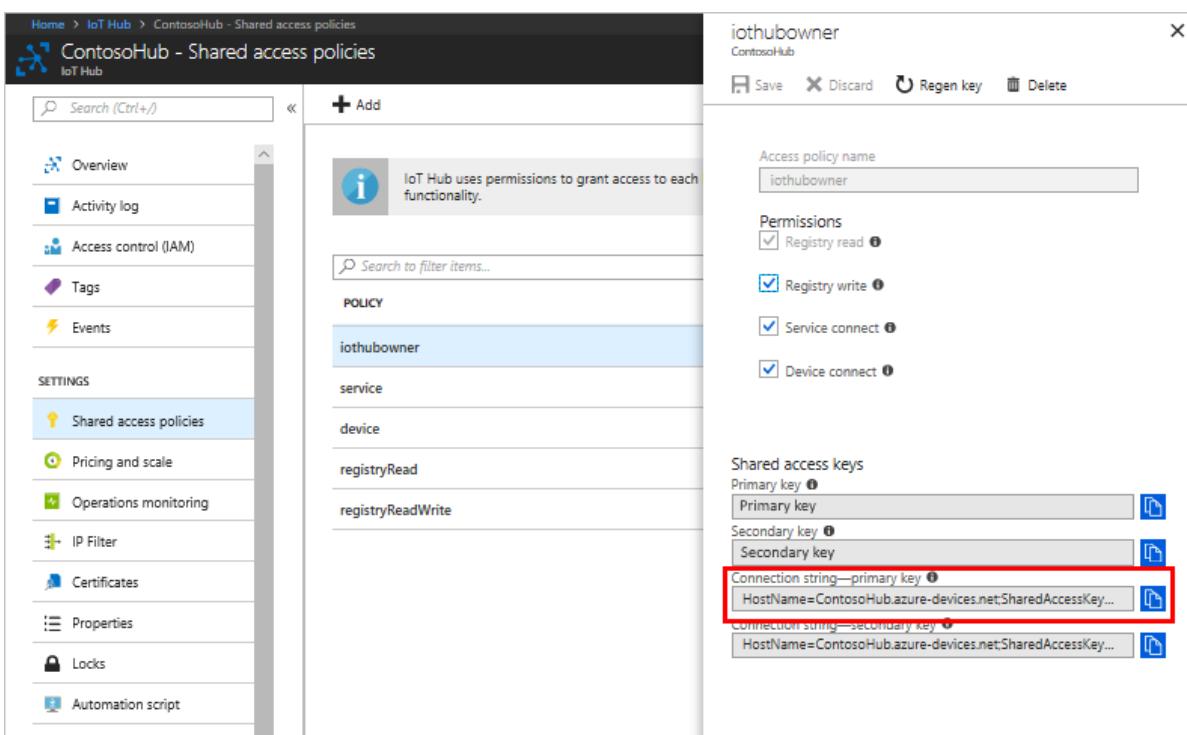
- Registry read
- Registry write
- Service connect
- Device connect

Shared access keys

Primary key	<input type="text" value="HostName=ContosoHub.azure-devices.net;SharedAccessKey..."/>
Secondary key	<input type="text" value="HostName=ContosoHub.azure-devices.net;SharedAccessKey..."/>

Connection string--primary key

Connection string--secondary key



For more information, see [Access control](#) in the "IoT Hub developer guide."

Register a new device in the IoT hub

In this section, you create a device identity in the identity registry in your IoT hub. A device cannot connect to IoT hub unless it has an entry in the identity registry. For more information, see the "Identity registry" section of the [IoT Hub developer guide](#)

1. In your IoT hub navigation menu, open **IoT Devices**, then click **Add** to register a new device in your IoT hub.

The screenshot shows the 'ContosoHub - IoT devices' blade in the Azure portal. The left sidebar contains navigation links: Events, SETTINGS (Shared access policies, Pricing and scale, Operations monitoring, IP Filter, Certificates, Properties, Locks, Automation script), EXPLORERS (Query explorer, IoT devices), and AUTOMATIC DEVICE MANAGEMENT (IoT Edge (preview)). The main area has a search bar, a red box around the '+ Add' button, and a message: 'You can use this tool to view, create, update, and delete devices on your IoT Hub.' Below is a query editor with a 'Query' section containing 'SELECT * FROM devices WHERE optional (e.g. tags.location='US')' and an 'Execute' button. At the bottom is a table header with columns: DEVICE ID, STATUS, LAST ACTIVITY, LAST STATUS..., AUTHENTICATI..., CLOUD TO DEV... followed by a row 'No results'.

2. Provide a name for your new device, such as **myDeviceId**, and click **Save**. This action creates a new device identity for your IoT hub.

Create a device

Learn more about creating devices

* Device ID [i](#)
myDeviceId ✓

Authentication type [i](#)
[Symmetric key](#) [X.509 Self-Signed](#) [X.509 CA Signed](#)

* Primary key [i](#)
Enter your primary key

* Secondary key [i](#)
Enter your secondary key

Auto-generate keys [i](#)

Connect this device to an IoT hub [i](#)
[Enable](#) [Disable](#)

Parent device (Preview) [i](#)
No parent device
[Set a parent device](#)

[Save](#)

IMPORTANT

The device ID may be visible in the logs collected for customer support and troubleshooting, so make sure to avoid any sensitive information while naming it.

3. After the device is created, open the device from the list in the **IoT devices** pane. Copy the **Connection string---primary key** to use later.

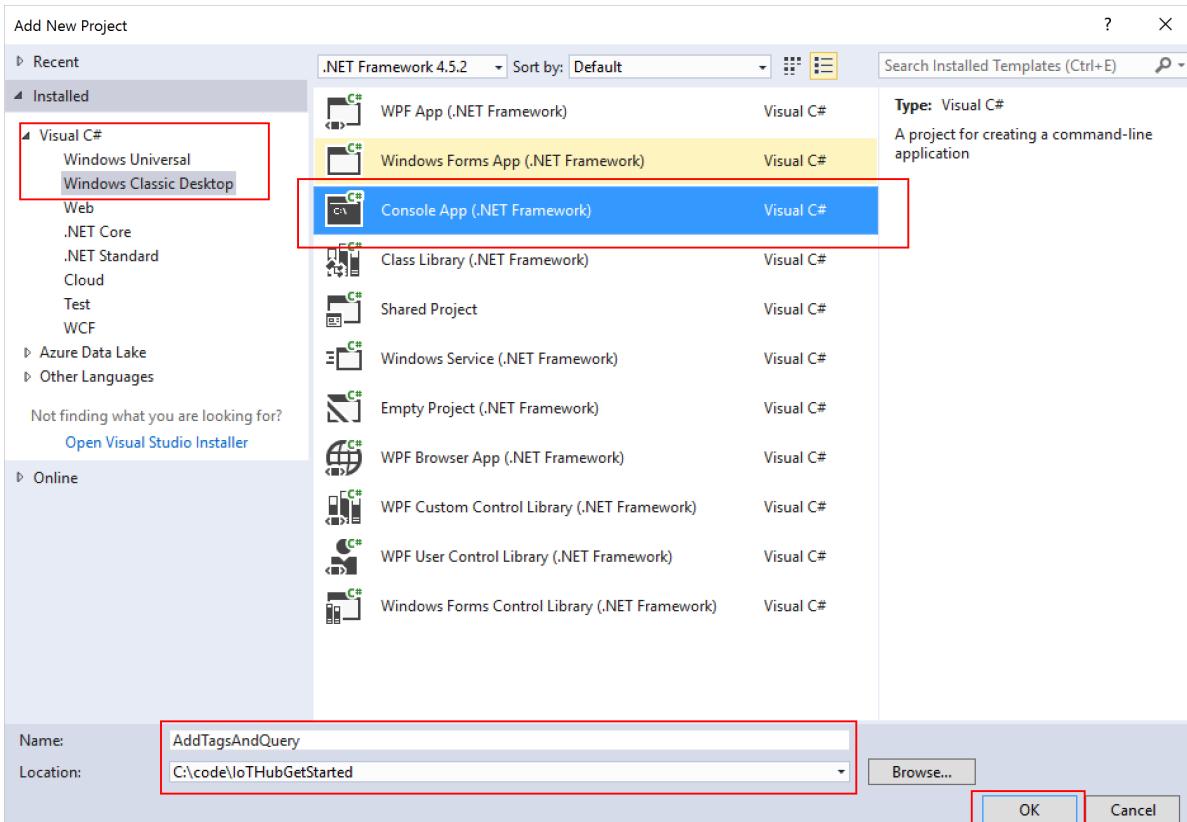
NOTE

The IoT Hub identity registry only stores device identities to enable secure access to the IoT hub. It stores device IDs and keys to use as security credentials, and an enabled/disabled flag that you can use to disable access for an individual device. If your application needs to store other device-specific metadata, it should use an application-specific store. For more information, see [IoT Hub developer guide](#).

Create the service app

In this section, you create a .NET console app (using C#) that adds location metadata to the device twin associated with **myDeviceId**. It then queries the device twins stored in the IoT hub selecting the devices located in the US, and then the ones that reported a cellular connection.

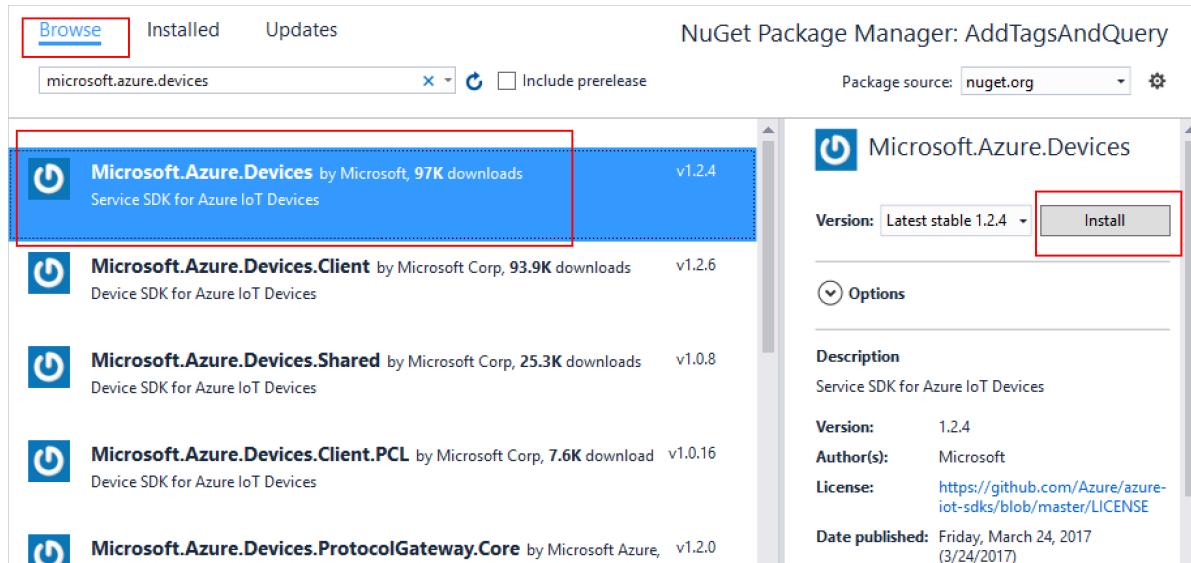
1. In Visual Studio, add a Visual C# Windows Classic Desktop project to the current solution by using the **Console Application** project template. Name the project **AddTagsAndQuery**.



2. In Solution Explorer, right-click the **AddTagsAndQuery** project, and then click **Manage NuGet**

Packages....

3. In the **NuGet Package Manager** window, select **Browse** and search for **Microsoft.Azure.Devices**. Select **Install** to install the **Microsoft.Azure.Devices** package, and accept the terms of use. This procedure downloads, installs, and adds a reference to the **Azure IoT service SDK** NuGet package and its dependencies.



4. Add the following `using` statements at the top of the **Program.cs** file:

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

5. Add the following fields to the **Program** class. Replace the placeholder value with the IoT Hub connection string for the hub that you created in the previous section.

```
static RegistryManager registryManager;  
static string connectionString = "{iot hub connection string}";
```

6. Add the following method to the **Program** class:

```

public static async Task AddTagsAndQuery()
{
    var twin = await registryManager.GetTwinAsync("myDeviceId");
    var patch =
        @"{
            tags: {
                location: {
                    region: 'US',
                    plant: 'Redmond43'
                }
            }
        }";
    await registryManager.UpdateTwinAsync(twin.DeviceId, patch, twin.ETag);

    var query = registryManager.CreateQuery(
        "SELECT * FROM devices WHERE tags.location.plant = 'Redmond43'", 100);
    var twinsInRedmond43 = await query.GetNextAsTwinAsync();
    Console.WriteLine("Devices in Redmond43: {0}",
        string.Join(", ", twinsInRedmond43.Select(t => t.DeviceId)));

    query = registryManager.CreateQuery("SELECT * FROM devices WHERE tags.location.plant = 'Redmond43' AND properties.reported.connectivity.type = 'cellular'", 100);
    var twinsInRedmond43UsingCellular = await query.GetNextAsTwinAsync();
    Console.WriteLine("Devices in Redmond43 using cellular network: {0}",
        string.Join(", ", twinsInRedmond43UsingCellular.Select(t => t.DeviceId)));
}

```

The **RegistryManager** class exposes all the methods required to interact with device twins from the service. The previous code first initializes the **registryManager** object, then retrieves the device twin for **myDeviceId**, and finally updates its tags with the desired location information.

After updating, it executes two queries: the first selects only the device twins of devices located in the **Redmond43** plant, and the second refines the query to select only the devices that are also connected through cellular network.

Note that the previous code, when it creates the **query** object, specifies a maximum number of returned documents. The **query** object contains a **HasMoreResults** boolean property that you can use to invoke the **GetNextAsTwinAsync** methods multiple times to retrieve all results. A method called **GetNextAsJson** is available for results that are not device twins, for example, results of aggregation queries.

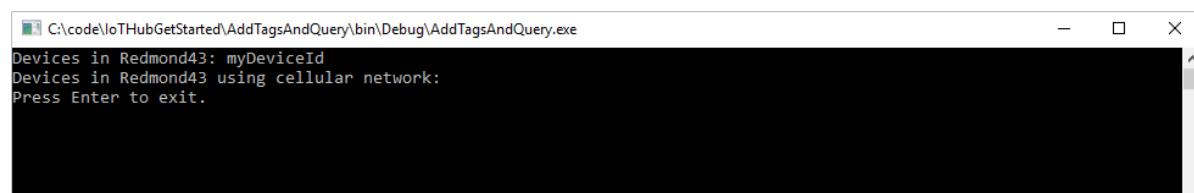
- Finally, add the following lines to the **Main** method:

```

registryManager = RegistryManager.CreateFromConnectionString(connectionString);
AddTagsAndQuery().Wait();
Console.WriteLine("Press Enter to exit.");
Console.ReadLine();

```

- In the Solution Explorer, open the **Set StartUp projects...** and make sure the **Action** for **AddTagsAndQuery** project is **Start**. Build the solution.
- Run this application by right-clicking on the **AddTagsAndQuery** project and selecting **Debug**, followed by **Start new instance**. You should see one device in the results for the query asking for all devices located in **Redmond43** and none for the query that restricts the results to devices that use a cellular network.

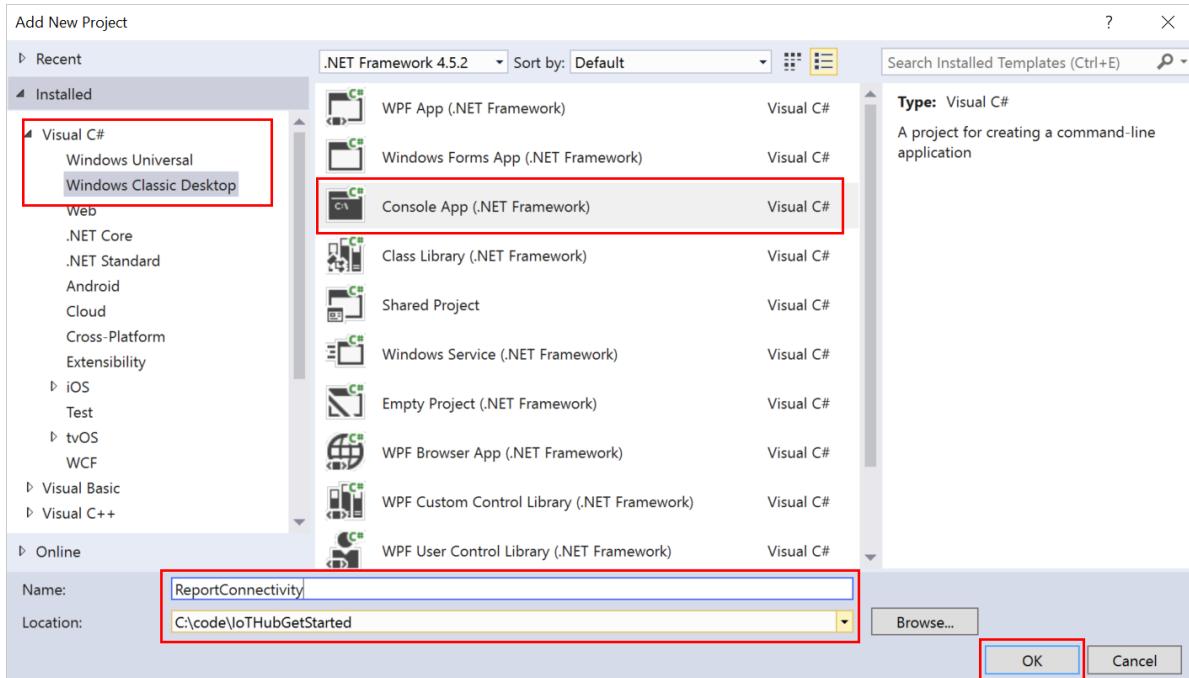


In the next section, you create a device app that reports the connectivity information and changes the result of the query in the previous section.

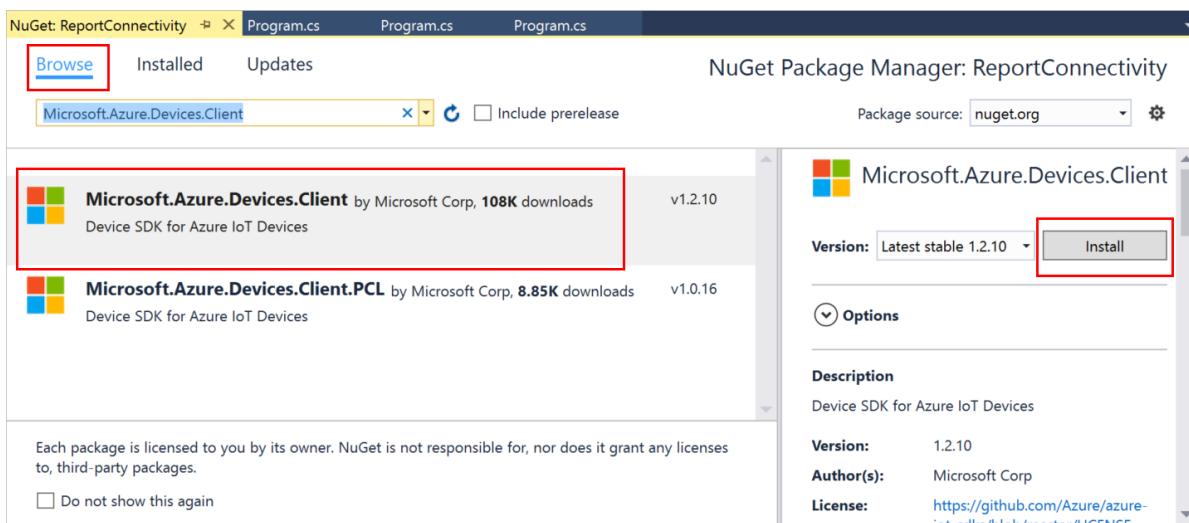
Create the device app

In this section, you create a .NET console app that connects to your hub as **myDeviceId**, and then updates its reported properties to contain the information that it is connected using a cellular network.

1. In Visual Studio, add a Visual C# Windows Classic Desktop project to the current solution by using the **Console Application** project template. Name the project **ReportConnectivity**.



2. In Solution Explorer, right-click the **ReportConnectivity** project, and then click **Manage NuGet Packages...**
3. In the **NuGet Package Manager** window, select **Browse** and search for **Microsoft.Azure.Devices.Client**. Select **Install** to install the **Microsoft.Azure.Devices.Client** package, and accept the terms of use. This procedure downloads, installs, and adds a reference to the [Azure IoT device SDK](#) NuGet package and its dependencies.



4. Add the following `using` statements at the top of the **Program.cs** file:

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

5. Add the following fields to the **Program** class. Replace the placeholder value with the device connection string that you noted in the previous section.

```
static string DeviceConnectionString = "HostName=<yourIoTHubName>.azure-devices.net;
DeviceId=<yourIoTDeviceName>;SharedAccessKey=<yourIoTDeviceAccessKey>";
static DeviceClient Client = null;
```

6. Add the following method to the **Program** class:

```
public static async void InitClient()
{
    try
    {
        Console.WriteLine("Connecting to hub");
        Client = DeviceClient.CreateFromConnectionString(DeviceConnectionString,
            TransportType.Mqtt);
        Console.WriteLine("Retrieving twin");
        await Client.GetTwinAsync();
    }
    catch (Exception ex)
    {
        Console.WriteLine();
        Console.WriteLine("Error in sample: {0}", ex.Message);
    }
}
```

The **Client** object exposes all the methods you require to interact with device twins from the device. The code shown above, initializes the **Client** object, and then retrieves the device twin for **myDeviceId**.

7. Add the following method to the **Program** class:

```
public static async void ReportConnectivity()
{
    try
    {
        Console.WriteLine("Sending connectivity data as reported property");

        TwinCollection reportedProperties, connectivity;
        reportedProperties = new TwinCollection();
        connectivity = new TwinCollection();
        connectivity["type"] = "cellular";
        reportedProperties["connectivity"] = connectivity;
        await Client.UpdateReportedPropertiesAsync(reportedProperties);
    }
    catch (Exception ex)
    {
        Console.WriteLine();
        Console.WriteLine("Error in sample: {0}", ex.Message);
    }
}
```

The code above updates **myDeviceId**'s reported property with the connectivity information.

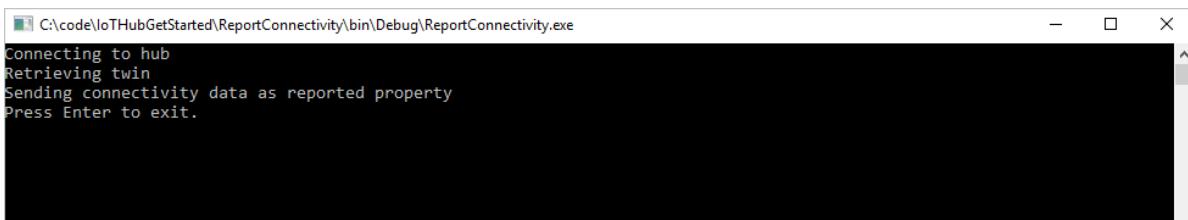
8. Finally, add the following lines to the **Main** method:

```

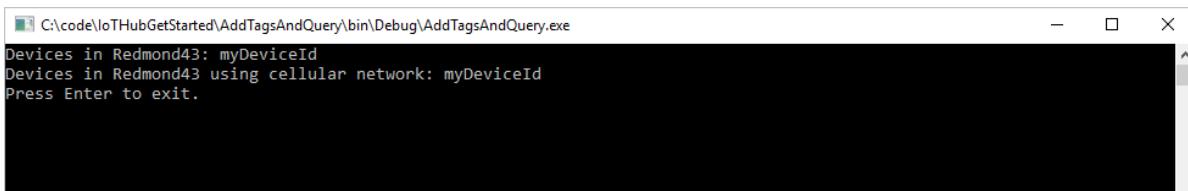
try
{
    InitClient();
    ReportConnectivity();
}
catch (Exception ex)
{
    Console.WriteLine();
    Console.WriteLine("Error in sample: {0}", ex.Message);
}
Console.WriteLine("Press Enter to exit.");
Console.ReadLine();

```

9. In the Solution Explorer, open the **Set StartUp projects...** and make sure the **Action** for **ReportConnectivity** project is **Start**. Build the solution.
10. Run this application by right-clicking on the **ReportConnectivity** project and selecting **Debug**, followed by **Start new instance**. You should see it getting the twin information, and then sending connectivity as a *reported property*.



11. Now that the device reported its connectivity information, it should appear in both queries. Run the .NET **AddTagsAndQuery** app to run the queries again. This time **myDeviceId** should appear in both query results.



Next steps

In this tutorial, you configured a new IoT hub in the Azure portal, and then created a device identity in the IoT hub's identity registry. You added device metadata as tags from a back-end app, and wrote a simulated device app to report device connectivity information in the device twin. You also learned how to query this information using the SQL-like IoT Hub query language.

Use the following resources to learn how to:

- send telemetry from devices with the [Send telemetry from a device to an IoT hub](#) tutorial,
- configure devices using device twin's desired properties with the [Use desired properties to configure devices](#) tutorial,
- control devices interactively (such as turning on a fan from a user-controlled app) with the [Use direct methods](#) tutorial.

Get started with device twins (Java)

3/6/2019 • 12 minutes to read

Device twins are JSON documents that store device state information (metadata, configurations, and conditions). IoT Hub persists a device twin for each device that connects to it.

NOTE

The features described in this article are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

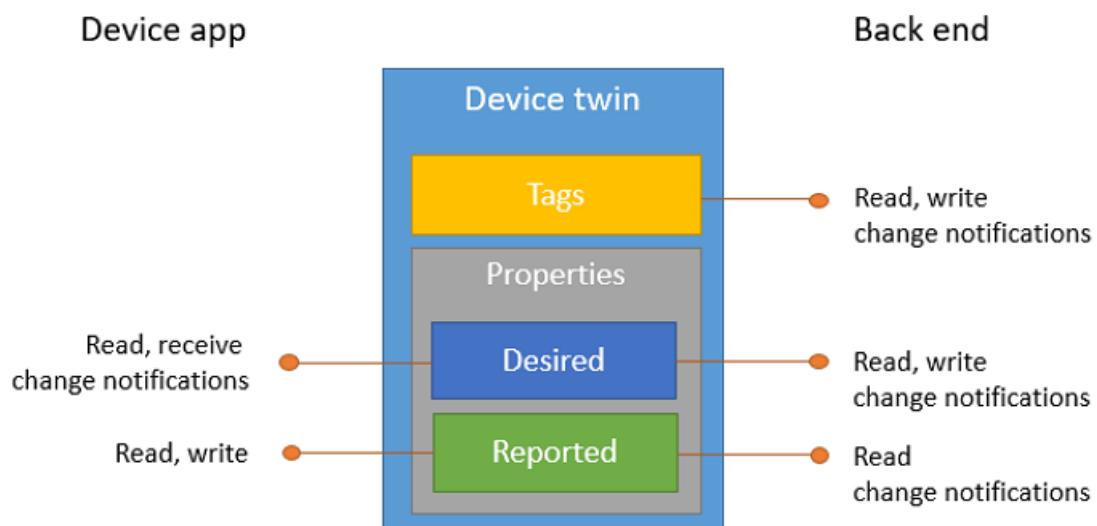
Use device twins to:

- Store device metadata from your solution back end.
- Report current state information such as available capabilities and conditions (for example, the connectivity method used) from your device app.
- Synchronize the state of long-running workflows (such as firmware and configuration updates) between a device app and a back-end app.
- Query your device metadata, configuration, or state.

Device twins are designed for synchronization and for querying device configurations and conditions. More information on when to use device twins can be found in [Understand device twins](#).

Device twins are stored in an IoT hub and contain:

- *tags*, device metadata accessible only by the solution back end;
- *desired properties*, JSON objects modifiable by the solution back end and observable by the device app; and
- *reported properties*, JSON objects modifiable by the device app and readable by the solution back end.
Tags and properties cannot contain arrays, but objects can be nested.



Additionally, the solution back end can query device twins based on all the above data. Refer to [Understand device twins](#) for more information about device twins, and to the [IoT Hub query language](#) reference for querying.

This tutorial shows you how to:

- Create a back-end app that adds *tags* to a device twin, and a simulated device app that reports its connectivity channel as a *reported property* on the device twin.
- Query devices from your back-end app using filters on the tags and properties previously created.

In this tutorial, you create two Java console apps:

- **add-tags-query**, a Java back-end app that adds tags and queries device twins.
- **simulated-device**, a Java device app that connects to your IoT hub and reports its connectivity condition using a reported property.

NOTE

The article [Azure IoT SDKs](#) provides information about the Azure IoT SDKs that you can use to build both device and back-end apps.

To complete this tutorial, you need:

- The latest [Java SE Development Kit 8](#)
- [Maven 3](#)
- An active Azure account. (If you don't have an account, you can create a [free account](#) in just a couple of minutes.)

Create an IoT hub

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose **+Create a resource**, then choose **Internet of Things**.
3. Click **Iot Hub** from the list on the right. You see the first screen for creating an IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Review + create

Create an IoT Hub to help you connect, monitor, and manage billions of your IoT assets. [Learn More](#)

PROJECT DETAILS

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

* Subscription

* Resource Group Create new Use existing

* Region

* IoT Hub Name

Review + create **Next: Size and scale »** Automation options

Fill in the fields.

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Review + create

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS

* Pricing and scale tier [?](#) S1: Standard tier [?](#) [Learn how to choose the right IoT Hub tier for your solution](#)

Number of S1 IoT Hub units [?](#) 1 [1](#) This determines your IoT Hub scale capability and can be changed as your need increases.

Device-to-cloud-messages [?](#) Enabled

Message routing [?](#) Enabled

Cloud-to-device commands [?](#) Enabled

IoT Edge [?](#) Enabled

Device management [?](#) Enabled

Advanced Settings

Device-to-cloud partitions [?](#) 4 [4](#)

[Review + create](#) [« Previous: Basics](#) [Automation options](#)

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: The number of messages allowed per unit per day depends on your hub's pricing tier. For example, if you want the IoT hub to support ingress of 700,000 messages, you choose two S1 tier units.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

Advanced / Device-to-cloud partitions: This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most IoT hubs only need four partitions.

5. Click **Review + create** to review your choices. You see something similar to this screen.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale **Review + create**

BASICS

Subscription	Microsoft Azure Internal Consumption
Resource Group	contoso-hub-rgrp
Region	West US
IoT Hub Name	contoso-test-hub

SIZE AND SCALE

Pricing and scale tier	S1
Number of S1 IoT Hub units	1
Messages per day	400,000
Cost per month	25.00 USD

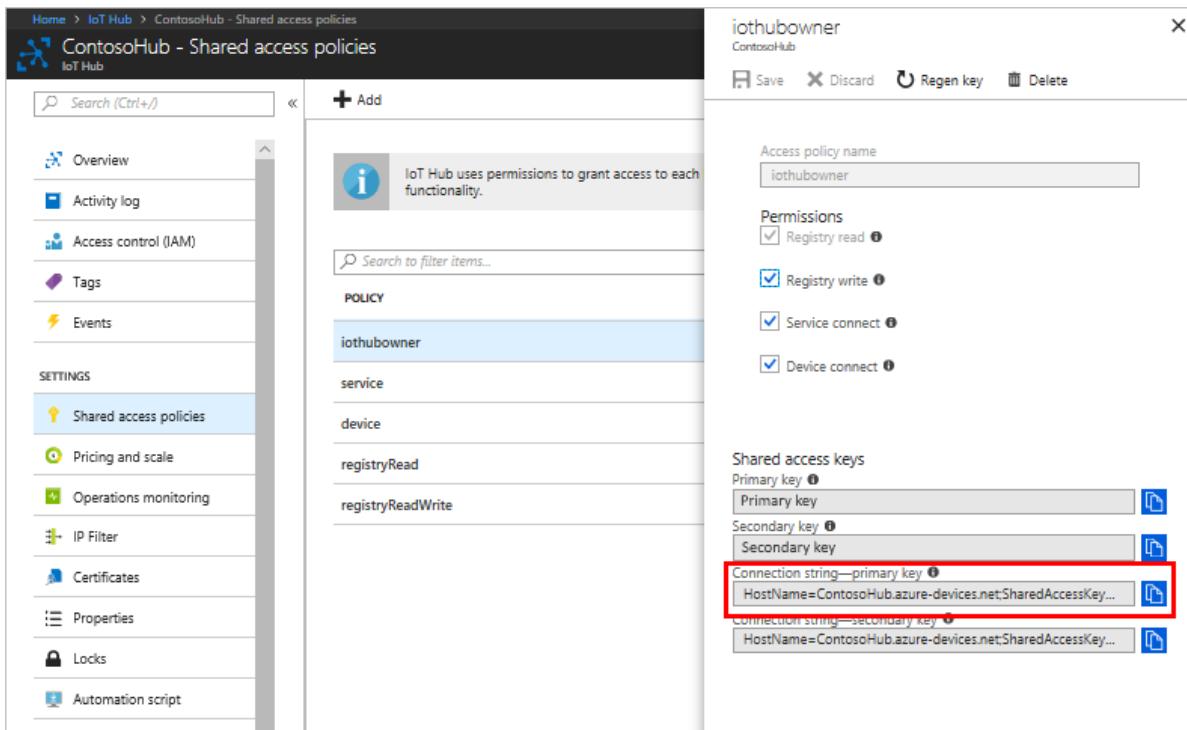
Create « Previous: Size and scale Automation options

6. Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Retrieve connection string for IoT hub

After your hub has been created, retrieve the connection string for the hub. This is used to connect devices and applications to your hub.

1. Click on your hub to see the IoT Hub pane with Settings, and so on. Click **Shared access policies**.
2. In **Shared access policies**, select the **iothubowner** policy.
3. Under **Shared access keys**, copy the **Connection string -- primary key** to be used later.



Home > IoT Hub > ContosoHub - Shared access policies

ContosoHub - Shared access policies

IoT Hub

Overview Activity log Access control (IAM) Tags Events

SETTINGS

- Shared access policies **Selected**
- Pricing and scale
- Operations monitoring
- IP Filter
- Certificates
- Properties
- Locks
- Automation script

Add

IoT Hub uses permissions to grant access to each functionality.

Search to filter items...

POLICY

iothubowner

service

device

registryRead

registryReadWrite

iothubowner

Save Discard Regen key Delete

Access policy name: iothubowner

Permissions:

- Registry read
- Registry write
- Service connect
- Device connect

Shared access keys:

Primary key: [Primary key](#)

Secondary key: [Secondary key](#)

Connection string—primary key: [Connection string—primary key](#)

HostName=ContosoHub.azure-devices.net;SharedAccessKey=...

Connection string—secondary key: [Connection string—secondary key](#)

HostName=ContosoHub.azure-devices.net;SharedAccessKey=...

For more information, see [Access control](#) in the "IoT Hub developer guide."

Register a new device in the IoT hub

In this section, you create a device identity in the identity registry in your IoT hub. A device cannot connect to IoT hub unless it has an entry in the identity registry. For more information, see the "Identity registry" section of the [IoT Hub developer guide](#)

1. In your IoT hub navigation menu, open **IoT Devices**, then click **Add** to register a new device in your IoT hub.

The screenshot shows the 'ContosoHub - IoT devices' blade in the Azure portal. The left sidebar contains several sections: 'Events', 'SETTINGS' (with options like Shared access policies, Pricing and scale, Operations monitoring, IP Filter, Certificates, Properties, Locks, and Automation script), 'EXPLORERS' (with Query explorer selected and highlighted with a red box), and 'AUTOMATIC DEVICE MANAGEMENT' (with IoT Edge (preview) listed). The main area features a large 'Add' button at the top center, also highlighted with a red box. Below it is a query editor with a placeholder 'optional (e.g. tags.location='US')' and an 'Execute' button. A message states: 'You can use this tool to view, create, update, and delete devices on your IoT Hub.' At the bottom, there is a table header with columns: DEVICE ID, STATUS, LAST ACTIVITY, LAST STATUS..., AUTHENTICATI..., and CLOUD TO DEV... followed by the message 'No results'.

2. Provide a name for your new device, such as **myDeviceId**, and click **Save**. This action creates a new device identity for your IoT hub.



Create a device

□ X



Learn more about creating devices



* Device ID ⓘ

myDeviceId



Authentication type ⓘ

Symmetric key

X.509 Self-Signed

X.509 CA Signed

* Primary key ⓘ

Enter your primary key

* Secondary key ⓘ

Enter your secondary key

Auto-generate keys ⓘ



Connect this device to an IoT hub ⓘ

Enable

Disable

Parent device (Preview) ⓘ

No parent device

[Set a parent device](#)

[Save](#)

IMPORTANT

The device ID may be visible in the logs collected for customer support and troubleshooting, so make sure to avoid any sensitive information while naming it.

3. After the device is created, open the device from the list in the **IoT devices** pane. Copy the **Connection string---primary key** to use later.

The screenshot shows the 'Device details' page in the Azure portal. At the top, there are navigation links: Save, Message to device, Direct method, Device twin, Add module identity, Regenerate keys, and Refresh. Below these are fields for Device Id (myDeviceId), Primary key (<Primary Key>), and Secondary key (<Secondary Key>). A red box highlights the 'Connection string (primary key)' field, which contains the value 'HostName=ContosoHub.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=<Primary Key>'. Another connection string field below it contains 'HostName=ContosoHub.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=<Secondary Key>'. There are also buttons for 'Connect this device to an IoT hub' (Enable or Disable), 'Parent device (Preview)', and 'Set a parent device'.

NOTE

The IoT Hub identity registry only stores device identities to enable secure access to the IoT hub. It stores device IDs and keys to use as security credentials, and an enabled/disabled flag that you can use to disable access for an individual device. If your application needs to store other device-specific metadata, it should use an application-specific store. For more information, see [IoT Hub developer guide](#).

Create the service app

In this section, you create a Java app that adds location metadata as a tag to the device twin in IoT Hub associated with **myDeviceId**. The app first queries IoT hub for devices located in the US, and then for devices that report a cellular network connection.

1. On your development machine, create an empty folder called `iot-java-twin-getstarted`.
2. In the `iot-java-twin-getstarted` folder, create a Maven project called **add-tags-query** using the following command at your command prompt. Note this is a single, long command:

```
mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=add-tags-query -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

3. At your command prompt, navigate to the `add-tags-query` folder.
4. Using a text editor, open the `pom.xml` file in the `add-tags-query` folder and add the following dependency to the **dependencies** node. This dependency enables you to use the **iot-service-client** package in your app to communicate with your IoT hub:

```
<dependency>
  <groupId>com.microsoft.azure.sdk.iot</groupId>
  <artifactId>iot-service-client</artifactId>
  <version>1.7.23</version>
  <type>jar</type>
</dependency>
```

NOTE

You can check for the latest version of **iot-service-client** using [Maven search](#).

5. Add the following **build** node after the **dependencies** node. This configuration instructs Maven to use

Java 1.8 to build the app:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.3</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

6. Save and close the `pom.xml` file.

7. Using a text editor, open the `add-tags-query\src\main\java\com\mycompany\app\App.java` file.

8. Add the following **import** statements to the file:

```
import com.microsoft.azure.sdk.iot.service.devicetwin.*;
import com.microsoft.azure.sdk.iot.service.exceptions.IotHubException;

import java.io.IOException;
import java.util.HashSet;
import java.util.Set;
```

9. Add the following class-level variables to the **App** class. Replace `{youriothubconnectionstring}` with your IoT hub connection string you noted in the *Create an IoT Hub* section:

```
public static final String iotHubConnectionString = "{youriothubconnectionstring}";
public static final String deviceId = "myDeviceId";

public static final String region = "US";
public static final String plant = "Redmond43";
```

10. Update the **main** method signature to include the following `throws` clause:

```
public static void main( String[] args ) throws IOException
```

11. Add the following code to the **main** method to create the **DeviceTwin** and **DeviceTwinDevice** objects. The **DeviceTwin** object handles the communication with your IoT hub. The **DeviceTwinDevice** object represents the device twin with its properties and tags:

```
// Get the DeviceTwin and DeviceTwinDevice objects
DeviceTwin twinClient = DeviceTwin.createFromConnectionString(iotHubConnectionString);
DeviceTwinDevice device = new DeviceTwinDevice(deviceId);
```

12. Add the following `try/catch` block to the **main** method:

```
try {  
    // Code goes here  
} catch (IoTHubException e) {  
    System.out.println(e.getMessage());  
} catch (IOException e) {  
    System.out.println(e.getMessage());  
}
```

13. To update the **region** and **plant** device twin tags in your device twin, add the following code in the `try` block:

```
// Get the device twin from IoT Hub  
System.out.println("Device twin before update:");  
twinClient.getTwin(device);  
System.out.println(device);  
  
// Update device twin tags if they are different  
// from the existing values  
String currentTags = device.tagsToString();  
if ((!currentTags.contains("region=" + region) && !currentTags.contains("plant=" + plant))) {  
    // Create the tags and attach them to the DeviceTwinDevice object  
    Set<Pair> tags = new HashSet<Pair>();  
    tags.add(new Pair("region", region));  
    tags.add(new Pair("plant", plant));  
    device.setTags(tags);  
  
    // Update the device twin in IoT Hub  
    System.out.println("Updating device twin");  
    twinClient.updateTwin(device);  
}  
  
// Retrieve the device twin with the tag values from IoT Hub  
System.out.println("Device twin after update:");  
twinClient.getTwin(device);  
System.out.println(device);
```

14. To query the device twins in IoT hub, add the following code to the `try` block after the code you added in the previous step. The code runs two queries. Each query returns a maximum of 100 devices:

```

// Query the device twins in IoT Hub
System.out.println("Devices in Redmond:");

// Construct the query
SqlQuery sqlQuery = SqlQuery.createSqlQuery("*", SqlQuery.FromType.DEVICES, "tags.plant='Redmond43'", null);

// Run the query, returning a maximum of 100 devices
Query twinQuery = twinClient.queryTwin(sqlQuery.getQuery(), 100);
while (twinClient.hasNextDeviceTwin(twinQuery)) {
    DeviceTwinDevice d = twinClient.getNextDeviceTwin(twinQuery);
    System.out.println(d.getDeviceId());
}

System.out.println("Devices in Redmond using a cellular network:");

// Construct the query
sqlQuery = SqlQuery.createSqlQuery("*", SqlQuery.FromType.DEVICES, "tags.plant='Redmond43' AND properties.reported.connectivityType = 'cellular'", null);

// Run the query, returning a maximum of 100 devices
twinQuery = twinClient.queryTwin(sqlQuery.getQuery(), 3);
while (twinClient.hasNextDeviceTwin(twinQuery)) {
    DeviceTwinDevice d = twinClient.getNextDeviceTwin(twinQuery);
    System.out.println(d.getDeviceId());
}

```

15. Save and close the `add-tags-query\src\main\java\com\mycompany\app\App.java` file

16. Build the **add-tags-query** app and correct any errors. At your command prompt, navigate to the `add-tags-query` folder and run the following command:

```
mvn clean package -DskipTests
```

Create a device app

In this section, you create a Java console app that sets a reported property value that is sent to IoT Hub.

1. In the `iot-java-twin-getstarted` folder, create a Maven project called **simulated-device** using the following command at your command prompt. Note this is a single, long command:

```
mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=simulated-device -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

2. At your command prompt, navigate to the `simulated-device` folder.

3. Using a text editor, open the `pom.xml` file in the `simulated-device` folder and add the following dependencies to the **dependencies** node. This dependency enables you to use the **iot-device-client** package in your app to communicate with your IoT hub:

```

<dependency>
    <groupId>com.microsoft.azure.sdk.iot</groupId>
    <artifactId>iot-device-client</artifactId>
    <version>1.14.2</version>
</dependency>
```

NOTE

You can check for the latest version of **iot-device-client** using [Maven search](#).

4. Add the following **build** node after the **dependencies** node. This configuration instructs Maven to use Java 1.8 to build the app:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.3</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

5. Save and close the `pom.xml` file.
6. Using a text editor, open the `simulated-device\src\main\java\com\mycompany\app\App.java` file.
7. Add the following **import** statements to the file:

```
import com.microsoft.azure.sdk.iot.device.*;
import com.microsoft.azure.sdk.iot.device.DeviceTwin.*;

import java.io.IOException;
import java.net.URISyntaxException;
import java.util.Scanner;
```

8. Add the following class-level variables to the **App** class. Replacing `{youriothubname}` with your IoT hub name, and `{yourdevicekey}` with the device key value you generated in the *Create a device identity* section:

```
private static String connString = "HostName={youriothubname}.azure-devices.net;DeviceId=myDeviceID;SharedAccessKey={yourdevicekey}";
private static IoTHubClientProtocol protocol = IoTHubClientProtocol.MQTT;
private static String deviceId = "myDeviceId";
```

This sample app uses the **protocol** variable when it instantiates a **DeviceClient** object.

9. Add the following method to the **App** class to print information about twin updates:

```
protected static class DeviceTwinStatusCallBack implements IoTHubEventCallback {
  @Override
  public void execute(IoTHubStatusCode status, Object context) {
    System.out.println("IoT Hub responded to device twin operation with status " + status.name());
  }
}
```

10. Add the following code to the **main** method to:

- Create a device client to communicate with IoT Hub.
- Create a **Device** object to store the device twin properties.

```

DeviceClient client = new DeviceClient(connString, protocol);

// Create a Device object to store the device twin properties
Device dataCollector = new Device() {
    // Print details when a property value changes
    @Override
    public void PropertyCall(String propertyKey, Object PropertyValue, Object context) {
        System.out.println(propertyKey + " changed to " + PropertyValue);
    }
};

```

11. Add the following code to the **main** method to create a **connectivityType** reported property and send it to IoT Hub:

```

try {
    // Open the DeviceClient and start the device twin services.
    client.open();
    client.startDeviceTwin(new DeviceTwinStatusCallBack(), null, dataCollector, null);

    // Create a reported property and send it to your IoT hub.
    dataCollector.setReportedProp(new Property("connectivityType", "cellular"));
    client.sendReportedProperties(dataCollector.getReportedProp());
}
catch (Exception e) {
    System.out.println("On exception, shutting down \n" + " Cause: " + e.getCause() + " \n" +
e.getMessage());
    dataCollector.clean();
    client.closeNow();
    System.out.println("Shutting down...");
}

```

12. Add the following code to the end of the **main** method. Waiting for the **Enter** key allows time for IoT Hub to report the status of the device twin operations:

```

System.out.println("Press any key to exit...");

Scanner scanner = new Scanner(System.in);
scanner.nextLine();

dataCollector.clean();
client.close();

```

13. Modify the signature of the **main** method to include the exceptions as follows:

```

public static void main(String[] args) throws URISyntaxException, IOException

```

14. Save and close the `simulated-device\src\main\java\com\mycompany\app\App.java` file.

15. Build the **simulated-device** app and correct any errors. At your command prompt, navigate to the `simulated-device` folder and run the following command:

```

mvn clean package -DskipTests

```

Run the apps

You are now ready to run the console apps.

- At a command prompt in the `add-tags-query` folder, run the following command to run the **add-tags-query** service app:

```
mvn exec:java -Dexec.mainClass="com.mycompany.app.App"
```

```
[INFO] -----
[INFO] -----
[INFO] --- exec-maven-plugin:1.4.0:java (default-cli) @ add-tags-query ---
Device twin before update:
Device ID: myFirstJavaDevice
Tags:{}
Reported Properties{connectivityType=WiFi}

Updating device twin
Device twin after update:
Device ID: myFirstJavaDevice
Tags:{plant=Redmond43, region=US}
Reported Properties{connectivityType=WiFi}

Devices in Redmond:
myFirstJavaDevice
Devices in Redmond using a cellular network:
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 4.918 s
[INFO] Finished at: 2017-07-05T15:06:58+01:00
[INFO] Final Memory: 11M/266M
[INFO] -----
c:\repos\samples\iot-java-twin-getstarted\add-tags-query>
```

You can see the **plant** and **region** tags added to the device twin. The first query returns your device, but the second does not.

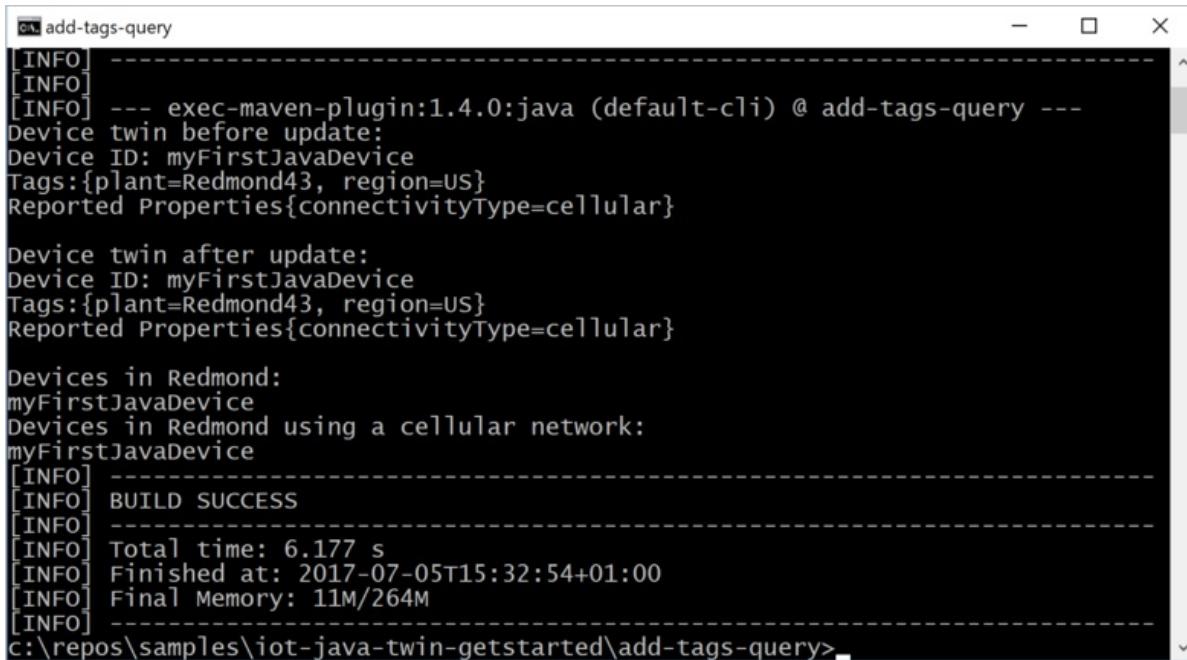
- At a command prompt in the `simulated-device` folder, run the following command to add the **connectivityType** reported property to the device twin:

```
mvn exec:java -Dexec.mainClass="com.mycompany.app.App"
```

```
c:\repos\samples\iot-java-twin-getstarted\simulated-device>mvn exec:java -Dexec.mainClass="com.mycompany.app.App"
[INFO] Scanning for projects...
[INFO] -----
[INFO] Building simulated-device 1.0-SNAPSHOT
[INFO] -----
[INFO] --- exec-maven-plugin:1.4.0:java (default-cli) @ simulated-device ---
log4j:WARN No appenders could be found for logger (com.microsoft.azure.sdk.iot.device.IotHubConnectionString).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
IoT Hub responded to device twin operation with status OK_EMPTY
Press any key to exit...
IoT Hub responded to device twin operation with status OK_EMPTY
IoT Hub responded to device twin operation with status OK
IoT Hub responded to device twin operation with status OK_EMPTY
```

- At a command prompt in the `add-tags-query` folder, run the following command to run the **add-tags-query** service app a second time:

```
mvn exec:java -Dexec.mainClass="com.mycompany.app.App"
```



```
c:\ add-tags-query
[INFO] -----
[INFO]
[INFO] --- exec-maven-plugin:1.4.0:java (default-cli) @ add-tags-query ---
Device twin before update:
Device ID: myFirstJavaDevice
Tags:{plant=Redmond43, region=US}
Reported Properties{connectivityType=cellular}

Device twin after update:
Device ID: myFirstJavaDevice
Tags:{plant=Redmond43, region=US}
Reported Properties{connectivityType=cellular}

Devices in Redmond:
myFirstJavaDevice
Devices in Redmond using a cellular network:
myFirstJavaDevice
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 6.177 s
[INFO] Finished at: 2017-07-05T15:32:54+01:00
[INFO] Final Memory: 11M/264M
[INFO] -----
c:\repos\samples\iot-java-twin-getstarted\add-tags-query>
```

Now your device has sent the **connectivityType** property to IoT Hub, the second query returns your device.

Next steps

In this tutorial, you configured a new IoT hub in the Azure portal, and then created a device identity in the IoT hub's identity registry. You added device metadata as tags from a back-end app, and wrote a device app to report device connectivity information in the device twin. You also learned how to query the device twin information using the SQL-like IoT Hub query language.

Use the following resources to learn how to:

- Send telemetry from devices with the [Get started with IoT Hub](#) tutorial.
- Control devices interactively (such as turning on a fan from a user-controlled app) with the [Use direct methods](#) tutorial.

Get started with device twins (Python)

3/6/2019 • 11 minutes to read

Device twins are JSON documents that store device state information (metadata, configurations, and conditions). IoT Hub persists a device twin for each device that connects to it.

NOTE

The features described in this article are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

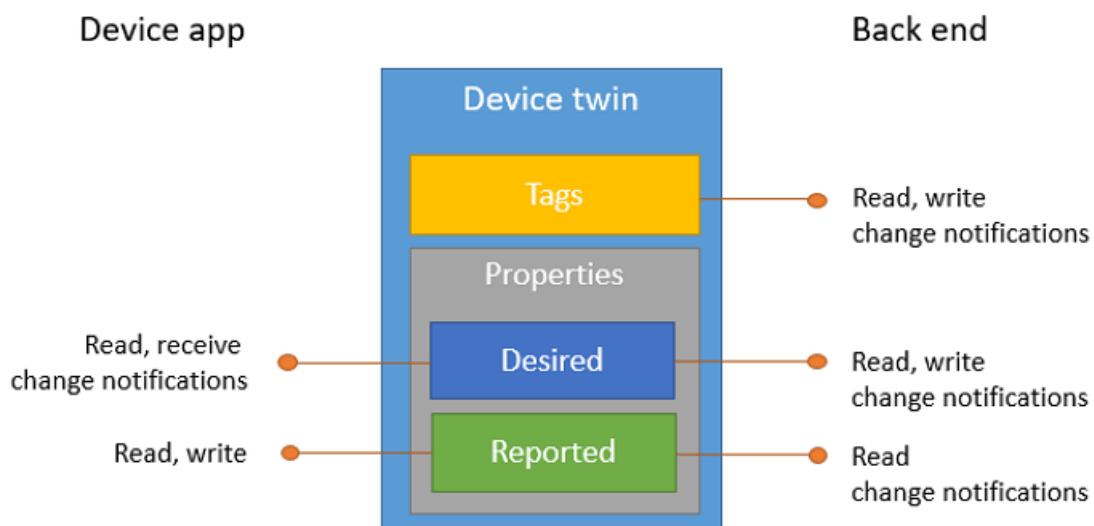
Use device twins to:

- Store device metadata from your solution back end.
- Report current state information such as available capabilities and conditions (for example, the connectivity method used) from your device app.
- Synchronize the state of long-running workflows (such as firmware and configuration updates) between a device app and a back-end app.
- Query your device metadata, configuration, or state.

Device twins are designed for synchronization and for querying device configurations and conditions. More information on when to use device twins can be found in [Understand device twins](#).

Device twins are stored in an IoT hub and contain:

- *tags*, device metadata accessible only by the solution back end;
- *desired properties*, JSON objects modifiable by the solution back end and observable by the device app; and
- *reported properties*, JSON objects modifiable by the device app and readable by the solution back end. Tags and properties cannot contain arrays, but objects can be nested.



Additionally, the solution back end can query device twins based on all the above data. Refer to [Understand device twins](#) for more information about device twins, and to the [IoT Hub query language](#) reference for querying.

This tutorial shows you how to:

- Create a back-end app that adds *tags* to a device twin, and a simulated device app that reports its connectivity channel as a *reported property* on the device twin.
- Query devices from your back-end app using filters on the tags and properties previously created.

At the end of this tutorial, you will have two Python console apps:

- **AddTagsAndQuery.py**, a Python back-end app, which adds tags and queries device twins.
- **ReportConnectivity.py**, a Python app, which simulates a device that connects to your IoT hub with the device identity created earlier, and reports its connectivity condition.

NOTE

The article [Azure IoT SDKs](#) provides information about the Azure IoT SDKs that you can use to build both device and back-end apps.

To complete this tutorial you need the following:

- [Python 2.x or 3.x](#). Make sure to use the 32-bit or 64-bit installation as required by your setup. When prompted during the installation, make sure to add Python to your platform-specific environment variable. If you are using Python 2.x, you may need to [install or upgrade pip, the Python package management system](#).
- If you are using Windows OS, then [Visual C++ redistributable package](#) to allow the use of native DLLs from Python.
- An active Azure account. (If you don't have an account, you can create a [free account](#) in just a couple of minutes.)

NOTE

The *pip* packages for `azure-iothub-service-client` and `azure-iothub-device-client` are currently available only for Windows OS. For Linux/Mac OS, please refer to the Linux and Mac OS-specific sections on the [Prepare your development environment for Python](#) post.

Create an IoT hub

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose +**Create a resource**, then choose **Internet of Things**.
3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Review + create

Create an IoT Hub to help you connect, monitor, and manage billions of your IoT assets. [Learn More](#)

PROJECT DETAILS

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

* Subscription

* Resource Group Create new Use existing

* Region

* IoT Hub Name

Review + create **Next: Size and scale »** Automation options

Fill in the fields.

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Review + create

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS

* Pricing and scale tier [?](#) S1: Standard tier [?](#) [Learn how to choose the right IoT Hub tier for your solution](#)

Number of S1 IoT Hub units [?](#) 1 [1](#) This determines your IoT Hub scale capability and can be changed as your need increases.

Device-to-cloud-messages [?](#) Enabled
Message routing [?](#) Enabled
Cloud-to-device commands [?](#) Enabled
IoT Edge [?](#) Enabled
Device management [?](#) Enabled

Advanced Settings

Device-to-cloud partitions [?](#) 4 [4](#)

[Review + create](#) [« Previous: Basics](#) [Automation options](#)

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: The number of messages allowed per unit per day depends on your hub's pricing tier. For example, if you want the IoT hub to support ingress of 700,000 messages, you choose two S1 tier units.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

Advanced / Device-to-cloud partitions: This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most IoT hubs only need four partitions.

5. Click **Review + create** to review your choices. You see something similar to this screen.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale **Review + create**

BASICS

Subscription	Microsoft Azure Internal Consumption
Resource Group	contoso-hub-rgrp
Region	West US
IoT Hub Name	contoso-test-hub

SIZE AND SCALE

Pricing and scale tier	S1
Number of S1 IoT Hub units	1
Messages per day	400,000
Cost per month	25.00 USD

Create « Previous: Size and scale Automation options

6. Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Retrieve connection string for IoT hub

After your hub has been created, retrieve the connection string for the hub. This is used to connect devices and applications to your hub.

1. Click on your hub to see the IoT Hub pane with Settings, and so on. Click **Shared access policies**.
2. In **Shared access policies**, select the **iothubowner** policy.
3. Under **Shared access keys**, copy the **Connection string -- primary key** to be used later.

Home > IoT Hub > ContosoHub - Shared access policies

ContosoHub - Shared access policies

IoT Hub

Overview Activity log Access control (IAM) Tags Events

SETTINGS

- Shared access policies **Selected**
- Pricing and scale
- Operations monitoring
- IP Filter
- Certificates
- Properties
- Locks
- Automation script

Add

IoT Hub uses permissions to grant access to each functionality.

Search for filter items...

POLICY

iothubowner
service
device
registryRead
registryReadWrite

Access policy name: iothubowner

Permissions:

- Registry read
- Registry write
- Service connect
- Device connect

Shared access keys:

Primary key: [Copy](#)

Secondary key: [Copy](#)

Connection string—primary key: [Copy](#)

HostName=ContosoHub.azure-devices.net;SharedAccessKey=...

Connection string—secondary key: [Copy](#)

HostName=ContosoHub.azure-devices.net;SharedAccessKey=...

For more information, see [Access control](#) in the "IoT Hub developer guide."

Register a new device in the IoT hub

In this section, you create a device identity in the identity registry in your IoT hub. A device cannot connect to IoT hub unless it has an entry in the identity registry. For more information, see the "Identity registry" section of the [IoT Hub developer guide](#)

1. In your IoT hub navigation menu, open **IoT Devices**, then click **Add** to register a new device in your IoT hub.

The screenshot shows the 'ContosoHub - IoT devices' blade. On the left, the navigation menu includes sections like Events, SETTINGS (Shared access policies, Pricing and scale, Operations monitoring, IP Filter, Certificates, Properties, Locks, Automation script), EXPLORERS (Query explorer, IoT devices), and AUTOMATIC DEVICE MANAGEMENT (IoT Edge (preview)). The 'IoT devices' item in the EXPLORERS section is highlighted with a red box. At the top right, there's a '+ Add' button (also highlighted with a red box), a Refresh button, and a Delete button. Below the buttons, a message says: 'You can use this tool to view, create, update, and delete devices on your IoT Hub.' There's a 'Query' input field containing a sample SQL query: 'SELECT * FROM devices WHERE optional (e.g. tags.location='US')'. A blue 'Execute' button is below the query field. A table header with columns DEVICE ID, STATUS, LAST ACTIVITY, LAST STATUS..., AUTHENTICATI..., and CLOUD TO DEV... is shown, followed by a message 'No results'.

2. Provide a name for your new device, such as **myDeviceId**, and click **Save**. This action creates a new device identity for your IoT hub.



Create a device

□ X



Learn more about creating devices



* Device ID i

myDeviceId



Authentication type i

Symmetric key

X.509 Self-Signed

X.509 CA Signed

* Primary key i

Enter your primary key

* Secondary key i

Enter your secondary key

Auto-generate keys i



Connect this device to an IoT hub i

Enable

Disable

Parent device (Preview) i

No parent device

[Set a parent device](#)

[Save](#)

IMPORTANT

The device ID may be visible in the logs collected for customer support and troubleshooting, so make sure to avoid any sensitive information while naming it.

3. After the device is created, open the device from the list in the **IoT devices** pane. Copy the **Connection string---primary key** to use later.

The screenshot shows the 'Device details' page for a device named 'myDeviceId'. It includes fields for Device Id, Primary key, Secondary key, and Connection strings. A note at the bottom indicates that the IoT Hub identity registry stores device IDs and keys for secure access, and provides a link to the IoT Hub developer guide.

NOTE

The IoT Hub identity registry only stores device identities to enable secure access to the IoT hub. It stores device IDs and keys to use as security credentials, and an enabled/disabled flag that you can use to disable access for an individual device. If your application needs to store other device-specific metadata, it should use an application-specific store. For more information, see [IoT Hub developer guide](#).

Create the service app

In this section, you create a Python console app that adds location metadata to the device twin associated with your **{Device ID}**. It then queries the device twins stored in the IoT hub selecting the devices located in Redmond, and then the ones that are reporting a cellular connection.

1. Open a command prompt and install the **Azure IoT Hub Service SDK for Python**. Close the command prompt after you install the SDK.

```
pip install azure-iothub-service-client
```

2. Using a text editor, create a new **AddTagsAndQuery.py** file.

3. Add the following code to import the required modules from the service SDK:

```
import sys
import iothub_service_client
from iothub_service_client import IoTHubRegistryManager, IoTHubRegistryManagerAuthMethod
from iothub_service_client import IoTHubDeviceTwin, IoTHubError
```

4. Add the following code, replacing the placeholder for **[IoTHub Connection String]** and **[Device Id]** with the connection string for the IoT hub and the device ID you created in the previous sections.

```
CONNECTION_STRING = "[IoTHub Connection String]"
DEVICE_ID = "[Device Id]"

UPDATE_JSON = "{\"properties\":{\"desired\":{\"location\":\"Redmond\"}}}"

UPDATE_JSON_SEARCH = "\"location\":\"Redmond\""
UPDATE_JSON_CLIENT_SEARCH = "\"connectivity\":\"cellular\""
```

5. Add the following code to the **AddTagsAndQuery.py** file:

```

def iothub_service_sample_run():
    try:
        iothub_registry_manager = IoTHubRegistryManager(CONNECTION_STRING)

        iothub_registry_statistics = iothub_registry_manager.get_statistics()
        print ( "Total device count           : "
{0}".format(iothub_registry_statistics.totalDeviceCount) )
        print ( "Enabled device count         : "
{0}".format(iothub_registry_statistics.enabledDeviceCount) )
        print ( "Disabled device count        : "
{0}".format(iothub_registry_statistics.disabledDeviceCount) )
        print ( "" )

        number_of_devices = iothub_registry_statistics.totalDeviceCount
        dev_list = iothub_registry_manager.get_device_list(number_of_devices)

        iothub_twin_method = IoTHubDeviceTwin(CONNECTION_STRING)

        for device in range(0, number_of_devices):
            if dev_list[device].deviceId == DEVICE_ID:
                twin_info = iothub_twin_method.update_twin(dev_list[device].deviceId, UPDATE_JSON)

        print ( "Devices in Redmond: " )
        for device in range(0, number_of_devices):
            twin_info = iothub_twin_method.get_twin(dev_list[device].deviceId)

            if twin_info.find(UPDATE_JSON_SEARCH) > -1:
                print ( dev_list[device].deviceId )

        print ( "" )

        print ( "Devices in Redmond using cellular network: " )
        for device in range(0, number_of_devices):
            twin_info = iothub_twin_method.get_twin(dev_list[device].deviceId)

            if twin_info.find(UPDATE_JSON_SEARCH) > -1:
                if twin_info.find(UPDATE_JSON_CLIENT_SEARCH) > -1:
                    print ( dev_list[device].deviceId )

        except IoTHubError as iothub_error:
            print ( "Unexpected error {0}".format(iothub_error) )
            return
        except KeyboardInterrupt:
            print ( "IoTHub sample stopped" )

```

The **Registry** object exposes all the methods required to interact with device twins from the service. The code first initializes the **Registry** object, then updates the device twin for **deviceId**, and finally runs two queries. The first selects only the device twins of devices located in the **Redmond43** plant, and the second refines the query to select only the devices that are also connected through cellular network.

- Add the following code at the end of **AddTagsAndQuery.py** to implement the **iothub_service_sample_run** function:

```

if __name__ == '__main__':
    print ( "Starting the IoT Hub Device Twins Python service sample..." )

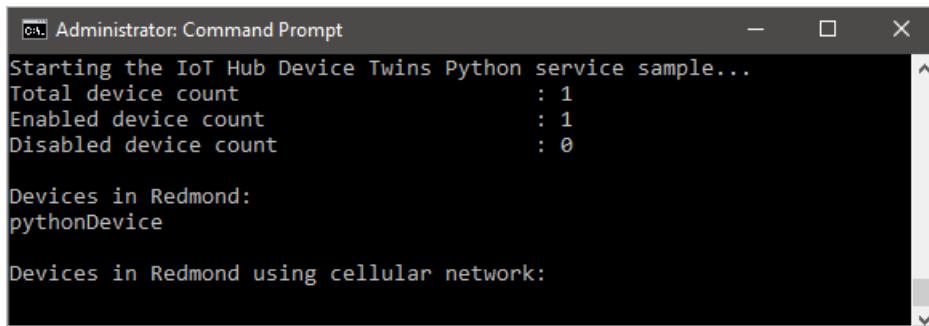
    iothub_service_sample_run()

```

- Run the application with:

```
python AddTagsAndQuery.py
```

You should see one device in the results for the query asking for all devices located in **Redmond43** and none for the query that restricts the results to devices that use a cellular network.



The screenshot shows the output of a command-line application. It starts with "Starting the IoT Hub Device Twins Python service sample...". Then it displays device counts: "Total device count : 1", "Enabled device count : 1", and "Disabled device count : 0". Below this, it lists "Devices in Redmond:" followed by "pythonDevice". At the bottom, it lists "Devices in Redmond using cellular network:".

```
Administrator: Command Prompt
Starting the IoT Hub Device Twins Python service sample...
Total device count : 1
Enabled device count : 1
Disabled device count : 0

Devices in Redmond:
pythonDevice

Devices in Redmond using cellular network:
```

In the next section, you create a device app that reports the connectivity information and changes the result of the query in the previous section.

Create the device app

In this section, you create a Python console app that connects to your hub as your **{Device ID}**, and then updates its device twin's reported properties to contain the information that it is connected using a cellular network.

1. Open a command prompt and install the **Azure IoT Hub Service SDK for Python**. Close the command prompt after you install the SDK.

```
pip install azure-iothub-device-client
```

2. Using a text editor, create a new **ReportConnectivity.py** file.

3. Add the following code to import the required modules from the service SDK:

```
import time
import iothub_client
from iothub_client import IoTHubClient, IoTHubClientError, IoTHubTransportProvider, IoTHubClientResult,
IoTHubError
```

4. Add the following code, replacing the placeholder for **[IoTHub Device Connection String]** with the connection string for the IoT hub device you created in the previous sections.

```
CONNECTION_STRING = "[IoTHub Device Connection String]"

# choose HTTP, AMQP, AMQP_WS or MQTT as transport protocol
PROTOCOL = IoTHubTransportProvider.MQTT

TIMER_COUNT = 5
TWIN_CONTEXT = 0
SEND_REPORTED_STATE_CONTEXT = 0
```

5. Add the following code to the **ReportConnectivity.py** file to implement the device twins functionality:

```

def device_twin_callback(update_state, payload, user_context):
    print ( "" )
    print ( "Twin callback called with:" )
    print ( "    updateStatus: %s" % update_state )
    print ( "    payload: %s" % payload )

def send_reported_state_callback(status_code, user_context):
    print ( "" )
    print ( "Confirmation for reported state called with:" )
    print ( "    status_code: %d" % status_code )

def iothub_client_init():
    client = IoTHubClient(CONNECTION_STRING, PROTOCOL)

    if client.protocol == IoTHubTransportProvider.MQTT or client.protocol ==
    IoTHubTransportProvider.MQTT_WS:
        client.set_device_twin_callback(
            device_twin_callback, TWIN_CONTEXT)

    return client

def iothub_client_sample_run():
    try:
        client = iothub_client_init()

        if client.protocol == IoTHubTransportProvider.MQTT:
            print ( "Sending data as reported property..." )

        reported_state = "{\"connectivity\":\"cellular\"}"

        client.send_reported_state(reported_state, len(reported_state),
send_reported_state_callback, SEND_REPORTED_STATE_CONTEXT)

        while True:
            print ( "Press Ctrl-C to exit" )

            status_counter = 0
            while status_counter <= TIMER_COUNT:
                status = client.get_send_status()
                time.sleep(10)
                status_counter += 1
            except IoTHubError as iothub_error:
                print ( "Unexpected error %s from IoTHub" % iothub_error )
                return
            except KeyboardInterrupt:
                print ( "IoTHubClient sample stopped" )

```

The **Client** object exposes all the methods you require to interact with device twins from the device. The previous code, after it initializes the **Client** object, retrieves the device twin for your device and updates its reported property with the connectivity information.

6. Add the following code at the end of **ReportConnectivity.py** to implement the **iothub_client_sample_run** function:

```

if __name__ == '__main__':
    print ( "Starting the IoT Hub Device Twins Python client sample..." )

    iothub_client_sample_run()

```

7. Run the device app

```
python ReportConnectivity.py
```

You should see confirmation the device twins were updated.

```
Administrator: Command Prompt - python ReportConnectivity.py
Starting the IoT Hub Python client sample...
Info: IoT Hub SDK for C, version 1.1.27
Sending data as reported property...
Press Ctrl-C to exit

Twin callback called with:
  updateStatus: COMPLETE
  payload: {
    "desired": {
      "location": "Redmond",
      "$version": 2
    },
    "reported": {
      "$version": 1
    }
}

Confirmation for reported state called with:
  status_code: 204
```

- Now that the device reported its connectivity information, it should appear in both queries. Go back and run the queries again:

```
python AddTagsAndQuery.py
```

This time your **{Device ID}** should appear in both query results.

```
Administrator: Command Prompt
Starting the IoT Hub Device Twins Python service sample...
Total device count          : 1
Enabled device count        : 1
Disabled device count       : 0

Devices in Redmond:
pythonDevice

Devices in Redmond using cellular network:
pythonDevice
```

Next steps

In this tutorial, you configured a new IoT hub in the Azure portal, and then created a device identity in the IoT hub's identity registry. You added device metadata as tags from a back-end app, and wrote a simulated device app to report device connectivity information in the device twin. You also learned how to query this information using the registry.

Use the following resources to learn how to:

- Send telemetry from devices with the [Get started with IoT Hub](#) tutorial,
- Configure devices using device twin's desired properties with the [Use desired properties to configure devices](#) tutorial,
- Control devices interactively (such as turning on a fan from a user-controlled app), with the [Use direct methods](#) tutorial.

Get started with IoT Hub module identity and module twin using the portal and .NET device

3/6/2019 • 7 minutes to read

NOTE

Module identities and module twins are similar to Azure IoT Hub device identity and device twin, but provide finer granularity. While Azure IoT Hub device identity and device twin enable the back-end application to configure a device and provides visibility on the device's conditions, a module identity and module twin provide these capabilities for individual components of a device. On capable devices with multiple components, such as operating system based devices or firmware devices, it allows for isolated configuration and conditions for each component.

In this tutorial, you will learn:

1. How to create a module identity in the portal.
2. How to use .NET device SDK update the module twin from your device.

NOTE

For information about the Azure IoT SDKs that you can use to build both applications to run on devices, and your solution back end, see [Azure IoT SDKs](#).

To complete this tutorial, you need the following:

- Visual Studio 2015 or Visual Studio 2017.
- An active Azure account. (If you don't have an account, you can create a [free account](#) in just a couple of minutes.)

Create an IoT hub

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose **+Create a resource**, then choose **Internet of Things**.
3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Review + create

Create an IoT Hub to help you connect, monitor, and manage billions of your IoT assets. [Learn More](#)

PROJECT DETAILS

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

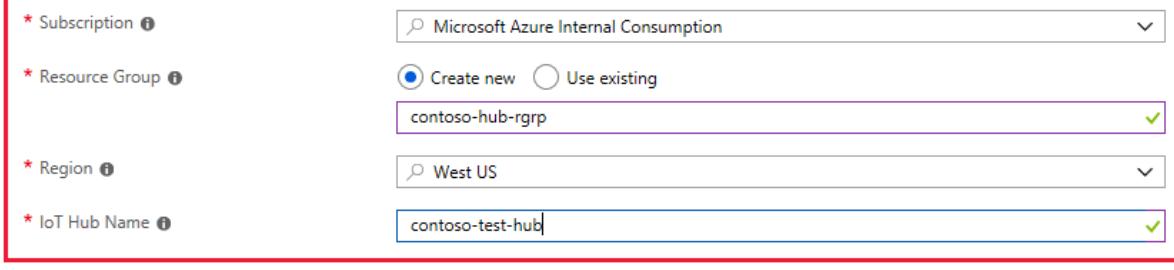
* Subscription [?](#) Microsoft Azure Internal Consumption

* Resource Group [?](#) Create new Use existing
contoso-hub-rgrp

* Region [?](#) West US

* IoT Hub Name [?](#) contoso-test-hub

Review + create **Next: Size and scale »** Automation options



Fill in the fields.

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics **Size and scale** Review + create

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS

* Pricing and scale tier [?](#) S1: Standard tier [▼](#) [Learn how to choose the right IoT Hub tier for your solution](#)

Number of S1 IoT Hub units [?](#) [1](#) This determines your IoT Hub scale capability and can be changed as your need increases.

Device-to-cloud-messages [?](#) Enabled

Message routing [?](#) Enabled

Cloud-to-device commands [?](#) Enabled

IoT Edge [?](#) Enabled

Device management [?](#) Enabled

Advanced Settings

Device-to-cloud partitions [?](#) [4](#)

Review + create [« Previous: Basics](#) Automation options

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: The number of messages allowed per unit per day depends on your hub's pricing tier. For example, if you want the IoT hub to support ingress of 700,000 messages, you choose two S1 tier units.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

Advanced / Device-to-cloud partitions: This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most IoT hubs only need four partitions.

5. Click **Review + create** to review your choices. You see something similar to this screen.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Review + create **Review + create**

BASICS

Subscription ⓘ	Microsoft Azure Internal Consumption
Resource Group ⓘ	contoso-hub-rgrp
Region ⓘ	West US
IoT Hub Name ⓘ	contoso-test-hub

SIZE AND SCALE

Pricing and scale tier ⓘ	S1
Number of S1 IoT Hub units ⓘ	1
Messages per day ⓘ	400,000
Cost per month	25.00 USD

Create « Previous: Size and scale Automation options

6. Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Retrieve connection string for IoT hub

After your hub has been created, retrieve the connection string for the hub. This is used to connect devices and applications to your hub.

1. Click on your hub to see the IoT Hub pane with Settings, and so on. Click **Shared access policies**.
2. In **Shared access policies**, select the **iothubowner** policy.
3. Under **Shared access keys**, copy the **Connection string -- primary key** to be used later.

Home > IoT Hub > ContosoHub - Shared access policies

ContosoHub - Shared access policies

IoT Hub

Search (Ctrl+ /) Add

Overview Activity log Access control (IAM) Tags Events

SETTINGS

Shared access policies Pricing and scale Operations monitoring IP Filter Certificates Properties Locks Automation script

iothubowner ContosoHub

Save Discard Regen key Delete

IoT Hub uses permissions to grant access to each functionality.

Search to filter items..

POLICY

iothubowner service device registryRead registryReadWrite

Access policy name: iothubowner

Permissions:

Registry read ⓘ
 Registry write ⓘ
 Service connect ⓘ
 Device connect ⓘ

Shared access keys

Primary key ⓘ HostName=ContosoHub.azure-devices.net;SharedAccessKey...
Secondary key ⓘ HostName=ContosoHub.azure-devices.net;SharedAccessKey...

Connection string—primary key ⓘ HostName=ContosoHub.azure-devices.net;SharedAccessKey...
Connection string—secondary key ⓘ HostName=ContosoHub.azure-devices.net;SharedAccessKey...

For more information, see [Access control](#) in the "IoT Hub developer guide."

Register a new device in the IoT hub

In this section, you create a device identity in the identity registry in your IoT hub. A device cannot connect to IoT hub unless it has an entry in the identity registry. For more information, see the "Identity registry" section of the [IoT Hub developer guide](#)

1. In your IoT hub navigation menu, open **IoT Devices**, then click **Add** to register a new device in your IoT hub.

The screenshot shows the 'ContosoHub - IoT devices' blade in the Azure portal. The left sidebar contains several sections: 'Events', 'SETTINGS' (with options like Shared access policies, Pricing and scale, Operations monitoring, IP Filter, Certificates, Properties, Locks, and Automation script), 'EXPLORERS' (with Query explorer and IoT devices selected), and 'AUTOMATIC DEVICE MANAGEMENT' (with IoT Edge (preview)). The main area features a red box around the '+ Add' button in the top toolbar. Below it is a query editor with a sample SQL query: 'SELECT * FROM devices WHERE optional (e.g. tags.location='US')'. At the bottom, a table header is shown with columns: DEVICE ID, STATUS, LAST ACTIVITY, LAST STATUS..., AUTHENTICATI..., and CLOUD TO DEV... . The message 'No results' is displayed below the table.

2. Provide a name for your new device, such as **myDeviceId**, and click **Save**. This action creates a new device identity for your IoT hub.

Create a device

Learn more about creating devices

* Device ID [i](#)
myDeviceId

Authentication type [i](#)

Symmetric key X.509 Self-Signed X.509 CA Signed

* Primary key [i](#)
Enter your primary key

* Secondary key [i](#)
Enter your secondary key

Auto-generate keys [i](#)

Connect this device to an IoT hub [i](#)

Enable Disable

Parent device (Preview) [i](#)
No parent device
Set a parent device

Save

IMPORTANT

The device ID may be visible in the logs collected for customer support and troubleshooting, so make sure to avoid any sensitive information while naming it.

3. After the device is created, open the device from the list in the **IoT devices** pane. Copy the **Connection string---primary key** to use later.

The screenshot shows the 'Device details' page for a device named 'myDeviceId'. It displays the primary key ('<Primary Key>') and secondary key ('<Secondary Key>'). Below these are two connection strings: 'Connection string (primary key)' (HostName=ContosoHub.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=<Primary Key>) and 'Connection string (secondary key)' (HostName=ContosoHub.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=<Secondary Key>). A red box highlights the primary key and its corresponding connection string. At the bottom, there are buttons for 'Enable' and 'Disable' the device, and a section for setting a parent device.

NOTE

The IoT Hub identity registry only stores device identities to enable secure access to the IoT hub. It stores device IDs and keys to use as security credentials, and an enabled/disabled flag that you can use to disable access for an individual device. If your application needs to store other device-specific metadata, it should use an application-specific store. For more information, see [IoT Hub developer guide](#).

Create a module identity in the portal

Within one device identity, you can create up to 20 module identities. Click the **Add Module Identity** button on top to create your first module identity called **myFirstModule**.

The screenshot shows the 'Device Details' page for a device named 'MyFirstDevice'. On the right, a modal window titled 'Add Module Identity' is open. It prompts for a 'Module Identity Name' (set to 'myFirstModule'), 'Authentication Type' (set to 'Symmetric Key'), and 'Primary Key' (with a placeholder 'Enter your primary key here'). There are also fields for 'Secondary Key' (placeholder 'Enter your secondary key here') and a checked 'Auto Generate Keys' option. At the bottom of the modal is a 'Save' button.

Save and click the just created module identity. You can see the module identity details. Save the connect string - primary key. It will be used in the next section where you set up your module on the device.

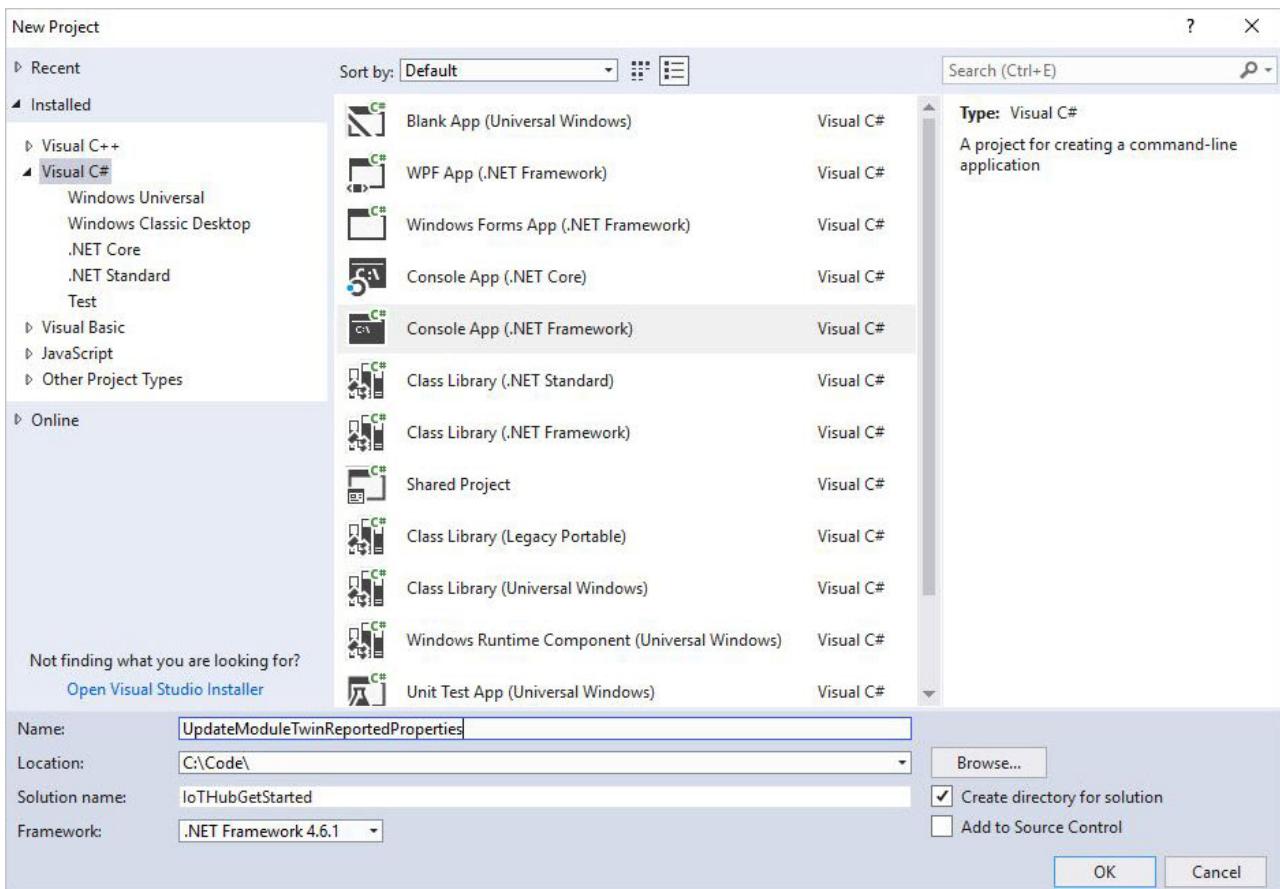
The screenshot shows the 'Module Identity Details' page in the Azure portal. At the top, it says 'MyFirstDevice/myFirstModule'. Below that, there are sections for 'Module Identity Name' (set to 'MyFirstDevice/myFirstModule'), 'Primary key' (a long black redacted string), 'Secondary key' (another long black redacted string), 'Connection string—primary key' (a long black redacted string), and 'Connection string—secondary key' (a long black redacted string). Each field has a blue 'edit' icon to its right.

Update the module twin using .NET device SDK

You've successfully created the module identity in your IoT Hub. Let's try to communicate to the cloud from your simulated device. Once a module identity is created, a module twin is implicitly created in IoT Hub. In this section, you will create a .NET console app on your simulated device that updates the module twin reported properties.

Create a Visual Studio project

In Visual Studio, add a Visual C# Windows Classic Desktop project to the existing solution by using the **Console App (.NET Framework)** project template. Make sure the .NET Framework version is 4.6.1 or later. Name the project **UpdateModuleTwinReportedProperties**.



Install the latest Azure IoT Hub .NET device SDK

Module identity and module twin is in public preview. It's only available in the IoT Hub prerelease device SDKs. In Visual Studio, open tools > Nuget package manager > manage Nuget packages for solution. Search Microsoft.Azure.Devices.Client. Make sure you've checked include prerelease check box. Select the latest version and install. Now you have access to all the module features.



Get your module connection string

Login to [Azure portal](#). Navigate to your IoT Hub and click IoT Devices. Find myFirstDevice, open it and you see myFirstModule was successfully created. Copy the module connection string. It is needed in the next step.

Create UpdateModuleTwinReportedProperties console app

Add the following `using` statements at the top of the **Program.cs** file:

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

Add the following fields to the **Program** class. Replace the placeholder value with the module connection string.

```
private const string ModuleConnectionString = "<Your module connection string>";
private static ModuleClient Client = null;
```

Add the following method **OnDesiredPropertyChanged** to the **Program** class:

```
private static async Task OnDesiredPropertyChanged(TwinCollection desiredProperties, object userContext)
{
    Console.WriteLine("desired property change:");
    Console.WriteLine(JsonConvert.SerializeObject(desiredProperties));
    Console.WriteLine("Sending current time as reported property");
    TwinCollection reportedProperties = new TwinCollection
    {
        ["DateTimeLastDesiredPropertyChangeReceived"] = DateTime.Now
    };

    await Client.UpdateReportedPropertiesAsync(reportedProperties).ConfigureAwait(false);
}
```

Finally, add the following lines to the **Main** method:

```
static void Main(string[] args)
{
    Microsoft.Azure.Devices.Client.TransportType transport = Microsoft.Azure.Devices.Client.TransportType.Amqp;

    try
    {
        Client = ModuleClient.CreateFromConnectionString(ModuleConnectionString, transport);
        Client.SetConnectionStatusChangesHandler(ConnectionStatusChangeHandler);
        Client.SetDesiredPropertyUpdateCallbackAsync(OnDesiredPropertyChanged, null).Wait();

        Console.WriteLine("Retrieving twin");
        var twinTask = Client.GetTwinAsync();
        twinTask.Wait();
        var twin = twinTask.Result;
        Console.WriteLine(JsonConvert.SerializeObject(twin));

        Console.WriteLine("Sending app start time as reported property");
        TwinCollection reportedProperties = new TwinCollection();
        reportedProperties["DateTimeLastAppLaunch"] = DateTime.Now;

        Client.UpdateReportedPropertiesAsync(reportedProperties);
    }
    catch (AggregateException ex)
    {
        Console.WriteLine("Error in sample: {0}", ex);
    }

    Console.WriteLine("Waiting for Events. Press enter to exit...");
    Console.ReadKey();
    Client.CloseAsync().Wait();
}

private static void ConnectionStatusChangeHandler(ConnectionStatus status, ConnectionStatusChangeReason reason)
{
    Console.WriteLine($"Status {status} changed: {reason}");
}
```

This code sample shows you how to retrieve the module twin and update reported properties with AMQP protocol. In public preview, we only support AMQP for module twin operations.

Run the apps

You are now ready to run the apps. In Visual Studio, in Solution Explorer, right-click your solution, and then click **Set StartUp projects**. Select **Multiple startup projects**, and then select **Start** as the action for the console app. And then press F5 to start both apps running.

Next steps

To continue getting started with IoT Hub and to explore other IoT scenarios, see:

- [Get started with IoT Hub module identity and module twin using .NET backup and .NET device](#)
- [Getting started with IoT Edge](#)

Get started with IoT Hub module identity and module twin using .NET back end and .NET device

1/22/2019 • 9 minutes to read

NOTE

Module identities and module twins are similar to Azure IoT Hub device identity and device twin, but provide finer granularity. While Azure IoT Hub device identity and device twin enable the back-end application to configure a device and provides visibility on the device's conditions, a module identity and module twin provide these capabilities for individual components of a device. On capable devices with multiple components, such as operating system based devices or firmware devices, it allows for isolated configuration and conditions for each component.

At the end of this tutorial, you have two .NET console apps:

- **CreateIdentities**, which creates a device identity, a module identity and associated security key to connect your device and module clients.
- **UpdateModuleTwinReportedProperties**, which sends updated module twin reported properties to your IoT Hub.

NOTE

For information about the Azure IoT SDKs that you can use to build both applications to run on devices, and your solution back end, see [Azure IoT SDKs](#).

To complete this tutorial, you need the following:

- Visual Studio 2017.
- An active Azure account. (If you don't have an account, you can create a [free account](#) in just a couple of minutes.)

Create an IoT hub

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose +**Create a resource**, then choose **Internet of Things**.
3. Click **Iot Hub** from the list on the right. You see the first screen for creating an IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Review + create

Create an IoT Hub to help you connect, monitor, and manage billions of your IoT assets. [Learn More](#)

PROJECT DETAILS

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

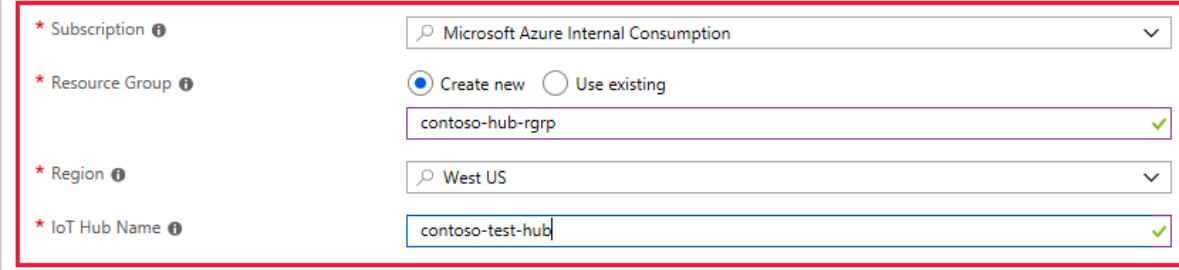
* Subscription

* Resource Group Create new Use existing

* Region

* IoT Hub Name

[Review + create](#) [Next: Size and scale »](#) [Automation options](#)



Fill in the fields.

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics **Size and scale** Review + create

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS

* Pricing and scale tier [?](#) S1: Standard tier [▼](#) [Learn how to choose the right IoT Hub tier for your solution](#)

Number of S1 IoT Hub units [?](#) [1](#) This determines your IoT Hub scale capability and can be changed as your need increases.

Device-to-cloud-messages [?](#) Enabled

Message routing [?](#) Enabled

Cloud-to-device commands [?](#) Enabled

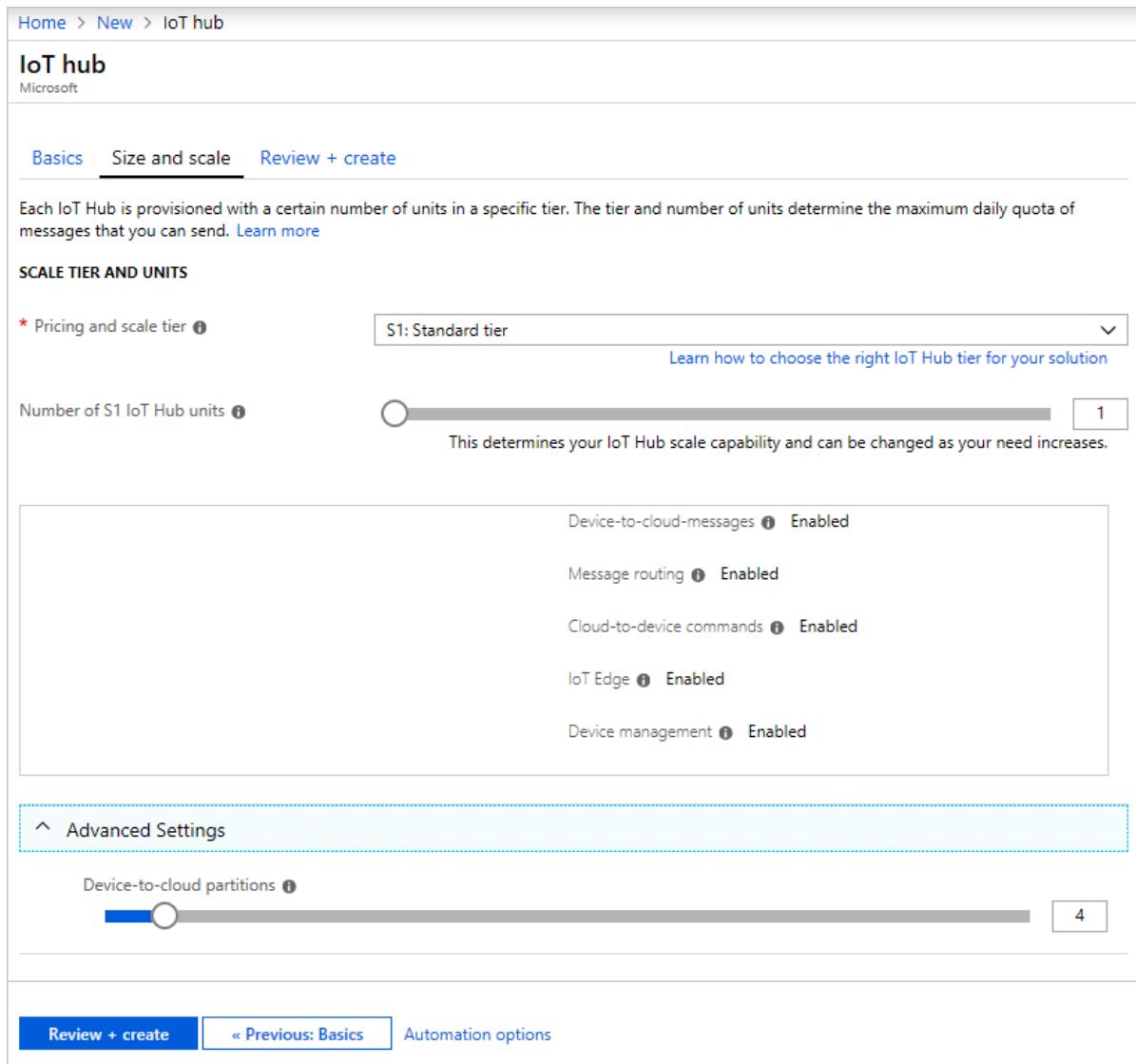
IoT Edge [?](#) Enabled

Device management [?](#) Enabled

Advanced Settings

Device-to-cloud partitions [?](#) [4](#)

Review + create [« Previous: Basics](#) Automation options



On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: The number of messages allowed per unit per day depends on your hub's pricing tier. For example, if you want the IoT hub to support ingress of 700,000 messages, you choose two S1 tier units.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

Advanced / Device-to-cloud partitions: This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most IoT hubs only need four partitions.

5. Click **Review + create** to review your choices. You see something similar to this screen.

IoT hub
Microsoft

Basics **Size and scale** **Review + create**

BASICS

Subscription	Microsoft Azure Internal Consumption
Resource Group	contoso-hub-rgrp
Region	West US
IoT Hub Name	contoso-test-hub

SIZE AND SCALE

Pricing and scale tier	S1
Number of S1 IoT Hub units	1
Messages per day	400,000
Cost per month	25.00 USD

Create [« Previous: Size and scale](#) [Automation options](#)

- Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Retrieve connection string for IoT hub

After your hub has been created, retrieve the connection string for the hub. This is used to connect devices and applications to your hub.

- Click on your hub to see the IoT Hub pane with Settings, and so on. Click **Shared access policies**.
- In **Shared access policies**, select the **iothubowner** policy.
- Under **Shared access keys**, copy the **Connection string -- primary key** to be used later.

ContosoHub - Shared access policies

iothubowner

Access policy name: iothubowner

Permissions: Registry read, Registry write, Service connect, Device connect

Shared access keys:

Primary key	HostName=ContosoHub.azure-devices.net;SharedAccessKey...
Secondary key	HostName=ContosoHub.azure-devices.net;SharedAccessKey...

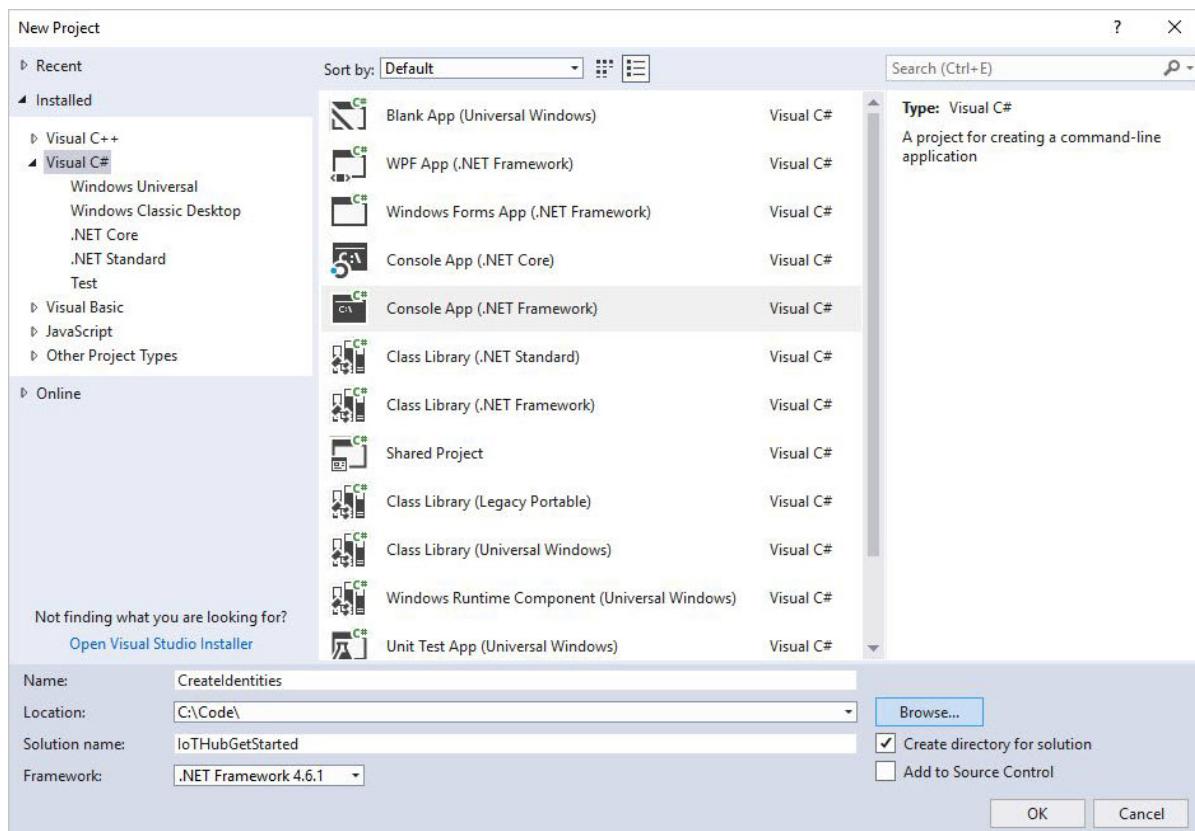
For more information, see [Access control](#) in the "IoT Hub developer guide."

You have now created your IoT hub, and you have the host name and IoT Hub connection string that you need to complete the rest of this tutorial.

Create a module identity

In this section, you create a .NET console app that creates a device identity and a module identity in the identity registry in your IoT hub. A device or module cannot connect to IoT hub unless it has an entry in the identity registry. For more information, see the [Identity Registry section of the IoT Hub developer guide](#). When you run this console app, it generates a unique ID and key for both device and module. Your device and module use these values to identify itself when it sends device-to-cloud messages to IoT Hub. The IDs are case-sensitive.

- 1. Create a Visual Studio project** - In Visual Studio, add a Visual C# Windows Classic Desktop project to a new solution by using the **Console App (.NET Framework)** project template. Make sure the .NET Framework version is 4.6.1 or later. Name the project **CreateIdentities** and name the solution **IoTHubGetStarted**.



- 2. Install Azure IoT Hub .NET service SDK V1.16.0-preview-001** - Module identity and module twin is in public preview. It's only available in the IoT Hub prerelease service SDKs. In Visual Studio, open tools > Nuget package manager > manage Nuget packages for solution. Search Microsoft.Azure.Devices. Make sure you've checked include prerelease check box. Select version 1.16.0-preview-001 and install. Now you have access to all the module features.



3. Add the following `using` statements at the top of the **Program.cs** file:

```
using Microsoft.Azure.Devices;
using Microsoft.Azure.Devices.Common.Exceptions;
```

4. Add the following fields to the **Program** class. Replace the placeholder value with the IoT Hub connection string for the hub that you created in the previous section.

```
const string connectionString = "<replace_with_iothub_connection_string>";
const string deviceID = "myFirstDevice";
const string moduleID = "myFirstModule";
```

5. Add the following code to the **Main** class.

```
static void Main(string[] args)
{
    AddDeviceAsync().Wait();
    AddModuleAsync().Wait();
}
```

6. Add the following methods to the **Program** class:

```
private static async Task AddDeviceAsync()
{
    RegistryManager registryManager =
        RegistryManager.CreateFromConnectionString(connectionString);
    Device device;

    try
    {
        device = await registryManager.AddDeviceAsync(new Device(deviceID));
    }
    catch (DeviceAlreadyExistsException)
    {
        device = await registryManager.GetDeviceAsync(deviceID);
    }

    Console.WriteLine("Generated device key: {0}",
        device.Authentication.SymmetricKey.PrimaryKey);
}

private static async Task AddModuleAsync()
{
    RegistryManager registryManager =
        RegistryManager.CreateFromConnectionString(connectionString);
    Module module;

    try
    {
        module =
            await registryManager.AddModuleAsync(new Module(deviceID, moduleID));
    }
    catch (ModuleAlreadyExistsException)
    {
        module = await registryManager.GetModuleAsync(deviceID, moduleID);
    }

    Console.WriteLine("Generated module key: {0}", module.Authentication.SymmetricKey.PrimaryKey);
}
```

The `AddDeviceAsync()` method creates a device identity with ID **myFirstDevice**. (If that device ID already

exists in the identity registry, the code simply retrieves the existing device information.) The app then displays the primary key for that identity. You use this key in the simulated device app to connect to your IoT hub.

The AddModuleAsync() method creates a module identity with ID **myFirstModule** under device **myFirstDevice**. (If that module ID already exists in the identity registry, the code simply retrieves the existing module information.) The app then displays the primary key for that identity. You use this key in the simulated module app to connect to your IoT hub.

IMPORTANT

The device ID may be visible in the logs collected for customer support and troubleshooting, so make sure to avoid any sensitive information while naming it.

7. Run this application, and make a note of the device key and module key.

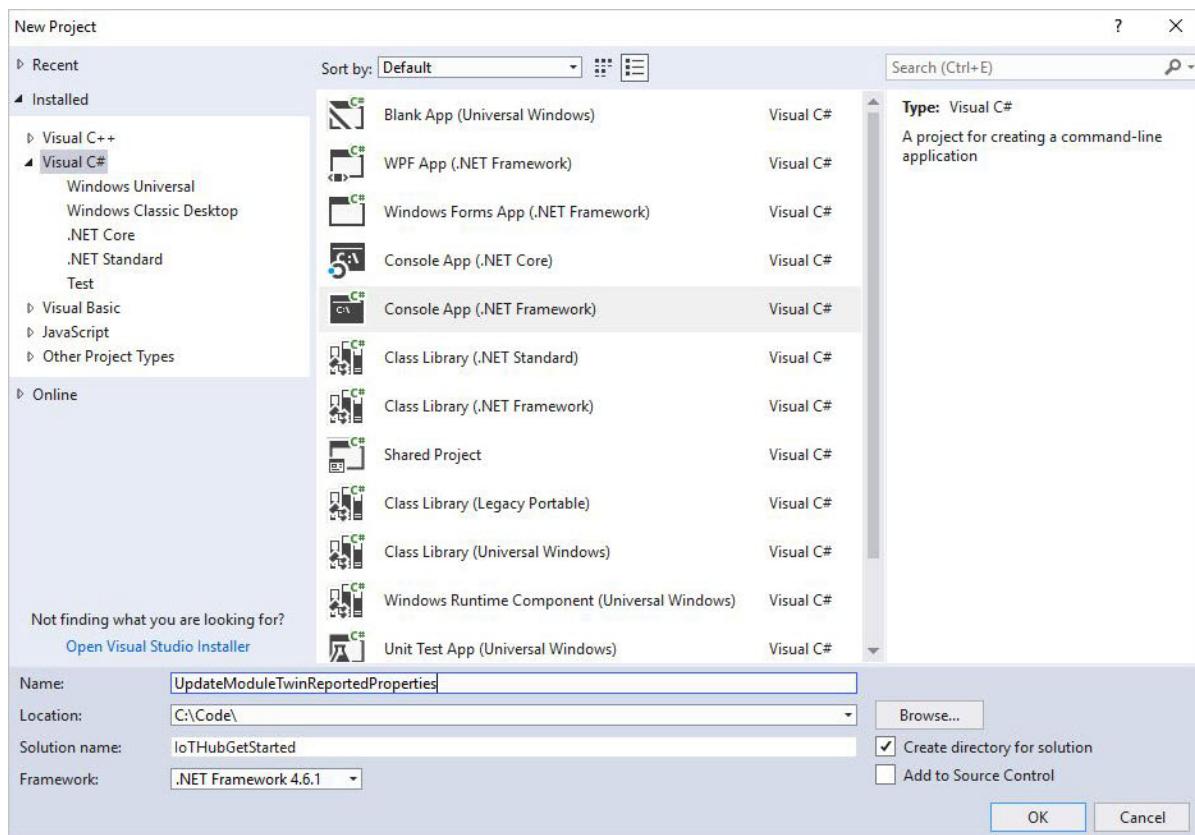
NOTE

The IoT Hub identity registry only stores device and module identities to enable secure access to the IoT hub. The identity registry stores device IDs and keys to use as security credentials. The identity registry also stores an enabled/disabled flag for each device that you can use to disable access for that device. If your application needs to store other device-specific metadata, it should use an application-specific store. There is no enabled/disabled flag for module identities. For more information, see [IoT Hub developer guide](#).

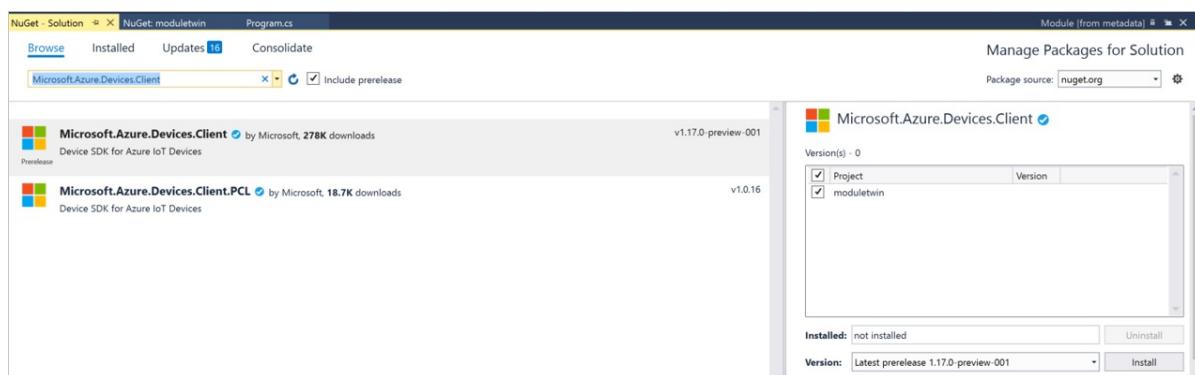
Update the module twin using .NET device SDK

In this section, you create a .NET console app on your simulated device that updates the module twin reported properties.

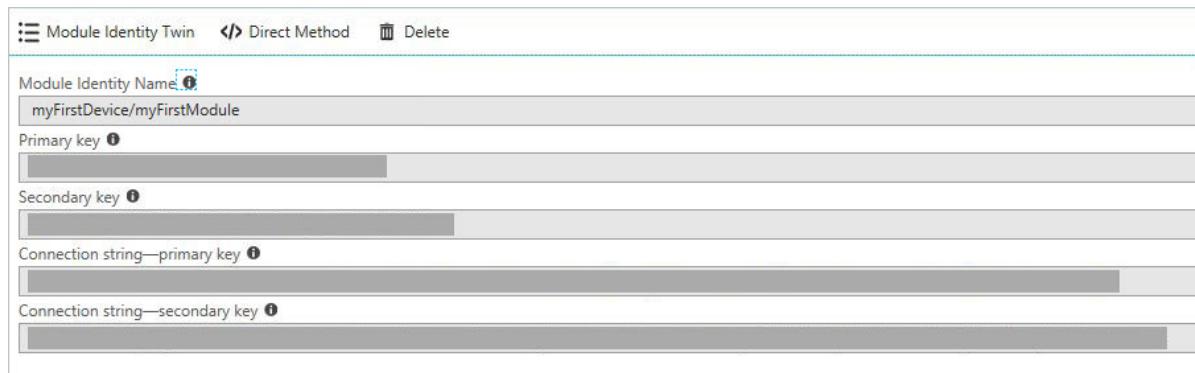
1. **Create a Visual Studio project:** In Visual Studio, add a Visual C# Windows Classic Desktop project to the existing solution by using the **Console App (.NET Framework)** project template. Make sure the .NET Framework version is 4.6.1 or later. Name the project **UpdateModuleTwinReportedProperties**.



2. **Install the latest Azure IoT Hub .NET device SDK:** Module identity and module twin is in public preview. It's only available in the IoT Hub prerelease device SDKs. In Visual Studio, open tools > Nuget package manager > manage Nuget packages for solution. Search Microsoft.Azure.Devices.Client. Make sure you've checked include prerelease check box. Select the latest version and install. Now you have access to all the module features.



3. **Get your module connection string** -- now if you login to [Azure portal](#). Navigate to your IoT Hub and click IoT Devices. Find myFirstDevice, open it and you see myFirstModule was successfully created. Copy the module connection string. It is needed in the next step.



4. **Create UpdateModuleTwinReportedProperties console app**

Add the following `using` statements at the top of the **Program.cs** file:

```
using Microsoft.Azure.Devices.Client;
using Microsoft.Azure.Devices.Shared;
using System.Threading.Tasks;
using Newtonsoft.Json;
```

Add the following fields to the **Program** class. Replace the placeholder value with the module connection string.

```
private const string ModuleConnectionString =
    "<Your module connection string>";
private static ModuleClient Client = null;
static void ConnectionStatusChangeHandler(ConnectionStatus status,
    ConnectionStatusChangeReason reason)
{
    Console.WriteLine("Connection Status Changed to {0}; the reason is {1}",
        status, reason);
}
```

Add the following method **OnDesiredPropertyChanged** to the **Program** class:

```
private static async Task OnDesiredPropertyChanged(TwinCollection desiredProperties,
    object userContext)
{
    Console.WriteLine("desired property change:");
    Console.WriteLine(JsonConvert.SerializeObject(desiredProperties));
    Console.WriteLine("Sending current time as reported property");
    TwinCollection reportedProperties = new TwinCollection
    {
        ["DateTimeLastDesiredPropertyChangeReceived"] = DateTime.Now
    };

    await Client.UpdateReportedPropertiesAsync(reportedProperties).ConfigureAwait(false);
}
```

Finally, add the following lines to the **Main** method:

```

static void Main(string[] args)
{
    Microsoft.Azure.Devices.Client.TransportType transport =
        Microsoft.Azure.Devices.Client.TransportType.Amqp;

    try
    {
        Client =
            ModuleClient.CreateFromConnectionString(ModuleConnectionString, transport);
        Client.SetConnectionStatusChangesHandler(ConnectionStatusChangeHandler);
        Client.SetDesiredPropertyUpdateCallbackAsync(OnDesiredPropertyChanged, null).Wait();

        Console.WriteLine("Retrieving twin");
        var twinTask = Client.GetTwinAsync();
        twinTask.Wait();
        var twin = twinTask.Result;
        Console.WriteLine(JsonConvert.SerializeObject(twin.Properties));

        Console.WriteLine("Sending app start time as reported property");
        TwinCollection reportedProperties = new TwinCollection();
        reportedProperties["DateTimeLastAppLaunch"] = DateTime.Now;

        Client.UpdateReportedPropertiesAsync(reportedProperties);
    }
    catch (AggregateException ex)
    {
        Console.WriteLine("Error in sample: {0}", ex);
    }

    Console.WriteLine("Waiting for Events. Press enter to exit...");
    Console.ReadLine();
    Client.CloseAsync().Wait();
}

```

This code sample shows you how to retrieve the module twin and update reported properties with AMQP protocol. In public preview, we only support AMQP for module twin operations.

5. In addition to the above **Main** method, you can add below code block to send event to IoT Hub from your module:

```

Byte[] bytes = new Byte[2];
bytes[0] = 0;
bytes[1] = 1;
var sendEventsTask = Client.SendEventAsync(new Message(bytes));
sendEventsTask.Wait();
Console.WriteLine("Event sent to IoT Hub.");

```

Run the apps

You are now ready to run the apps. In Visual Studio, in Solution Explorer, right-click your solution, and then click **Set StartUp projects**. Select **Multiple startup projects**, and then select **Start** as the action for the console app. And then press F5 to start the app.

Next steps

To continue getting started with IoT Hub and to explore other IoT scenarios, see:

- [Getting started with device management](#)
- [Getting started with IoT Edge](#)

Get started with IoT Hub module identity and module twin using Python back end and Python device

3/6/2019 • 4 minutes to read

NOTE

Module identities and module twins are similar to Azure IoT Hub device identity and device twin, but provide finer granularity. While Azure IoT Hub device identity and device twin enable the back-end application to configure a device and provides visibility on the device's conditions, a module identity and module twin provide these capabilities for individual components of a device. On capable devices with multiple components, such as operating system based devices or firmware devices, it allows for isolated configuration and conditions for each component.

At the end of this tutorial, you have two Python apps:

- **CreateIdentities**, which creates a device identity, a module identity and associated security key to connect your device and module clients.
- **UpdateModuleTwinReportedProperties**, which sends updated module twin reported properties to your IoT Hub.

NOTE

For information about the Azure IoT SDKs that you can use to build both applications to run on devices, and your solution back end, see [Azure IoT SDKs](#).

To complete this tutorial, you need the following:

- An active Azure account. (If you don't have an account, you can create a [free account](#) in just a couple of minutes.)
- An IoT Hub.
- Install the latest [Python SDK](#).

You have now created your IoT hub, and you have the host name and IoT Hub connection string that you need to complete the rest of this tutorial.

Create a device identity and a module identity in IoT Hub

In this section, you create a Python app that creates a device identity and a module identity in the identity registry in your IoT hub. A device or module cannot connect to IoT hub unless it has an entry in the identity registry. For more information, see the "Identity registry" section of the [IoT Hub developer guide](#). When you run this console app, it generates a unique ID and key for both device and module. Your device and module use these values to identify itself when it sends device-to-cloud messages to IoT Hub. The IDs are case-sensitive.

Add the following code to your Python file:

```

import sys
import iothub_service_client
from iothub_service_client import IoTHubRegistryManager, IoTHubRegistryManagerAuthMethod, IoTHubError

CONNECTION_STRING = "YourConnString"
DEVICE_ID = "myFirstDevice"
MODULE_ID = "myFirstModule"

try:
    # RegistryManager
    iothub_registry_manager = IoTHubRegistryManager(CONNECTION_STRING)

    # CreateDevice
    primary_key = ""
    secondary_key = ""
    auth_method = IoTHubRegistryManagerAuthMethod.SHARED_PRIVATE_KEY
    new_device = iothub_registry_manager.create_device(DEVICE_ID, primary_key, secondary_key, auth_method)
    print("new_device <" + DEVICE_ID + "> has primary key = " + new_device.primaryKey)

    # CreateModule
    new_module = iothub_registry_manager.create_module(DEVICE_ID, primary_key, secondary_key, MODULE_ID,
auth_method)
    print("device/new_module <" + DEVICE_ID + "/" + MODULE_ID + "> has primary key = " + new_module.primaryKey)

except IoTHubError as iothub_error:
    print ( "Unexpected error {0}".format(iothub_error) )
except KeyboardInterrupt:
    print ( "IoTHubRegistryManager sample stopped" )

```

This app creates a device identity with ID **myFirstDevice** and a module identity with ID **myFirstModule** under device **myFirstDevice**. (If that module ID already exists in the identity registry, the code simply retrieves the existing module information.) The app then displays the primary key for that identity. You use this key in the simulated module app to connect to your IoT hub.

NOTE

The IoT Hub identity registry only stores device and module identities to enable secure access to the IoT hub. The identity registry stores device IDs and keys to use as security credentials. The identity registry also stores an enabled/disabled flag for each device that you can use to disable access for that device. If your application needs to store other device-specific metadata, it should use an application-specific store. There is no enabled/disabled flag for module identities. For more information, see [IoT Hub developer guide](#).

Update the module twin using Python device SDK

In this section, you create a Python app on your simulated device that updates the module twin reported properties.

1. **Get your module connection string** -- now if you login to [Azure portal](#). Navigate to your IoT Hub and click IoT Devices. Find myFirstDevice, open it and you see myFirstModule was successfully created. Copy the module connection string. It is needed in the next step.

2. **Create UpdateModuleTwinReportedProperties app** Add the following `using` statements at the top of the `Program.cs` file:

```

import sys
import iothub_service_client
from iothub_service_client import IoTHubRegistryManager, IoTHubRegistryManagerAuthMethod,
IoTHubDeviceTwin, IoTHubError

CONNECTION_STRING = "FILL IN CONNECTION STRING"
DEVICE_ID = "MyFirstDevice"
MODULE_ID = "MyFirstModule"

UPDATE_JSON = "{\"properties\":{\"desired\":{\"telemetryInterval\":122}}}"

try:
    iothub_twin = IoTHubDeviceTwin(CONNECTION_STRING)

    twin_info = iothub_twin.get_twin(DEVICE_ID, MODULE_ID)
    print ( "" )
    print ( "Twin before update      :" )
    print ( "{0}".format(twin_info) )

    twin_info_updated = iothub_twin.update_twin(DEVICE_ID, MODULE_ID, UPDATE_JSON)
    print ( "" )
    print ( "Twin after update      :" )
    print ( "{0}".format(twin_info_updated) )

except IoTHubError as iothub_error:
    print ( "Unexpected error {0}".format(iothub_error) )
except KeyboardInterrupt:
    print ( "IoTHubRegistryManager sample stopped" )

```

This code sample shows you how to retrieve the module twin and update reported properties with AMQP protocol.

Get updates on the device side

In addition to the above code, you can add below code block to get the twin update message on your device.

```
import random
import time
import sys
import iothub_client
from iothub_client import IoTHubModuleClient, IoTHubClientError, IoTHubTransportProvider, IoTHubClientResult

PROTOCOL = IoTHubTransportProvider.AMQP
CONNECTION_STRING = ""

def module_twin_callback(update_state, payload, user_context):
    print ("")
    print ("Twin callback called with:")
    print ("updateStatus: %s" % update_state )
    print ("context: %s" % user_context )
    print ("payload: %s" % payload )

try:
    module_client = IoTHubModuleClient(CONNECTION_STRING, PROTOCOL)
    module_client.set_module_twin_callback(module_twin_callback, 1234)

    print ("Waiting for incoming twin messages. Hit Control-C to exit.")
    while True:

        time.sleep(1000000)

except IoTHubError as iothub_error:
    print ( "Unexpected error {0}".format(iothub_error) )
except KeyboardInterrupt:
    print ( "module client sample stopped" )
```

Next steps

To continue getting started with IoT Hub and to explore other IoT scenarios, see:

- [Getting started with device management](#)
- [Getting started with IoT Edge](#)

Get started with IoT Hub module identity and module twin using C backend and C device

1/22/2019 • 6 minutes to read

NOTE

Module identities and module twins are similar to Azure IoT Hub device identity and device twin, but provide finer granularity. While Azure IoT Hub device identity and device twin enable the back-end application to configure a device and provides visibility on the device's conditions, a module identity and module twin provide these capabilities for individual components of a device. On capable devices with multiple components, such as operating system based devices or firmware devices, it allows for isolated configuration and conditions for each component.

At the end of this tutorial, you have two C apps:

- **CreateIdentities**, which creates a device identity, a module identity and associated security key to connect your device and module clients.
- **UpdateModuleTwinReportedProperties**, which sends updated module twin reported properties to your IoT Hub.

NOTE

For information about the Azure IoT SDKs that you can use to build both applications to run on devices, and your solution backend, see [Azure IoT SDKs](#).

To complete this tutorial, you need the following:

- An active Azure account. (If you don't have an account, you can create an [Azure free account](#) in just a couple of minutes.)
- An IoT Hub.
- The latest [Azure IoT C SDK](#).

You have now created your IoT hub, and you have the host name and IoT Hub connection string that you need to complete the rest of this tutorial.

Create a device identity and a module identity in IoT Hub

In this section, you create a C app that creates a device identity and a module identity in the identity registry in your IoT hub. A device or module cannot connect to IoT hub unless it has an entry in the identity registry. For more information, see the **Identity registry** section of the [IoT Hub developer guide](#). When you run this console app, it generates a unique ID and key for both device and module. Your device and module use these values to identify itself when it sends device-to-cloud messages to IoT Hub. The IDs are case-sensitive.

Add the following code to your C file:

```
#include <stdio.h>
#include <stdlib.h>

#include "azure_c_shared_utility/crt_abstractions.h"
#include "azure_c_shared_utility/threadapi.h"
#include "azure_c_shared_utility/platform.h"
```

```

#include "iothub_service_client_auth.h"
#include "iothub_registrymanager.h"

static const char* hubConnectionString = "[your hub's connection string]"; // modify

static void createDevice(IOTHUB_REGISTRYMANAGER_HANDLE
    iotHubRegistryManagerHandle, const char* deviceId)
{
    IOTHUB_REGISTRY_DEVICE_CREATE_EX deviceCreateInfo;
    IOTHUB_REGISTRYMANAGER_RESULT result;

    (void)memset(&deviceCreateInfo, 0, sizeof(deviceCreateInfo));
    deviceCreateInfo.version = 1;
    deviceCreateInfo.deviceId = deviceId;
    deviceCreateInfo.primaryKey = "";
    deviceCreateInfo.secondaryKey = "";
    deviceCreateInfo.authMethod = IOTHUB_REGISTRYMANAGER_AUTH_SPK;

    IOTHUB_DEVICE_EX deviceInfoEx;
    memset(&deviceInfoEx, 0, sizeof(deviceInfoEx));
    deviceInfoEx.version = 1;

    // Create device
    result = IoTHubRegistryManager_CreateDevice_Ex(iotHubRegistryManagerHandle,
        &deviceCreateInfo, &deviceInfoEx);
    if (result == IOTHUB_REGISTRYMANAGER_OK)
    {
        (void)printf("IoTHubRegistryManager_CreateDevice: Device has been created successfully: deviceId=%s,
primaryKey=%s\n", deviceInfoEx.deviceId, deviceInfoEx.primaryKey);
    }
    else if (result == IOTHUB_REGISTRYMANAGER_DEVICE_EXIST)
    {
        (void)printf("IoTHubRegistryManager_CreateDevice: Device already exists\n");
    }
    else if (result == IOTHUB_REGISTRYMANAGER_ERROR)
    {
        (void)printf("IoTHubRegistryManager_CreateDevice failed\n");
    }
    // You will need to Free the returned device information after it was created
    IoTHubRegistryManager_FreeDeviceExMembers(&deviceInfoEx);
}

static void createModule(IOTHUB_REGISTRYMANAGER_HANDLE iotHubRegistryManagerHandle, const char* deviceId, const
char* moduleId)
{
    IOTHUB_REGISTRY_MODULE_CREATE moduleCreateInfo;
    IOTHUB_REGISTRYMANAGER_RESULT result;

    (void)memset(&moduleCreateInfo, 0, sizeof(moduleCreateInfo));
    moduleCreateInfo.version = 1;
    moduleCreateInfo.deviceId = deviceId;
    moduleCreateInfo.moduleId = moduleId;
    moduleCreateInfo.primaryKey = "";
    moduleCreateInfo.secondaryKey = "";
    moduleCreateInfo.authMethod = IOTHUB_REGISTRYMANAGER_AUTH_SPK;

    IOTHUB_MODULE moduleInfo;
    memset(&moduleInfo, 0, sizeof(moduleInfo));
    moduleInfo.version = 1;

    // Create module
    result = IoTHubRegistryManager_CreateModule(iotHubRegistryManagerHandle, &moduleCreateInfo, &moduleInfo);
    if (result == IOTHUB_REGISTRYMANAGER_OK)
    {
        (void)printf("IoTHubRegistryManager_CreateModule: Module has been created successfully: deviceId=%s,
moduleId=%s, primaryKey=%s\n", moduleInfo.deviceId, moduleInfo.moduleId, moduleInfo.primaryKey);
    }
    else if (result == IOTHUB_REGISTRYMANAGER_ERROR)
    {
        (void)printf("IoTHubRegistryManager_CreateModule failed\n");
    }
}

```

```

else if (result == IOTHUB_REGISTRYMANAGER_DEVICE_EXISTS)
{
    (void)printf("IoTHubRegistryManager_CreateModule: Module already exists\n");
}
else if (result == IOTHUB_REGISTRYMANAGER_ERROR)
{
    (void)printf("IoTHubRegistryManager_CreateModule failed\n");
}
// You will need to Free the returned module information after it was created
IoTHubRegistryManager_FreeModuleMembers(&moduleInfo);
}

int main(void)
{
    (void)platform_init();

    const char* deviceId = "myFirstDevice";
    const char* moduleId = "myFirstModule";
    IOTHUB_SERVICE_CLIENT_AUTH_HANDLE iotHubServiceClientHandle = NULL;
    IOTHUB_REGISTRYMANAGER_HANDLE iotHubRegistryManagerHandle = NULL;

    if ((iotHubServiceClientHandle = IoTHubServiceClientAuth_CreateFromConnectionString(hubConnectionString))
== NULL)
    {
        (void)printf("IoTHubServiceClientAuth_CreateFromConnectionString failed\n");
    }
    else if ((iotHubRegistryManagerHandle = IoTHubRegistryManager_Create(iotHubServiceClientHandle)) == NULL)
    {
        (void)printf("IoTHubServiceClientAuth_CreateFromConnectionString failed\n");
    }
    else
    {
        createDevice(iotHubRegistryManagerHandle, deviceId);
        createModule(iotHubRegistryManagerHandle, deviceId, moduleId);
    }

    if (iotHubRegistryManagerHandle != NULL)
    {
        (void)printf("Calling IoTHubRegistryManager_Destroy...\n");
        IoTHubRegistryManager_Destroy(iotHubRegistryManagerHandle);
    }

    if (iotHubServiceClientHandle != NULL)
    {
        (void)printf("Calling IoTHubServiceClientAuth_Destroy...\n");
        IoTHubServiceClientAuth_Destroy(iotHubServiceClientHandle);
    }

    platform_deinit();
    return 0;
}

```

This app creates a device identity with ID **myFirstDevice** and a module identity with ID **myFirstModule** under device **myFirstDevice**. (If that module ID already exists in the identity registry, the code simply retrieves the existing module information.) The app then displays the primary key for that identity. You use this key in the simulated module app to connect to your IoT hub.

NOTE

The IoT Hub identity registry only stores device and module identities to enable secure access to the IoT hub. The identity registry stores device IDs and keys to use as security credentials. The identity registry also stores an enabled/disabled flag for each device that you can use to disable access for that device. If your application needs to store other device-specific metadata, it should use an application-specific store. There is no enabled/disabled flag for module identities. For more information, see [IoT Hub developer guide](#).

Update the module twin using C device SDK

In this section, you create a C app on your simulated device that updates the module twin reported properties.

1. **Get your module connection string** -- now if you login to [Azure portal](#). Navigate to your IoT Hub and click IoT Devices. Find myFirstDevice, open it and you see myFirstModule was successfully created. Copy the module connection string. It is needed in the next step.

The screenshot shows the 'Module Identity Twin' page in the Azure portal. At the top, there are buttons for 'Module Identity Twin', 'Direct Method', and 'Delete'. Below these are fields for 'Module Identity Name' containing 'myFirstDevice/myFirstModule', 'Primary key' (redacted), 'Secondary key' (redacted), 'Connection string—primary key' (redacted), and 'Connection string—secondary key' (redacted).

2. **Create UpdateModuleTwinReportedProperties app** Add the following `using` statements at the top of the **Program.cs** file:

```

#include <stdio.h>
#include <stdlib.h>

#include "azure_c_shared_utility/crt_abstractions.h"
#include "azure_c_shared_utility/threadapi.h"
#include "azure_c_shared_utility/platform.h"

#include "iothub_service_client_auth.h"
#include "iothub_devicetwin.h"

const char* deviceId = "bugbash-test-2";
const char* moduleId = "module-id-1";
static const char* hubConnectionString = "[your hub's connection string]"; // modify
const char* testJson = "{\"properties\":{\"desired\":{\"integer_property\": b-1234, \"string_property\": \"abcd\"}}}";

int main(void)
{
    (void)platform_init();

    IOTHUB_SERVICE_CLIENT_AUTH_HANDLE iotHubServiceClientHandle = NULL;
    IOTHUB_SERVICE_CLIENT_DEVICE_TWIN_HANDLE iothubDeviceTwinHandle = NULL;

    if ((iotHubServiceClientHandle =
        IoTHubServiceClientAuth_CreateFromConnectionString(moduleConnectionString)) == NULL)
    {
        (void)printf("IoTHubServiceClientAuth_CreateFromConnectionString failed\n");
    }
    else if ((iothubDeviceTwinHandle = IoTHubDeviceTwin_Create(iotHubServiceClientHandle)) == NULL)
    {
        (void)printf("IoTHubServiceClientAuth_CreateFromConnectionString failed\n");
    }
    else
    {
        char *result = IoTHubDeviceTwin_UpdateModuleTwin(iothubDeviceTwinHandle, deviceId, moduleId,
testJson);
        printf("IoTHubDeviceTwin_UpdateModuleTwin returned %s\n", result);
    }

    if (iothubDeviceTwinHandle != NULL)
    {
        (void)printf("Calling IoTHubDeviceTwin_Destroy...\n");
        IoTHubDeviceTwin_Destroy(iothubDeviceTwinHandle);
    }

    if (iotHubServiceClientHandle != NULL)
    {
        (void)printf("Calling IoTHubServiceClientAuth_Destroy...\n");
        IoTHubServiceClientAuth_Destroy(iotHubServiceClientHandle);
    }

    platform_deinit();
    return 0;
}

```

This code sample shows you how to retrieve the module twin and update reported properties.

Get updates on the device side

In addition to the above code, you can add below code block to get the twin update message on your device.

```

#include <stdio.h>
#include <stdlib.h>

#include "azure_c_shared_utility/crt_abstractions.h"

```

```

-- 
#include "azure_c_shared_utility/macro_utils.h"
#include "azure_c_shared_utility/threadapi.h"
#include "azure_c_shared_utility/platform.h"
#include "iothub_module_client_ll.h"
#include "iothub_client_options.h"
#include "iothub_message.h"

// The protocol you wish to use should be uncommented
//
//#define SAMPLE_MQTT
//#define SAMPLE_MQTT_OVER_WEBSOCKETS
#define SAMPLE_AMQP
//#define SAMPLE_AMQP_OVER_WEBSOCKETS
//#define SAMPLE_HTTP

#ifndef SAMPLE_MQTT
    #include "iothubtransportmqtt.h"
#endif // SAMPLE_MQTT
#ifndef SAMPLE_MQTT_OVER_WEBSOCKETS
    #include "iothubtransportmqtt_websockets.h"
#endif // SAMPLE_MQTT_OVER_WEBSOCKETS
#ifndef SAMPLE_AMQP
    #include "iothubtransportamqp.h"
#endif // SAMPLE_AMQP
#ifndef SAMPLE_AMQP_OVER_WEBSOCKETS
    #include "iothubtransportamqp_websockets.h"
#endif // SAMPLE_AMQP_OVER_WEBSOCKETS
#ifndef SAMPLE_HTTP
    #include "iothubtransporthttp.h"
#endif // SAMPLE_HTTP

/* Paste in the your iothub connection string */
static const char* connectionString = "[Fill in connection string]";

static bool g_continueRunning;
#define DOWORK_LOOP_NUM      3

static void deviceTwinCallback(DEVICE_TWIN_UPDATE_STATE update_state, const unsigned char* payLoad, size_t size, void* userContextCallback)
{
    (void)userContextCallback;

    printf("Device Twin update received (state=%s, size=%zu): %s\r\n",
           ENUM_TO_STRING(DEVICE_TWIN_UPDATE_STATE, update_state), size, payLoad);
}

static void reportedStateCallback(int status_code, void* userContextCallback)
{
    (void)userContextCallback;
    printf("Device Twin reported properties update completed with result: %d\r\n", status_code);

    g_continueRunning = false;
}

void iothub_module_client_sample_device_twin_run(void)
{
    IOTHUB_CLIENT_TRANSPORT_PROVIDER protocol;
    IOTHUB_MODULE_CLIENT_LL_HANDLE iothubModuleClientHandle;
    g_continueRunning = true;

    // Select the Protocol to use with the connection
#ifndef SAMPLE_MQTT
    protocol = MQTT_Protocol;
#endif // SAMPLE_MQTT
#ifndef SAMPLE_MQTT_OVER_WEBSOCKETS
    protocol = MQTT_WebSocket_Protocol;
#endif // SAMPLE_MQTT_OVER_WEBSOCKETS
#ifndef SAMPLE_AMQP
    protocol = AMQP_Protocol;

```

```

    protocol = AMQP_Protocol_over_WebSocketsTls;
#endif // SAMPLE_AMQP_OVER_WEBSOCKETS
#endif // SAMPLE_AMQP_OVER_WEBSOCKETS
#ifndef SAMPLE_HTTP
    protocol = HTTP_Protocol;
#endif // SAMPLE_HTTP

    if (platform_init() != 0)
    {
        (void)printf("Failed to initialize the platform.\r\n");
    }
    else
    {
        if ((iotHubModuleClientHandle = IoTHubModuleClient_LL_CreateFromConnectionString(connectionString,
protocol)) == NULL)
        {
            (void)printf("ERROR: iotHubModuleClientHandle is NULL!\r\n");
        }
        else
        {
            bool traceOn = true;
            const char* reportedState = "{ 'device_property': 'new_value'}";
            size_t reportedStateSize = strlen(reportedState);

            (void)IoTHubModuleClient_LL_SetOption(iotHubModuleClientHandle, OPTION_LOG_TRACE, &traceOn);

            // Check the return of all API calls when developing your solution. Return checks omitted for
sample simplification.

            (void)IoTHubModuleClient_LL_SetModuleTwinCallback(iotHubModuleClientHandle, deviceTwinCallback,
iotHubModuleClientHandle);
            (void)IoTHubModuleClient_LL_SendReportedState(iotHubModuleClientHandle, (const unsigned
char*)reportedState, reportedStateSize, reportedStateCallback, iotHubModuleClientHandle);

            do
            {
                IoTHubModuleClient_LL_DoWork(iotHubModuleClientHandle);
                ThreadAPI_Sleep(1);
            } while (g_continueRunning);

            for (size_t index = 0; index < DOWORK_LOOP_NUM; index++)
            {
                IoTHubModuleClient_LL_DoWork(iotHubModuleClientHandle);
                ThreadAPI_Sleep(1);
            }

            IoTHubModuleClient_LL_Destroy(iotHubModuleClientHandle);
        }
        platform_deinit();
    }
}

int main(void)
{
    iothub_module_client_sample_device_twin_run();
    return 0;
}

```

Next steps

To continue getting started with IoT Hub and to explore other IoT scenarios, see:

- [Getting started with device management](#)
- [Getting started with IoT Edge](#)

Get started with IoT Hub module identity and module twin using Node.js back end and Node.js device

3/13/2019 • 5 minutes to read

NOTE

Module identities and module twins are similar to Azure IoT Hub device identity and device twin, but provide finer granularity. While Azure IoT Hub device identity and device twin enable the back-end application to configure a device and provides visibility on the device's conditions, a module identity and module twin provide these capabilities for individual components of a device. On capable devices with multiple components, such as operating system based devices or firmware devices, it allows for isolated configuration and conditions for each component.

At the end of this tutorial, you have two Node.js apps:

- **CreateIdentities**, which creates a device identity, a module identity and associated security key to connect your device and module clients.
- **UpdateModuleTwinReportedProperties**, which sends updated module twin reported properties to your IoT Hub.

NOTE

For information about the Azure IoT SDKs that you can use to build both applications to run on devices, and your solution back end, see [Azure IoT SDKs](#).

To complete this tutorial, you need the following:

- An active Azure account. (If you don't have an account, you can create a [free account](#) in just a couple of minutes.)
- An IoT Hub.
- Install the latest [Node.js SDK](#).

You have now created your IoT hub, and you have the host name and IoT Hub connection string that you need to complete the rest of this tutorial.

Create a device identity and a module identity in IoT Hub

In this section, you create a Node.js app that creates a device identity and a module identity in the identity registry in your IoT hub. A device or module cannot connect to IoT hub unless it has an entry in the identity registry. For more information, see the "Identity registry" section of the [IoT Hub developer guide](#). When you run this console app, it generates a unique ID and key for both device and module. Your device and module use these values to identify itself when it sends device-to-cloud messages to IoT Hub. The IDs are case-sensitive.

1. Create a directory to hold your code.
2. Inside of that directory, first run **npm init -y** to create an empty package.json with defaults. This is the project file for your code.
3. Run **npm install -S azure-iothub@modules-preview** to install the service SDK inside the **node_modules** subdirectory.

NOTE

The subdirectory name node_modules uses the word module to mean "a node library". The term here has nothing to do with IoT Hub modules.

4. Create the following js file in your directory. Call it **add.js**. Copy and paste your hub connection string and hub name.

```
var Registry = require('azure-iothub').Registry;
var uuid = require('uuid');
// Copy/paste your connection string and hub name here
var serviceConnectionString = '<hub connection string from portal>';
var hubName = '<hub name>.azure-devices.net';
// Create an instance of the IoTHub registry
var registry = Registry.fromConnectionString(serviceConnectionString);
// Insert your device ID and moduleId here.
var deviceId = 'myFirstDevice';
var moduleId = 'myFirstModule';
// Create your device as a SAS authentication device
var primaryKey = new Buffer(uuid.v4()).toString('base64');
var secondaryKey = new Buffer(uuid.v4()).toString('base64');
var deviceDescription = {
    deviceId: deviceId,
    status: 'enabled',
    authentication: {
        type: 'sas',
        symmetricKey: {
            primaryKey: primaryKey,
            secondaryKey: secondaryKey
        }
    }
};

// First, create a device identity
registry.create(deviceDescription, function(err) {
    if (err) {
        console.log('Error creating device identity: ' + err);
        process.exit(1);
    }
    console.log('device connection string = "HostName=' + hubName + ';DeviceId=' + deviceId +
    ';SharedAccessKey=' + primaryKey + "'");

    // Then add a module to that device
    registry.addModule({ deviceId: deviceId, moduleId: moduleId }, function(err) {
        if (err) {
            console.log('Error creating module identity: ' + err);
            process.exit(1);
        }

        // Finally, retrieve the module details from the hub so we can construct the connection string
        registry.getModule(deviceId, moduleId, function(err, foundModule) {
            if (err) {
                console.log('Error getting module back from hub: ' + err);
                process.exit(1);
            }
            console.log('module connection string = "HostName=' + hubName + ';DeviceId=' +
            foundModule.deviceId + ';ModuleId=' + foundModule.moduleId +';SharedAccessKey=' +
            foundModule.authentication.symmetricKey.primaryKey + "'");

            process.exit(0);
        });
    });
});
```

This app creates a device identity with ID **myFirstDevice** and a module identity with ID **myFirstModule** under

device **myFirstDevice**. (If that module ID already exists in the identity registry, the code simply retrieves the existing module information.) The app then displays the primary key for that identity. You use this key in the simulated module app to connect to your IoT hub.

1. Run this using node add.js. It will give you a connection string for your device identity and another one for your module identity.

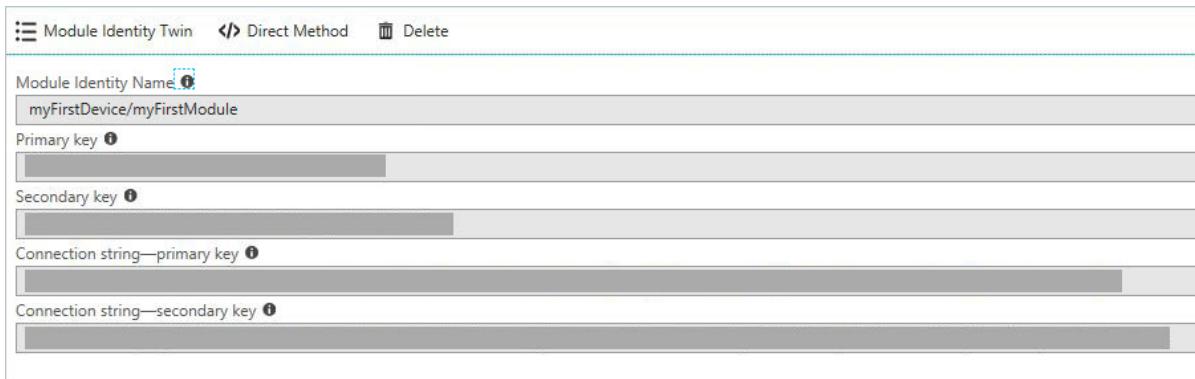
NOTE

The IoT Hub identity registry only stores device and module identities to enable secure access to the IoT hub. The identity registry stores device IDs and keys to use as security credentials. The identity registry also stores an enabled/disabled flag for each device that you can use to disable access for that device. If your application needs to store other device-specific metadata, it should use an application-specific store. There is no enabled/disabled flag for module identities. For more information, see [IoT Hub developer guide](#).

Update the module twin using Node.js device SDK

In this section, you create a Node.js app on your simulated device that updates the module twin reported properties.

1. **Get your module connection string** -- now if you login to [Azure portal](#). Navigate to your IoT Hub and click IoT Devices. Find myFirstDevice, open it and you see myFirstModule was successfully created. Copy the module connection string. It is needed in the next step.



2. Similar to what you did in the step above, create a directory for your device code and use NPM to initialize it and install the device SDK (**npm install -S azure-iot-device-amqp@modules-preview**).

NOTE

The npm install command may feel slow. Be patient, it's pulling down lots of code from the package repository.

NOTE

If you see an error that says npm ERR! registry error parsing json, this is safe to ignore. If you see an error that says npm ERR! registry error parsing json, this is safe to ignore.

3. Create a file called twin.js. Copy and paste your module identity string.

```

var Client = require('azure-iot-device').Client;
var Protocol = require('azure-iot-device-amqp').Amqp;
// Copy/paste your module connection string here.
var connectionString = '<insert module connection string here>';
// Create a client using the Amqp protocol.
var client = Client.fromConnectionString(connectionString, Protocol);
client.on('error', function (err) {
    console.error(err.message);
});
// connect to the hub
client.open(function(err) {
    if (err) {
        console.error('error connecting to hub: ' + err);
        process.exit(1);
    }
    console.log('client opened');
// Create device Twin
    client.getTwin(function(err, twin) {
        if (err) {
            console.error('error getting twin: ' + err);
            process.exit(1);
        }
        // Output the current properties
        console.log('twin contents:');
        console.log(twin.properties);
// Add a handler for desired property changes
        twin.on('properties.desired', function(delta) {
            console.log('new desired properties received:');
            console.log(JSON.stringify(delta));
        });
// create a patch to send to the hub
        var patch = {
            updateTime: new Date().toString(),
            firmwareVersion:'1.2.1',
            weather:{
                temperature: 72,
                humidity: 17
            }
        };
// send the patch
        twin.properties.reported.update(patch, function(err) {
            if (err) throw err;
            console.log('twin state reported');
        });
    });
});
});

```

4. Now, run this using the command **node twin.js**.

```

F:\temp\module_twin>node twin.js
client opened
twin contents:
{ reported: { update: [Function: update], '$version': 1 },
  desired: { '$version': 1 } }
new desired properties received:
{"$version":1}
twin state reported

```

Next steps

To continue getting started with IoT Hub and to explore other IoT scenarios, see:

- [Getting started with device management](#)

- [Getting started with IoT Edge](#)

Get started with device management (Node)

3/6/2019 • 10 minutes to read

Back-end apps can use Azure IoT Hub primitives, such as [device twin](#) and [direct methods](#), to remotely start and monitor device management actions on devices. This tutorial shows you how a back-end app and a device app can work together to initiate and monitor a remote device reboot using IoT Hub.

NOTE

The features described in this article are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

Use a direct method to initiate device management actions (such as reboot, factory reset, and firmware update) from a back-end app in the cloud. The device is responsible for:

- Handling the method request sent from IoT Hub.
- Initiating the corresponding device-specific action on the device.
- Providing status updates through *reported properties* to IoT Hub.

You can use a back-end app in the cloud to run device twin queries to report on the progress of your device management actions.

This tutorial shows you how to:

- Use the Azure portal to create an IoT Hub and create a device identity in your IoT hub.
- Create a simulated device app that contains a direct method that reboots that device. Direct methods are invoked from the cloud.
- Create a Node.js console app that calls the reboot direct method in the simulated device app through your IoT hub.

At the end of this tutorial, you have two Node.js console apps:

dmpatterns_getstarted_device.js, which connects to your IoT hub with the device identity created earlier, receives a reboot direct method, simulates a physical reboot, and reports the time for the last reboot.

dmpatterns_getstarted_service.js, which calls a direct method in the simulated device app, displays the response, and displays the updated reported properties.

To complete this tutorial, you need the following:

- Node.js version 4.0.x or later,
[Prepare your development environment](#) describes how to install Node.js for this tutorial on either Windows or Linux.
- An active Azure account. (If you don't have an account, you can create a [free account](#) in just a couple of minutes.)

Create an IoT hub

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose +**Create a resource**, then choose **Internet of Things**.

3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.

The screenshot shows the 'Basics' step of the IoT Hub creation wizard. The 'Subscription' dropdown is set to 'Microsoft Azure Internal Consumption'. The 'Resource Group' dropdown shows 'Create new' selected, with 'contoso-hub-rgrp' listed. The 'Region' dropdown is set to 'West US'. The 'IoT Hub Name' field contains 'contoso-test-hub', which has a green checkmark next to it. At the bottom, there are three buttons: 'Review + create' (blue), 'Next: Size and scale »' (red box), and 'Automation options'.

Fill in the fields.

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Review + create

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS

* Pricing and scale tier [?](#) S1: Standard tier [?](#) Learn how to choose the right IoT Hub tier for your solution

Number of S1 IoT Hub units [?](#) 1 This determines your IoT Hub scale capability and can be changed as your need increases.

Device-to-cloud-messages [?](#) Enabled

Message routing [?](#) Enabled

Cloud-to-device commands [?](#) Enabled

IoT Edge [?](#) Enabled

Device management [?](#) Enabled

Advanced Settings

Device-to-cloud partitions [?](#) 4

Review + create [« Previous: Basics](#) [Automation options](#)

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: The number of messages allowed per unit per day depends on your hub's pricing tier. For example, if you want the IoT hub to support ingress of 700,000 messages, you choose two S1 tier units.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

Advanced / Device-to-cloud partitions: This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most IoT hubs only need four partitions.

5. Click **Review + create** to review your choices. You see something similar to this screen.

The screenshot shows the 'Review + create' step of the IoT hub creation wizard. It displays basic information like subscription, resource group, region, and IoT Hub name, as well as size and scale details. At the bottom, the 'Create' button is highlighted with a red box.

6. Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Retrieve connection string for IoT hub

After your hub has been created, retrieve the connection string for the hub. This is used to connect devices and applications to your hub.

1. Click on your hub to see the IoT Hub pane with Settings, and so on. Click **Shared access policies**.
2. In **Shared access policies**, select the **iothubowner** policy.
3. Under **Shared access keys**, copy the **Connection string -- primary key** to be used later.

The screenshot shows the 'Shared access policies' blade for the ContosoHub IoT Hub. It displays the 'iothubowner' policy configuration. The 'Connection string--primary key' field is highlighted with a red box.

For more information, see [Access control](#) in the "IoT Hub developer guide."

Create a device identity

In this section, you use the Azure CLI to create a device identity for this tutorial. The Azure CLI is preinstalled in the [Azure Cloud Shell](#), or you can [install it locally](#). Device IDs are case sensitive.

1. Run the following command in the command-line environment where you are using the Azure CLI to install the IoT extension:

```
az extension add --name azure-cli-iot-ext
```

2. If you are running the Azure CLI locally, use the following command to sign in to your Azure account (if you are using the Cloud Shell, you are signed in automatically and you don't need to run this command):

```
az login
```

3. Finally, create a new device identity called `myDeviceId` and retrieve the device connection string with these commands:

```
az iot hub device-identity create --device-id myDeviceId --hub-name {Your IoT Hub name}  
az iot hub device-identity show-connection-string --device-id myDeviceId --hub-name {Your IoT Hub name} -o table
```

IMPORTANT

The device ID may be visible in the logs collected for customer support and troubleshooting, so make sure to avoid any sensitive information while naming it.

Make a note of the device connection string from the result. This device connection string is used by the device app to connect to your IoT Hub as a device.

Create a simulated device app

In this section, you will

- Create a Node.js console app that responds to a direct method called by the cloud
- Trigger a simulated device reboot
- Use the reported properties to enable device twin queries to identify devices and when they last rebooted

1. Create an empty folder called **manageddevice**. In the **manageddevice** folder, create a package.json file using the following command at your command prompt. Accept all the defaults:

```
npm init
```

2. At your command prompt in the **manageddevice** folder, run the following command to install the **azure-iot-device** Device SDK package and **azure-iot-device-mqtt** package:

```
npm install azure-iot-device azure-iot-device-mqtt --save
```

3. Using a text editor, create a **dmpatterns_getstarted_device.js** file in the **manageddevice** folder.
4. Add the following 'require' statements at the start of the **dmpatterns_getstarted_device.js** file:

```
'use strict';

var Client = require('azure-iot-device').Client;
var Protocol = require('azure-iot-device-mqtt').Mqtt;
```

5. Add a **connectionString** variable and use it to create a **Client** instance. Replace the connection string with your device connection string.

```
var connectionString = 'HostName={youriothostname};DeviceId=myDeviceId;SharedAccessKey=
{yourdevicekey}';
var client = Client.fromConnectionString(connectionString, Protocol);
```

6. Add the following function to implement the direct method on the device

```
var onReboot = function(request, response) {

    // Respond the cloud app for the direct method
    response.send(200, 'Reboot started', function(err) {
        if (err) {
            console.error('An error occurred when sending a method response:\n' + err.toString());
        } else {
            console.log('Response to method \'' + request.methodName + '\' sent successfully.');
        }
    });

    // Report the reboot before the physical restart
    var date = new Date();
    var patch = {
        iothubDM : {
            reboot : {
                lastReboot : date.toISOString(),
            }
        }
    };

    // Get device Twin
    client.getTwin(function(err, twin) {
        if (err) {
            console.error('could not get twin');
        } else {
            console.log('twin acquired');
            twin.properties.reported.update(patch, function(err) {
                if (err) throw err;
                console.log('Device reboot twin state reported')
            });
        }
    });

    // Add your device's reboot API for physical restart.
    console.log('Rebooting!');
};
```

7. Open the connection to your IoT hub and start the direct method listener:

```
client.open(function(err) {
  if (err) {
    console.error('Could not open IoT Hub client');
  } else {
    console.log('Client opened. Waiting for reboot method.');
    client.onDeviceMethod('reboot', onReboot);
  }
});
```

- Save and close the **dmpatterns_getstarted_device.js** file.

NOTE

To keep things simple, this tutorial does not implement any retry policy. In production code, you should implement retry policies (such as an exponential backoff), as suggested in the article, [Transient Fault Handling](#).

Trigger a remote reboot on the device using a direct method

In this section, you create a Node.js console app that initiates a remote reboot on a device using a direct method. The app uses device twin queries to discover the last reboot time for that device.

- Create an empty folder called **triggerrebootondevice**. In the **triggerrebootondevice** folder, create a `package.json` file using the following command at your command prompt. Accept all the defaults:

```
npm init
```

- At your command prompt in the **triggerrebootondevice** folder, run the following command to install the **azure-iothub** Device SDK package and **azure-iot-device-mqtt** package:

```
npm install azure-iothub --save
```

- Using a text editor, create a **dmpatterns_getstarted_service.js** file in the **triggerrebootondevice** folder.

- Add the following 'require' statements at the start of the **dmpatterns_getstarted_service.js** file:

```
'use strict';

var Registry = require('azure-iothub').Registry;
var Client = require('azure-iothub').Client;
```

- Add the following variable declarations and replace the placeholder values:

```
var connectionString = '{iothubconnectionstring}';
var registry = Registry.fromConnectionString(connectionString);
var client = Client.fromConnectionString(connectionString);
var deviceToReboot = 'myDeviceId';
```

- Add the following function to invoke the device method to reboot the target device:

```

var startRebootDevice = function(twin) {

    var methodName = "reboot";

    var methodParams = {
        methodName: methodName,
        payload: null,
        timeoutInSeconds: 30
    };

    client.invokeDeviceMethod(deviceToReboot, methodParams, function(err, result) {
        if (err) {
            console.error("Direct method error: "+err.message);
        } else {
            console.log("Successfully invoked the device to reboot.");
        }
    });
};

}

```

- Add the following function to query for the device and get the last reboot time:

```

var queryTwinLastReboot = function() {

    registry.getTwin(deviceToReboot, function(err, twin){

        if (twin.properties.reported.iothubDM != null)
        {
            if (err) {
                console.error('Could not query twins: ' + err.constructor.name + ': ' + err.message);
            } else {
                var lastRebootTime = twin.properties.reported.iothubDM.reboot.lastReboot;
                console.log('Last reboot time: ' + JSON.stringify(lastRebootTime, null, 2));
            }
        } else
            console.log('Waiting for device to report last reboot time.');
    });
};

```

- Add the following code to call the functions that trigger the reboot direct method and query for the last reboot time:

```

startRebootDevice();
setInterval(queryTwinLastReboot, 2000);

```

- Save and close the **dmpatterns_getstarted_service.js** file.

Run the apps

You are now ready to run the apps.

- At the command prompt in the **manageddevice** folder, run the following command to begin listening for the reboot direct method.

```

node dmpatterns_getstarted_device.js

```

- At the command prompt in the **triggerrebootondevice** folder, run the following command to trigger the remote reboot and query for the device twin to find the last reboot time.

```
node dmpatterns_getstarted_service.js
```

3. You see the device response to the direct method in the console.

Customize and extend the device management actions

Your IoT solutions can expand the defined set of device management patterns or enable custom patterns by using the device twin and cloud-to-device method primitives. Other examples of device management actions include factory reset, firmware update, software update, power management, network and connectivity management, and data encryption.

Device maintenance windows

Typically, you configure devices to perform actions at a time that minimizes interruptions and downtime. Device maintenance windows are a commonly used pattern to define the time when a device should update its configuration. Your back-end solutions can use the desired properties of the device twin to define and activate a policy on your device that enables a maintenance window. When a device receives the maintenance window policy, it can use the reported property of the device twin to report the status of the policy. The back-end app can then use device twin queries to attest to compliance of devices and each policy.

Next steps

In this tutorial, you used a direct method to trigger a remote reboot on a device. You used the reported properties to report the last reboot time from the device, and queried the device twin to discover the last reboot time of the device from the cloud.

To continue getting started with IoT Hub and device management patterns such as remote over the air firmware update, see [How to do a firmware update](#)

To learn how to extend your IoT solution and schedule method calls on multiple devices, see [Schedule and broadcast jobs](#).

Get started with device management (.NET/.NET)

2/28/2019 • 11 minutes to read

Back-end apps can use Azure IoT Hub primitives, such as [device twin](#) and [direct methods](#), to remotely start and monitor device management actions on devices. This tutorial shows you how a back-end app and a device app can work together to initiate and monitor a remote device reboot using IoT Hub.

NOTE

The features described in this article are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

Use a direct method to initiate device management actions (such as reboot, factory reset, and firmware update) from a back-end app in the cloud. The device is responsible for:

- Handling the method request sent from IoT Hub.
- Initiating the corresponding device-specific action on the device.
- Providing status updates through *reported properties* to IoT Hub.

You can use a back-end app in the cloud to run device twin queries to report on the progress of your device management actions.

This tutorial shows you how to:

- Use the Azure portal to create an IoT Hub and create a device identity in your IoT hub.
- Create a simulated device app that contains a direct method that reboots that device. Direct methods are invoked from the cloud.
- Create a .NET console app that calls the reboot direct method in the simulated device app through your IoT hub.

At the end of this tutorial, you have two .NET console apps:

- **SimulateManagedDevice**, which connects to your IoT hub with the device identity created earlier, receives a reboot direct method, simulates a physical reboot, and reports the time for the last reboot.
- **TriggerReboot**, which calls a direct method in the simulated device app, displays the response, and displays the updated reported properties.

To complete this tutorial, you need the following:

- Visual Studio 2017.
- An active Azure account. (If you don't have an account, you can create a [free account](#) in just a couple of minutes.)

Create an IoT hub

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose +**Create a resource**, then choose **Internet of Things**.

3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.

Create an IoT Hub to help you connect, monitor, and manage billions of your IoT assets. [Learn More](#)

PROJECT DETAILS

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

* Subscription

* Resource Group Create new Use existing

* Region

* IoT Hub Name

[Review + create](#) [Next: Size and scale >](#) [Automation options](#)

Fill in the fields.

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics **Size and scale** Review + create

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS

* Pricing and scale tier **S1: Standard tier** [Learn how to choose the right IoT Hub tier for your solution](#)

Number of S1 IoT Hub units **1** This determines your IoT Hub scale capability and can be changed as your need increases.

Device-to-cloud-messages **Enabled**
Message routing **Enabled**
Cloud-to-device commands **Enabled**
IoT Edge **Enabled**
Device management **Enabled**

Advanced Settings

Device-to-cloud partitions **4**

Review + create [« Previous: Basics](#) [Automation options](#)

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: The number of messages allowed per unit per day depends on your hub's pricing tier. For example, if you want the IoT hub to support ingress of 700,000 messages, you choose two S1 tier units.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

Advanced / Device-to-cloud partitions: This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most IoT hubs only need four partitions.

5. Click **Review + create** to review your choices. You see something similar to this screen.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale **Review + create**

BASICS

Subscription	Microsoft Azure Internal Consumption
Resource Group	contoso-hub-rgrp
Region	West US
IoT Hub Name	contoso-test-hub

SIZE AND SCALE

Pricing and scale tier	S1
Number of S1 IoT Hub units	1
Messages per day	400,000
Cost per month	25.00 USD

Create « Previous: Size and scale Automation options

6. Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Retrieve connection string for IoT hub

After your hub has been created, retrieve the connection string for the hub. This is used to connect devices and applications to your hub.

1. Click on your hub to see the IoT Hub pane with Settings, and so on. Click **Shared access policies**.
2. In **Shared access policies**, select the **iothubowner** policy.
3. Under **Shared access keys**, copy the **Connection string -- primary key** to be used later.

Home > IoT Hub > ContosoHub - Shared access policies

ContosoHub - Shared access policies

IoT Hub

Overview Activity log Access control (IAM) Tags Events

SETTINGS

- Shared access policies **Selected**
- Pricing and scale
- Operations monitoring
- IP Filter
- Certificates
- Properties
- Locks
- Automation script

Add

IoT Hub uses permissions to grant access to each functionality.

Search to filter items...

POLICY

- iothubowner **Selected**
- service
- device
- registryRead
- registryReadWrite

iothubowner

Save Discard Regen key Delete

Access policy name: iothubowner

Permissions:

- Registry read
- Registry write
- Service connect
- Device connect

Shared access keys:

Primary key: **HostName=ContosoHub.azure-devices.net;SharedAccessKey...**

Secondary key: **HostName=ContosoHub.azure-devices.net;SharedAccessKey...**

Connection string--primary key: **HostName=ContosoHub.azure-devices.net;SharedAccessKey...**

Connection string--secondary key: **HostName=ContosoHub.azure-devices.net;SharedAccessKey...**

For more information, see [Access control](#) in the "IoT Hub developer guide."

Register a new device in the IoT hub

In this section, you create a device identity in the identity registry in your IoT hub. A device cannot connect to IoT hub unless it has an entry in the identity registry. For more information, see the "Identity registry" section of the [IoT Hub developer guide](#)

1. In your IoT hub navigation menu, open **IoT Devices**, then click **Add** to register a new device in your IoT hub.

The screenshot shows the 'ContosoHub - IoT devices' blade in the Azure portal. The left sidebar contains a navigation menu with several items: 'Events', 'SETTINGS' (with 'Shared access policies', 'Pricing and scale', 'Operations monitoring', 'IP Filter', 'Certificates', 'Properties', 'Locks', and 'Automation script'), 'EXPLORERS' (with 'Query explorer' and 'IoT devices' highlighted with a red box), and 'AUTOMATIC DEVICE MANAGEMENT' (with 'IoT Edge (preview)'). The main area features a large 'Add' button at the top left, a query editor with a sample SQL query, and a table below showing 'No results'. The table columns are: DEVICE ID, STATUS, LAST ACTIVITY, LAST STATUS..., AUTHENTICATI..., and CLOUD TO DEV... .

2. Provide a name for your new device, such as **myDeviceId**, and click **Save**. This action creates a new device identity for your IoT hub.

Create a device

Learn more about creating devices

* Device ID ⓘ
myDeviceId

Authentication type ⓘ
Symmetric key X.509 Self-Signed X.509 CA Signed

* Primary key ⓘ
Enter your primary key

* Secondary key ⓘ
Enter your secondary key

Auto-generate keys ⓘ

Connect this device to an IoT hub ⓘ
Enable Disable

Parent device (Preview) ⓘ
No parent device
Set a parent device

Save

IMPORTANT

The device ID may be visible in the logs collected for customer support and troubleshooting, so make sure to avoid any sensitive information while naming it.

3. After the device is created, open the device from the list in the **IoT devices** pane. Copy the **Connection string---primary key** to use later.

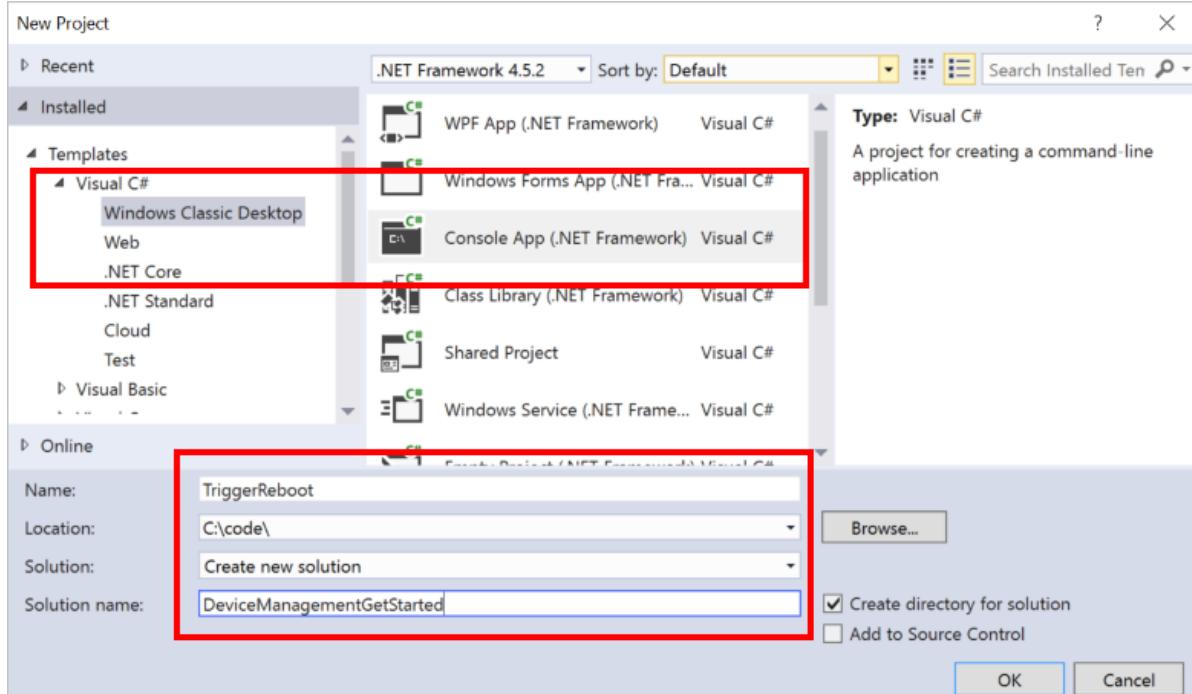
NOTE

The IoT Hub identity registry only stores device identities to enable secure access to the IoT hub. It stores device IDs and keys to use as security credentials, and an enabled/disabled flag that you can use to disable access for an individual device. If your application needs to store other device-specific metadata, it should use an application-specific store. For more information, see [IoT Hub developer guide](#).

Trigger a remote reboot on the device using a direct method

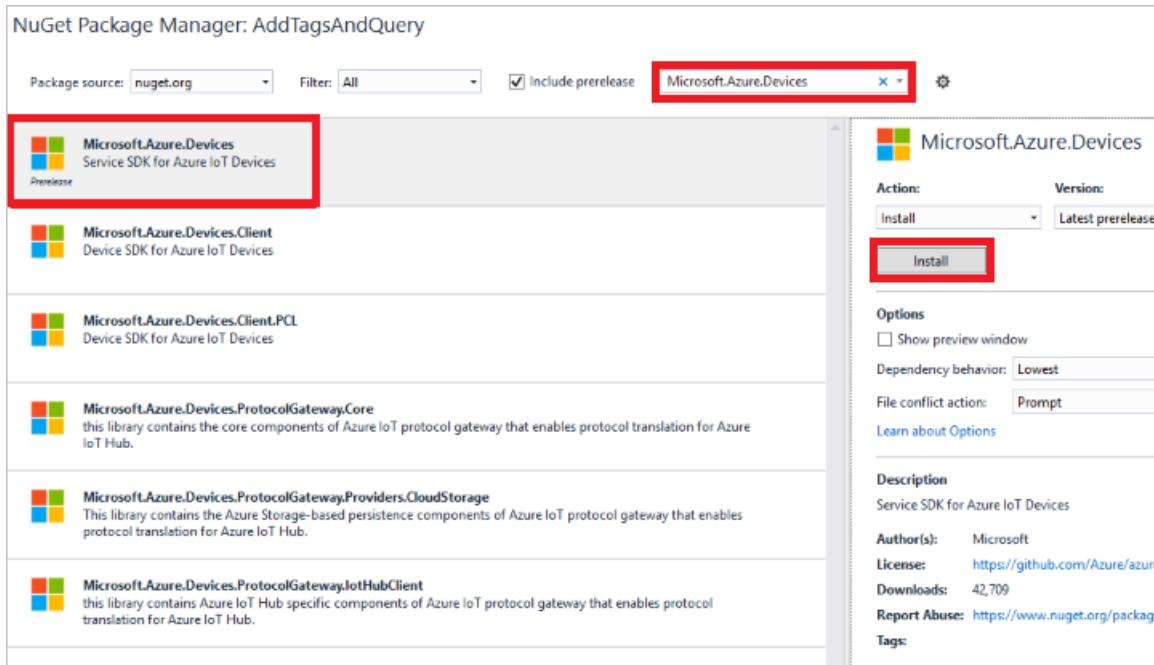
In this section, you create a .NET console app (using C#) that initiates a remote reboot on a device using a direct method. The app uses device twin queries to discover the last reboot time for that device.

1. In Visual Studio, add a Visual C# Windows Classic Desktop project to a new solution by using the **Console App (.NET Framework)** project template. Make sure the .NET Framework version is 4.5.1 or later. Name the project **TriggerReboot**.



2. In Solution Explorer, right-click the **TriggerReboot** project, and then click **Manage NuGet Packages**.
3. In the **NuGet Package Manager** window, select **Browse**, search for **Microsoft.Azure.Devices**, select **Install** to install the **Microsoft.Azure.Devices** package, and accept the terms of use. This procedure downloads, installs, and adds a reference to the [Azure IoT service SDK](#) NuGet package and its

dependencies.



4. Add the following `using` statements at the top of the **Program.cs** file:

```
using Microsoft.Azure.Devices;
using Microsoft.Azure.Devices.Shared;
```

5. Add the following fields to the **Program** class. Replace the placeholder value with the IoT Hub connection string for the hub that you created in the section "Create an IoT hub."

```
static RegistryManager registryManager;
static string connString = "{iot hub connection string}";
static ServiceClient client;
static string targetDevice = "myDeviceId";
```

6. Add the following method to the **Program** class. This code gets the device twin for the rebooting device and outputs the reported properties.

```
public static async Task QueryTwinRebootReported()
{
    Twin twin = await registryManager.GetTwinAsync(targetDevice);
    Console.WriteLine(twin.Properties.Reported.ToJson());
}
```

7. Add the following method to the **Program** class. This code initiates the reboot on the device using a direct method.

```
public static async Task StartReboot()
{
    client = ServiceClient.CreateFromConnectionString(connString);
    CloudToDeviceMethod method = new CloudToDeviceMethod("reboot");
    method.ResponseTimeout = TimeSpan.FromSeconds(30);

    CloudToDeviceMethodResult result = await
        client.InvokeDeviceMethodAsync(targetDevice, method);

    Console.WriteLine("Invoked firmware update on device.");
}
```

- Finally, add the following lines to the **Main** method:

```
registryManager = RegistryManager.CreateFromConnectionString(connString);
StartReboot().Wait();
QueryTwinRebootReported().Wait();
Console.WriteLine("Press ENTER to exit.");
Console.ReadLine();
```

- Build the solution.

NOTE

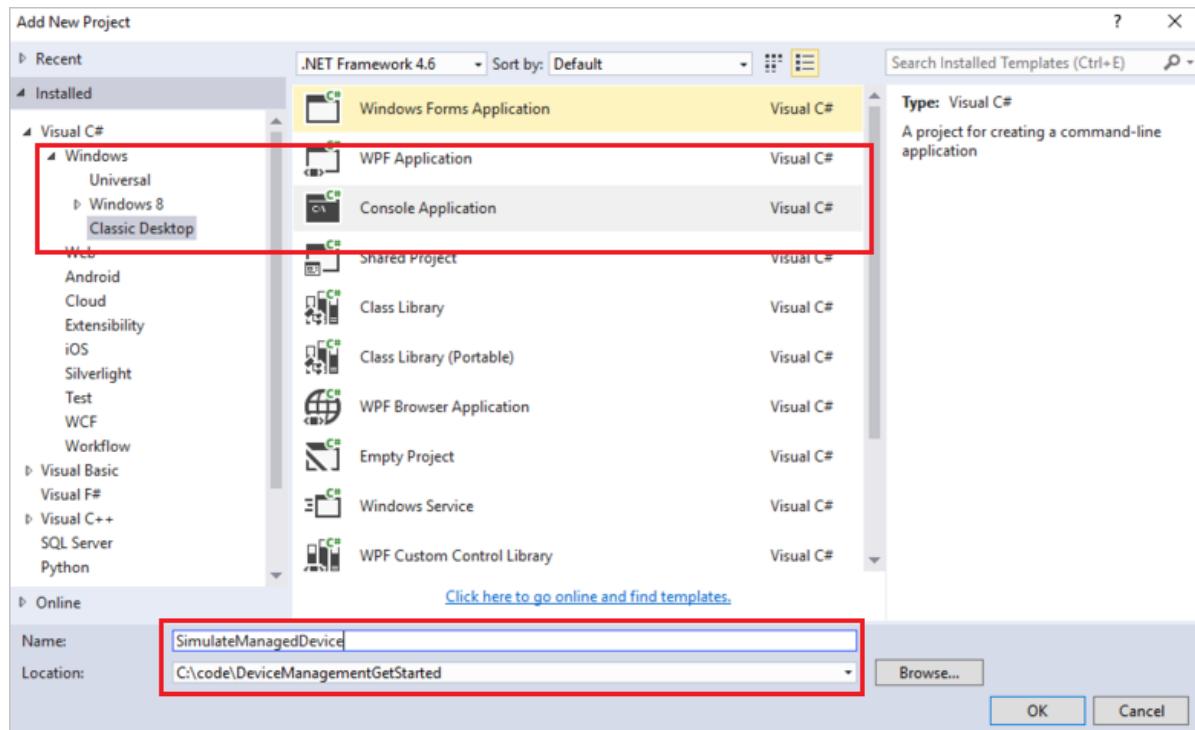
This tutorial performs only a single query for the device's reported properties. In production code, we recommend polling to detect changes in the reported properties.

Create a simulated device app

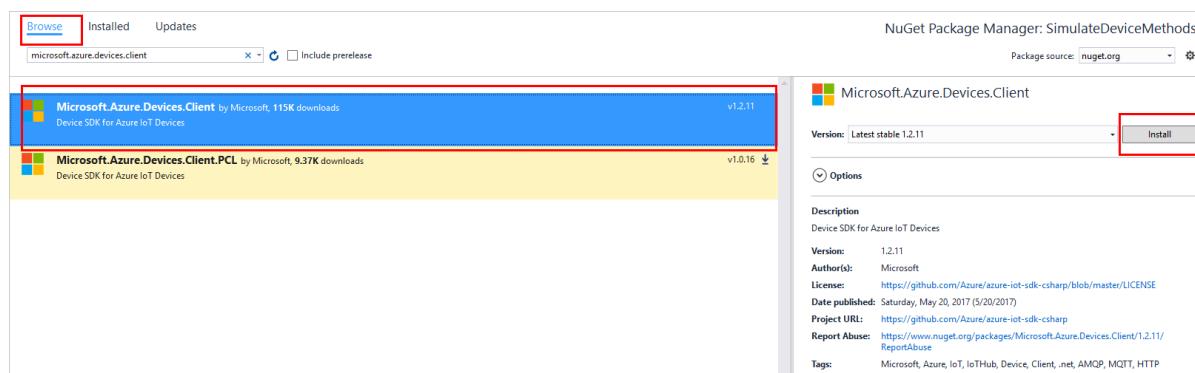
In this section, you do the following:

- Create a .NET console app that responds to a direct method called by the cloud.
- Trigger a simulated device reboot.
- Use the reported properties to enable device twin queries to identify devices and when they were last rebooted.

- In Visual Studio, add a Visual C# Windows Classic Desktop project to the current solution by using the **Console Application** project template. Name the project **SimulateManagedDevice**.



2. In Solution Explorer, right-click the **SimulateManagedDevice** project, and then click **Manage NuGet Packages....**
3. In the **NuGet Package Manager** window, select **Browse** and search for **Microsoft.Azure.Devices.Client**. Select **Install** to install the **Microsoft.Azure.Devices.Client** package, and accept the terms of use. This procedure downloads, installs, and adds a reference to the **Azure IoT device SDK** NuGet package and its dependencies.



4. Add the following `using` statements at the top of the **Program.cs** file:

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

5. Add the following fields to the **Program** class. Replace the placeholder value with the device connection string that you noted in the previous section.

```
static string DeviceConnectionString =
    "HostName=<yourIoTHubName>.azure-devices.net;DeviceId=<yourIoTDeviceName>;SharedAccessKey=<yourIoTDeviceAccessKey>";
static DeviceClient Client = null;
```

6. Add the following to implement the direct method on the device:

```

static Task<MethodResponse> onReboot(MethodRequest methodRequest, object userContext)
{
    // In a production device, you would trigger a reboot
    // scheduled to start after this method returns.
    // For this sample, we simulate the reboot by writing to the console
    // and updating the reported properties.
    try
    {
        Console.WriteLine("Rebooting!");

        // Update device twin with reboot time.
        TwinCollection reportedProperties, reboot, lastReboot;
        lastReboot = new TwinCollection();
        reboot = new TwinCollection();
        reportedProperties = new TwinCollection();
        lastReboot["lastReboot"] = DateTime.Now;
        reboot["reboot"] = lastReboot;
        reportedProperties["iothubDM"] = reboot;
        Client.UpdateReportedPropertiesAsync(reportedProperties).Wait();
    }
    catch (Exception ex)
    {
        Console.WriteLine();
        Console.WriteLine("Error in sample: {0}", ex.Message);
    }

    string result = "'Reboot started.'";
    return Task.FromResult(new MethodResponse(Encoding.UTF8.GetBytes(result), 200));
}

```

- Finally, add the following code to the **Main** method to open the connection to your IoT hub and initialize the method listener:

```

try
{
    Console.WriteLine("Connecting to hub");
    Client = DeviceClient.CreateFromConnectionString(DeviceConnectionString,
        TransportType.Mqtt);

    // setup callback for "reboot" method
    Client.SetMethodHandlerAsync("reboot", onReboot, null).Wait();
    Console.WriteLine("Waiting for reboot method\n Press enter to exit.");
    Console.ReadLine();

    Console.WriteLine("Exiting...");

    // as a good practice, remove the "reboot" handler
    Client.SetMethodHandlerAsync("reboot", null, null).Wait();
    Client.CloseAsync().Wait();
}
catch (Exception ex)
{
    Console.WriteLine();
    Console.WriteLine("Error in sample: {0}", ex.Message);
}

```

- In the Visual Studio Solution Explorer, right-click your solution, and then click **Set StartUp Projects...**. Select **Single startup project**, and then select the **SimulateManagedDevice** project in the dropdown menu. Build the solution.

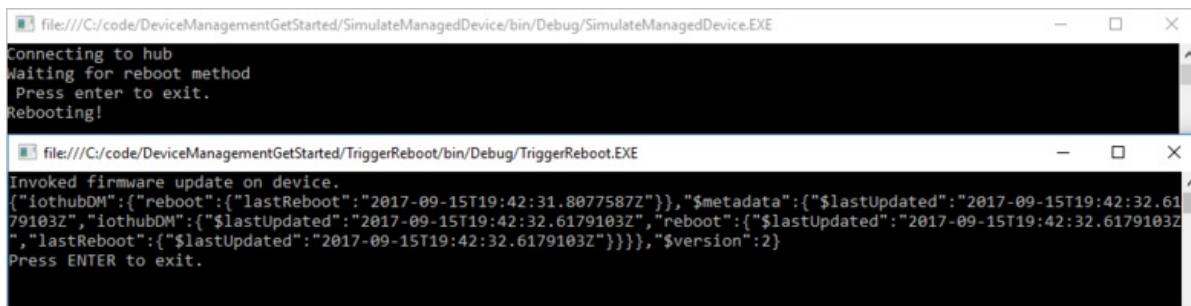
NOTE

To keep things simple, this tutorial does not implement any retry policy. In production code, you should implement retry policies (such as an exponential backoff), as suggested in the article, [Transient Fault Handling](#).

Run the apps

You are now ready to run the apps.

1. To run the .NET device app **SimulateManagedDevice**, right-click the **SimulateManagedDevice** project, select **Debug**, and then select **Start new instance**. It should start listening for method calls from your IoT hub.
2. Now that the device is connected and waiting for method invocations, run the .NET **TriggerReboot** app to invoke the reboot method in the simulated device app. To do this, right-click the **TriggerReboot** project, select **Debug**, and then select **Start new instance**. You should see "Rebooting!" written in the **SimulatedManagedDevice** console and the reported properties of the device, which include the last reboot time, written in the **TriggerReboot** console.



The image shows two terminal windows side-by-side. The top window is titled 'file:///C:/code/DeviceManagementGetStarted/SimulateManagedDevice/bin/Debug/SimulateManagedDevice.EXE'. Its output is:

```
Connecting to hub
Waiting for reboot method
Press enter to exit.
Rebooting!
```

The bottom window is titled 'file:///C:/code/DeviceManagementGetStarted/TriggerReboot/bin/Debug/TriggerReboot.EXE'. Its output is:

```
Invoked firmware update on device.
{"iothubDM":{"reboot":{"lastReboot":"2017-09-15T19:42:31.8077587Z"}}, "$metadata": {"$lastUpdated": "2017-09-15T19:42:32.6179103Z", "iothubDM": {"$lastUpdated": "2017-09-15T19:42:32.6179103Z", "reboot": {"$lastUpdated": "2017-09-15T19:42:32.6179103Z", "lastReboot": {"$lastUpdated": "2017-09-15T19:42:32.6179103Z"}}}, "$version": 2}
Press ENTER to exit.
```

Customize and extend the device management actions

Your IoT solutions can expand the defined set of device management patterns or enable custom patterns by using the device twin and cloud-to-device method primitives. Other examples of device management actions include factory reset, firmware update, software update, power management, network and connectivity management, and data encryption.

Device maintenance windows

Typically, you configure devices to perform actions at a time that minimizes interruptions and downtime. Device maintenance windows are a commonly used pattern to define the time when a device should update its configuration. Your back-end solutions can use the desired properties of the device twin to define and activate a policy on your device that enables a maintenance window. When a device receives the maintenance window policy, it can use the reported property of the device twin to report the status of the policy. The back-end app can then use device twin queries to attest to compliance of devices and each policy.

Next steps

In this tutorial, you used a direct method to trigger a remote reboot on a device. You used the reported properties to report the last reboot time from the device, and queried the device twin to discover the last reboot time of the device from the cloud.

To continue getting started with IoT Hub and device management patterns such as remote over the air firmware update, see [How to do a firmware update](#)

To learn how to extend your IoT solution and schedule method calls on multiple devices, see [Schedule and](#)

broadcast jobs.

Get started with device management (Java)

3/6/2019 • 14 minutes to read

Back-end apps can use Azure IoT Hub primitives, such as [device twin](#) and [direct methods](#), to remotely start and monitor device management actions on devices. This tutorial shows you how a back-end app and a device app can work together to initiate and monitor a remote device reboot using IoT Hub.

NOTE

The features described in this article are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

Use a direct method to initiate device management actions (such as reboot, factory reset, and firmware update) from a back-end app in the cloud. The device is responsible for:

- Handling the method request sent from IoT Hub.
- Initiating the corresponding device-specific action on the device.
- Providing status updates through *reported properties* to IoT Hub.

You can use a back-end app in the cloud to run device twin queries to report on the progress of your device management actions.

This tutorial shows you how to:

- Use the Azure portal to create an IoT Hub and create a device identity in your IoT hub.
- Create a simulated device app that implements a direct method to reboot the device. Direct methods are invoked from the cloud.
- Create an app that invokes the reboot direct method in the simulated device app through your IoT hub. This app then monitors the reported properties from the device to see when the reboot operation is complete.

At the end of this tutorial, you have two Java console apps:

simulated-device. This app:

- Connects to your IoT hub with the device identity created earlier.
- Receives a reboot direct method call.
- Simulates a physical reboot.
- Reports the time of the last reboot through a reported property.

trigger-reboot. This app:

- Calls a direct method in the simulated device app.
- Displays the response to the direct method call sent by the simulated device.
- Displays the updated reported properties.

NOTE

For information about the SDKs that you can use to build applications to run on devices and your solution back end, see [Azure IoT SDKs](#).

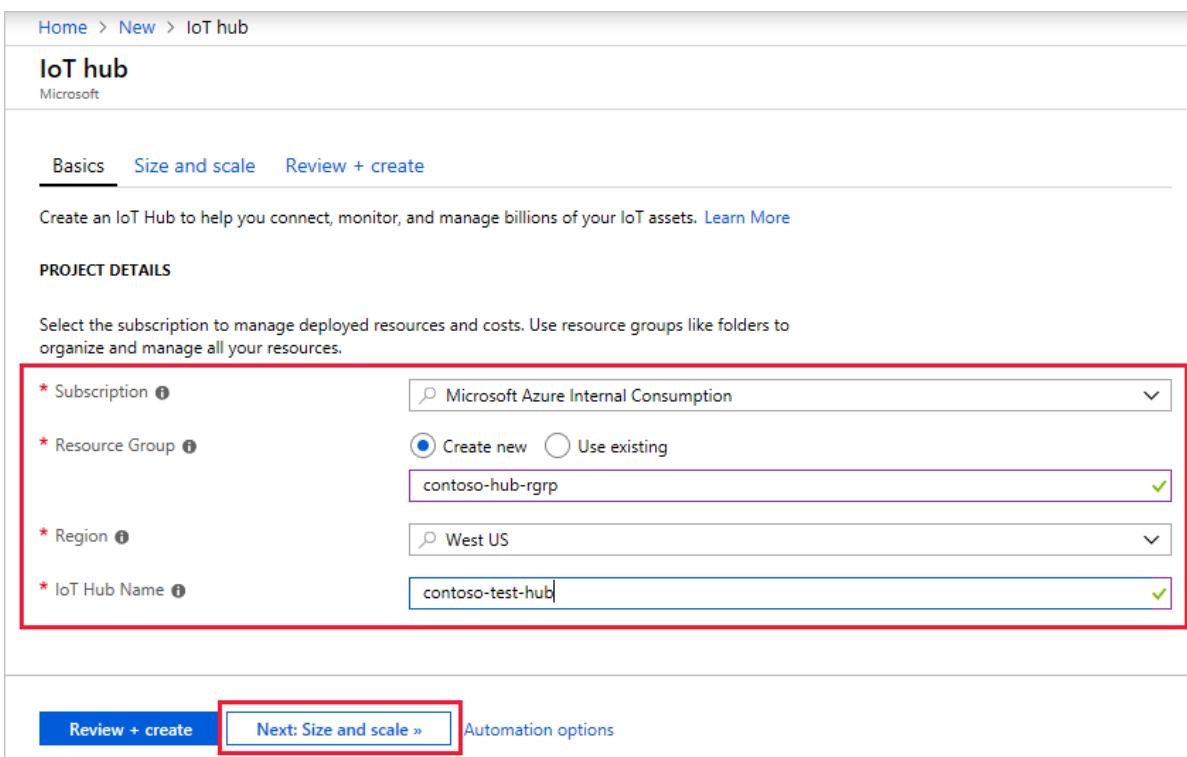
To complete this tutorial, you need:

- Java SE 8.
[Prepare your development environment](#) describes how to install Java for this tutorial on either Windows or Linux.
- Maven 3.
[Prepare your development environment](#) describes how to install [Maven](#) for this tutorial on either Windows or Linux.
- [Node.js version 0.10.0 or later](#).

Create an IoT hub

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose +**Create a resource**, then choose **Internet of Things**.
3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.



Fill in the fields.

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

The screenshot shows the 'Size and scale' tab of the IoT hub creation wizard. It includes sections for selecting a tier (S1: Standard tier), specifying the number of units (1 unit selected), and enabling various features like Device-to-cloud-messages, Message routing, Cloud-to-device commands, IoT Edge, and Device management. Advanced settings for device-to-cloud partitions are also shown, currently set to 4. Navigation buttons at the bottom include 'Review + create', '<< Previous: Basics', and 'Automation options'.

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: The number of messages allowed per unit per day depends on your hub's pricing tier. For example, if you want the IoT hub to support ingress of 700,000 messages, you choose two S1 tier units.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

Advanced / Device-to-cloud partitions: This property relates the device-to-cloud messages to the

number of simultaneous readers of the messages. Most IoT hubs only need four partitions.

5. Click **Review + create** to review your choices. You see something similar to this screen.

The screenshot shows the 'Review + create' step of creating an IoT hub in the Azure portal. At the top, there's a breadcrumb navigation: Home > New > IoT hub. Below it, the title 'IoT hub' and the Microsoft logo. A navigation bar has three tabs: Basics, Size and scale, and Review + create (which is highlighted with a red box). Under 'Basics', there are four sections: Subscription (Microsoft Azure Internal Consumption), Resource Group (contoso-hub-rgrp), Region (West US), and IoT Hub Name (contoso-test-hub). Under 'SIZE AND SCALE', there are four sections: Pricing and scale tier (S1), Number of S1 IoT Hub units (1), Messages per day (400,000), and Cost per month (25.00 USD). At the bottom, there are three buttons: 'Create' (highlighted with a red box), '< Previous: Size and scale', and 'Automation options'.

6. Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Retrieve connection string for IoT hub

After your hub has been created, retrieve the connection string for the hub. This is used to connect devices and applications to your hub.

1. Click on your hub to see the IoT Hub pane with Settings, and so on. Click **Shared access policies**.
2. In **Shared access policies**, select the **iothubowner** policy.
3. Under **Shared access keys**, copy the **Connection string -- primary key** to be used later.

The screenshot shows the 'Shared access policies' section of the Azure IoT Hub 'ContosoHub' settings. On the left, a sidebar lists various settings like Overview, Activity log, Access control (IAM), Tags, Events, and Shared access policies (which is selected). The main pane displays the 'iothubowner' policy details. The policy grants 'Registry read', 'Registry write', 'Service connect', and 'Device connect' permissions. Under 'Shared access keys', the primary key is shown as a connection string: `HostName=ContosoHub.azure-devices.net;SharedAccessKey=...`. This connection string is highlighted with a red rectangle.

For more information, see [Access control](#) in the "IoT Hub developer guide."

Create a device identity

In this section, you use the Azure CLI to create a device identity for this tutorial. The Azure CLI is preinstalled in the [Azure Cloud Shell](#), or you can [install it locally](#). Device IDs are case sensitive.

1. Run the following command in the command-line environment where you are using the Azure CLI to install the IoT extension:

```
az extension add --name azure-cli-iot-ext
```

2. If you are running the Azure CLI locally, use the following command to sign in to your Azure account (if you are using the Cloud Shell, you are signed in automatically and you don't need to run this command):

```
az login
```

3. Finally, create a new device identity called `myDeviceId` and retrieve the device connection string with these commands:

```
az iot hub device-identity create --device-id myDeviceId --hub-name {Your IoT Hub name}  
az iot hub device-identity show-connection-string --device-id myDeviceId --hub-name {Your IoT Hub name}  
-o table
```

IMPORTANT

The device ID may be visible in the logs collected for customer support and troubleshooting, so make sure to avoid any sensitive information while naming it.

Make a note of the device connection string from the result. This device connection string is used by the device app to connect to your IoT Hub as a device.

Trigger a remote reboot on the device using a direct method

In this section, you create a Java console app that:

1. Invokes the reboot direct method in the simulated device app.
2. Displays the response.
3. Polls the reported properties sent from the device to determine when the reboot is complete.

This console app connects to your IoT Hub to invoke the direct method and read the reported properties.

1. Create an empty folder called dm-get-started.
2. In the dm-get-started folder, create a Maven project called **trigger-reboot** using the following command at your command prompt. The following shows a single, long command:

```
mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=trigger-reboot -  
DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

3. At your command prompt, navigate to the trigger-reboot folder.
4. Using a text editor, open the pom.xml file in the trigger-reboot folder and add the following dependency to the **dependencies** node. This dependency enables you to use the iot-service-client package in your app to communicate with your IoT hub:

```
<dependency>  
  <groupId>com.microsoft.azure.sdk.iot</groupId>  
  <artifactId>iot-service-client</artifactId>  
  <version>1.7.23</version>  
  <type>jar</type>  
</dependency>
```

NOTE

You can check for the latest version of **iot-service-client** using [Maven search](#).

5. Add the following **build** node after the **dependencies** node. This configuration instructs Maven to use Java 1.8 to build the app:

```
<build>  
  <plugins>  
    <plugin>  
      <groupId>org.apache.maven.plugins</groupId>  
      <artifactId>maven-compiler-plugin</artifactId>  
      <version>3.3</version>  
      <configuration>  
        <source>1.8</source>  
        <target>1.8</target>  
      </configuration>  
    </plugin>  
  </plugins>  
</build>
```

6. Save and close the pom.xml file.
7. Using a text editor, open the trigger-reboot\src\main\java\com\mycompany\app\App.java source file.
8. Add the following **import** statements to the file:

```
import com.microsoft.azure.sdk.iot.service.devicetwin.DeviceMethod;
import com.microsoft.azure.sdk.iot.service.devicetwin.MethodResult;
import com.microsoft.azure.sdk.iot.service.exceptions.IotHubException;
import com.microsoft.azure.sdk.iot.service.devicetwin.DeviceTwin;
import com.microsoft.azure.sdk.iot.service.devicetwin.DeviceTwinDevice;

import java.io.IOException;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
```

9. Add the following class-level variables to the **App** class. Replace `{youriothubconnectionstring}` with your IoT hub connection string you noted in the *Create an IoT Hub* section:

```
public static final String iotHubConnectionString = "{youriothubconnectionstring}";
public static final String deviceId = "myDeviceId";

private static final String methodName = "reboot";
private static final Long responseTimeout = TimeUnit.SECONDS.toSeconds(30);
private static final Long connectTimeout = TimeUnit.SECONDS.toSeconds(5);
```

10. To implement a thread that reads the reported properties from the device twin every 10 seconds, add the following nested class to the **App** class:

```
private static class ShowReportedProperties implements Runnable {
    public void run() {
        try {
            DeviceTwin deviceTwins = DeviceTwin.createFromConnectionString(iotHubConnectionString);
            DeviceTwinDevice twinDevice = new DeviceTwinDevice(deviceId);
            while (true) {
                System.out.println("Get reported properties from device twin");
                deviceTwins.getTwin(twinDevice);
                System.out.println(twinDevice.reportedPropertiesToString());
                Thread.sleep(10000);
            }
        } catch (Exception ex) {
            System.out.println("Exception reading reported properties: " + ex.getMessage());
        }
    }
}
```

11. Modify the signature of the **main** method to throw the following exception:

```
public static void main(String[] args) throws IOException
```

12. To invoke the reboot direct method on the simulated device, add the following code to the **main** method:

```

System.out.println("Starting sample...");
DeviceMethod methodClient = DeviceMethod.createFromConnectionString(iotHubConnectionString);

try
{
    System.out.println("Invoke reboot direct method");
    MethodResult result = methodClient.invoke(deviceId, methodName, responseTimeout, connectTimeout,
null);

    if(result == null)
    {
        throw new IOException("Invoke direct method reboot returns null");
    }
    System.out.println("Invoked reboot on device");
    System.out.println("Status for device: " + result.getStatus());
    System.out.println("Message from device: " + result.getPayload());
}
catch (IoTException e)
{
    System.out.println(e.getMessage());
}

```

13. To start the thread to poll the reported properties from the simulated device, add the following code to the **main** method:

```

ShowReportedProperties showReportedProperties = new ShowReportedProperties();
ExecutorService executor = Executors.newFixedThreadPool(1);
executor.execute(showReportedProperties);

```

14. To enable you to stop the app, add the following code to the **main** method:

```

System.out.println("Press ENTER to exit.");
System.in.read();
executor.shutdownNow();
System.out.println("Shutting down sample...");

```

15. Save and close the trigger-reboot\src\main\java\com\mycompany\app\App.java file.

16. Build the **trigger-reboot** back-end app and correct any errors. At your command prompt, navigate to the trigger-reboot folder and run the following command:

```
mvn clean package -DskipTests
```

Create a simulated device app

In this section, you create a Java console app that simulates a device. The app listens for the reboot direct method call from your IoT hub and immediately responds to that call. The app then sleeps for a while to simulate the reboot process before it uses a reported property to notify the **trigger-reboot** back-end app that the reboot is complete.

1. In the dm-get-started folder, create a Maven project called **simulated-device** using the following command at your command prompt. The following is a single, long command:

```
mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=simulated-device -
DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

2. At your command prompt, navigate to the simulated-device folder.
3. Using a text editor, open the pom.xml file in the simulated-device folder and add the following dependency to the **dependencies** node. This dependency enables you to use the iot-service-client package in your app

to communicate with your IoT hub:

```
<dependency>
  <groupId>com.microsoft.azure.sdk.iot</groupId>
  <artifactId>iot-device-client</artifactId>
  <version>1.3.32</version>
</dependency>
```

NOTE

You can check for the latest version of **iot-device-client** using [Maven search](#).

4. Add the following **build** node after the **dependencies** node. This configuration instructs Maven to use Java 1.8 to build the app:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.3</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

5. Save and close the pom.xml file.
6. Using a text editor, open the simulated-device\src\main\java\com\mycompany\app\App.java source file.
7. Add the following **import** statements to the file:

```
import com.microsoft.azure.sdk.iot.device.*;
import com.microsoft.azure.sdk.iot.device.DeviceTwin.*;

import java.io.IOException;
import java.net.URISyntaxException;
import java.time.LocalDateTime;
import java.util.Scanner;
import java.util.Set;
import java.util.HashSet;
```

8. Add the following class-level variables to the **App** class. Replace `{yourdeviceconnectionstring}` with the device connection string you noted in the *Create a device identity* section:

```
private static final int METHOD_SUCCESS = 200;
private static final int METHOD_NOT_DEFINED = 404;

private static IoTHubClientProtocol protocol = IoTHubClientProtocol.MQTT;
private static String connString = "{yourdeviceconnectionstring}";
private static DeviceClient client;
```

9. To implement a callback handler for direct method status events, add the following nested class to the **App** class:

```

protected static class DirectMethodStatusCallback implements IoTHubEventCallback
{
    public void execute(IotHubStatusCode status, Object context)
    {
        System.out.println("IoT Hub responded to device method operation with status " + status.name());
    }
}

```

10. To implement a callback handler for device twin status events, add the following nested class to the **App** class:

```

protected static class DeviceTwinStatusCallback implements IoTHubEventCallback
{
    public void execute(IotHubStatusCode status, Object context)
    {
        System.out.println("IoT Hub responded to device twin operation with status " + status.name());
    }
}

```

11. To implement a callback handler for property events, add the following nested class to the **App** class:

```

protected static class PropertyCallback implements PropertyCallBack<String, String>
{
    public void PropertyCall(String propertyKey, String propertyValue, Object context)
    {
        System.out.println("PropertyKey:      " + propertyKey);
        System.out.println("PropertyKvalue:  " + propertyKey);
    }
}

```

12. To implement a thread to simulate the device reboot, add the following nested class to the **App** class. The thread sleeps for five seconds and then sets the **lastReboot** reported property:

```

protected static class RebootDeviceThread implements Runnable {
    public void run() {
        try {
            System.out.println("Rebooting...");
            Thread.sleep(5000);
            Property property = new Property("lastReboot", LocalDateTime.now());
            Set<Property> properties = new HashSet<Property>();
            properties.add(property);
            client.sendReportedProperties(properties);
            System.out.println("Rebooted");
        }
        catch (Exception ex) {
            System.out.println("Exception in reboot thread: " + ex.getMessage());
        }
    }
}

```

13. To implement the direct method on the device, add the following nested class to the **App** class. When the simulated app receives a call to the **reboot** direct method, it returns an acknowledgement to the caller and then starts a thread to process the reboot:

```

protected static class DirectMethodCallback implements
com.microsoft.azure.sdk.iot.device.DeviceTwin.DeviceMethodCallback
{
    @Override
    public DeviceMethodData call(String methodName, Object methodData, Object context)
    {
        DeviceMethodData deviceMethodData;
        switch (methodName)
        {
            case "reboot" :
            {
                int status = METHOD_SUCCESS;
                System.out.println("Received reboot request");
                deviceMethodData = new DeviceMethodData(status, "Started reboot");
                RebootDeviceThread rebootThread = new RebootDeviceThread();
                Thread t = new Thread(rebootThread);
                t.start();
                break;
            }
            default:
            {
                int status = METHOD_NOT_DEFINED;
                deviceMethodData = new DeviceMethodData(status, "Not defined direct method " + methodName);
            }
        }
        return deviceMethodData;
    }
}

```

14. Modify the signature of the **main** method to throw the following exceptions:

```
public static void main(String[] args) throws IOException, URISyntaxException
```

15. To instantiate a **DeviceClient**, add the following code to the **main** method:

```
System.out.println("Starting device client sample...");
client = new DeviceClient(connString, protocol);
```

16. To start listening for direct method calls, add the following code to the **main** method:

```

try
{
    client.open();
    client.subscribeToDeviceMethod(new DirectMethodCallback(), null, new DirectMethodStatusCallback(),
null);
    client.startDeviceTwin(new DeviceTwinStatusCallback(), null, new PropertyCallback(), null);
    System.out.println("Subscribed to direct methods and polling for reported properties. Waiting...");
}
catch (Exception e)
{
    System.out.println("On exception, shutting down \n" + " Cause: " + e.getCause() + " \n" +
e.getMessage());
    client.close();
    System.out.println("Shutting down...");
}
```

17. To shut down the device simulator, add the following code to the **main** method:

```
System.out.println("Press any key to exit...");
Scanner scanner = new Scanner(System.in);
scanner.nextLine();
scanner.close();
client.close();
System.out.println("Shutting down...");
```

18. Save and close the simulated-device\src\main\java\com\mycompany\app\App.java file.
19. Build the **simulated-device** back-end app and correct any errors. At your command prompt, navigate to the simulated-device folder and run the following command:

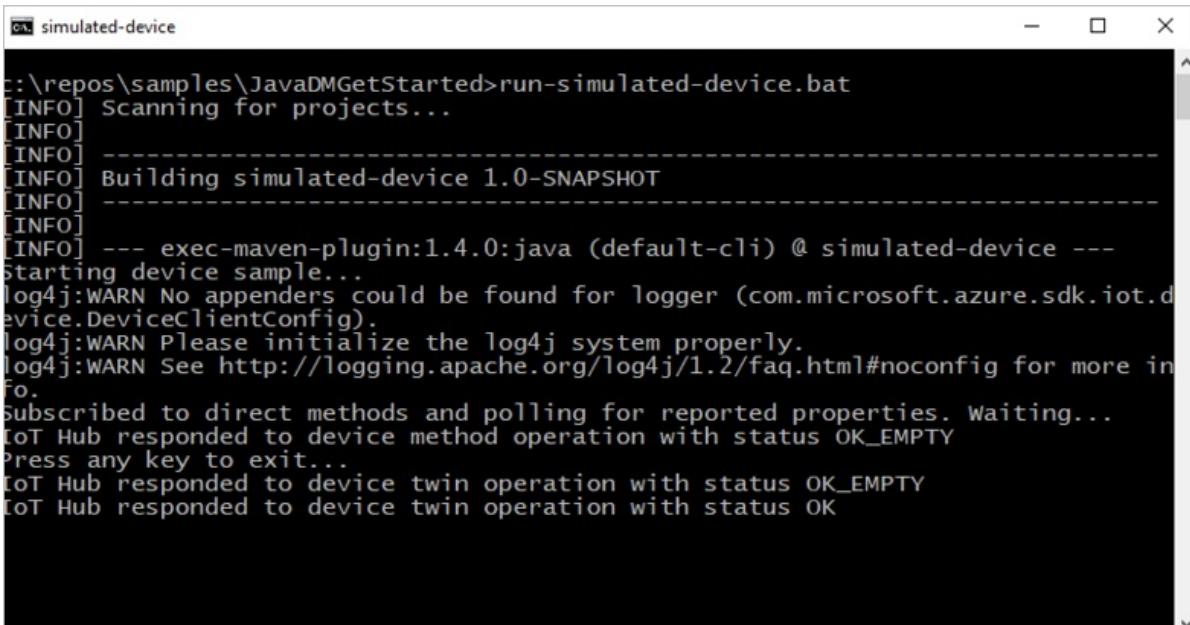
```
mvn clean package -DskipTests
```

Run the apps

You are now ready to run the apps.

1. At a command prompt in the simulated-device folder, run the following command to begin listening for reboot method calls from your IoT hub:

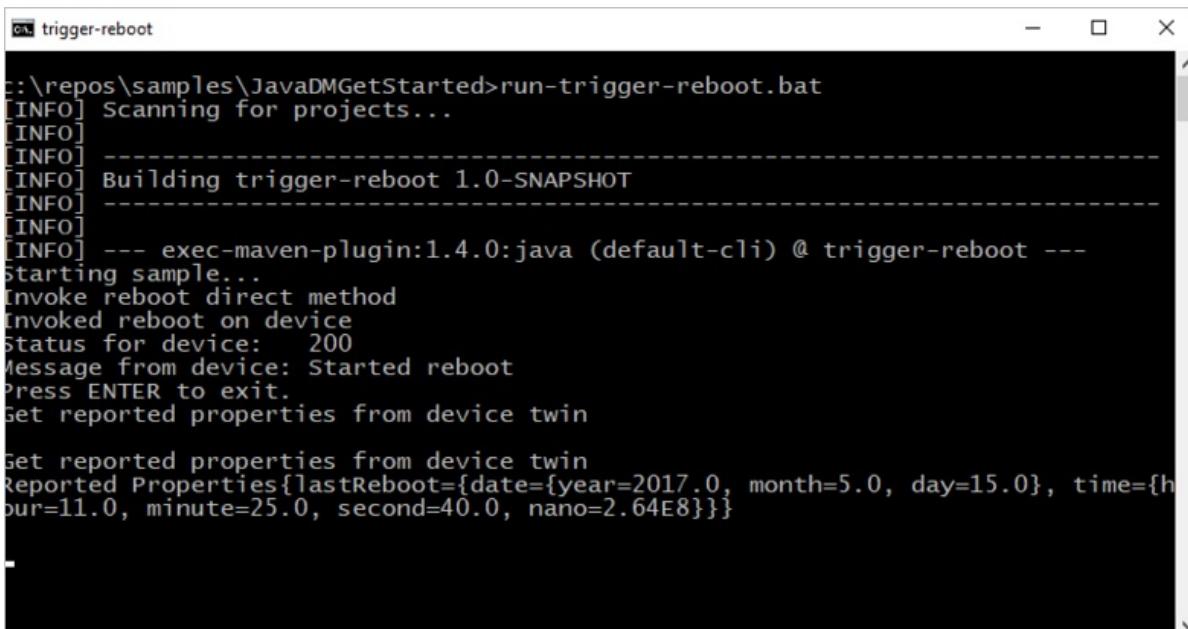
```
mvn exec:java -Dexec.mainClass="com.mycompany.app.App"
```



```
simulated-device
C:\repos\samples\JavaDMGetStarted>run-simulated-device.bat
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building simulated-device 1.0-SNAPSHOT
[INFO] -----
[INFO] --- exec-maven-plugin:1.4.0:java (default-cli) @ simulated-device ---
Starting device sample...
log4j:WARN No appenders could be found for logger (com.microsoft.azure.sdk.iot.device.DeviceClientConfig).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
Subscribed to direct methods and polling for reported properties. Waiting...
IoT Hub responded to device method operation with status OK_EMPTY
Press any key to exit...
IoT Hub responded to device twin operation with status OK_EMPTY
IoT Hub responded to device twin operation with status OK
```

2. At a command prompt in the trigger-reboot folder, run the following command to call the reboot method on your simulated device from your IoT hub:

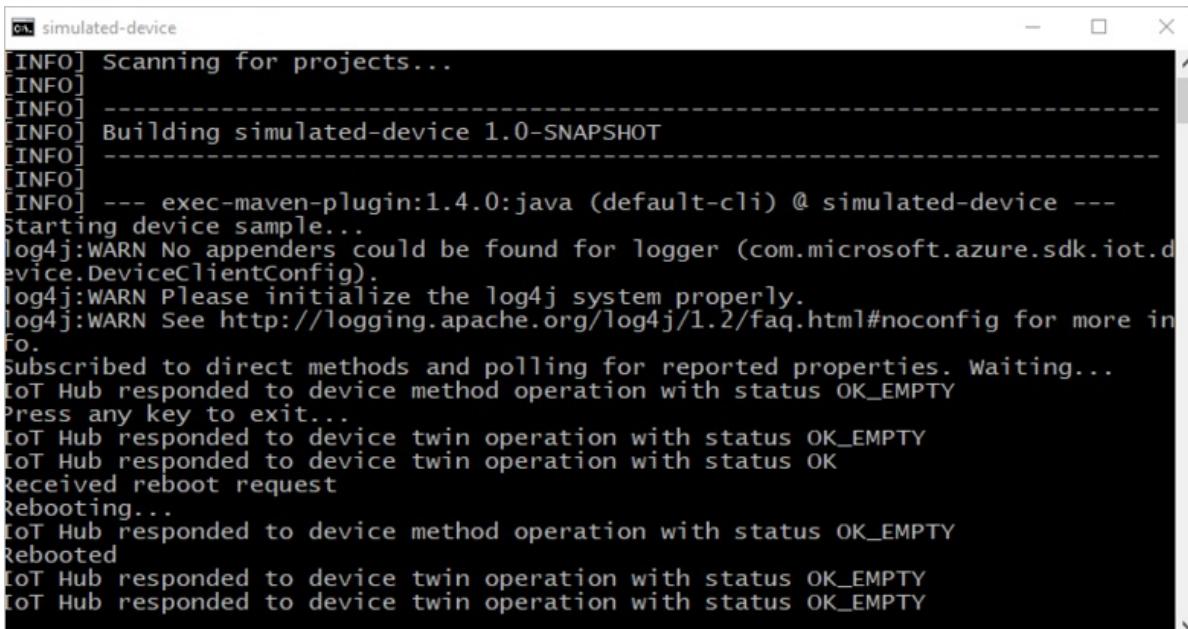
```
mvn exec:java -Dexec.mainClass="com.mycompany.app.App"
```



```
c:\repos\samples\JavaDMGetStarted>run-trigger-reboot.bat
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building trigger-reboot 1.0-SNAPSHOT
[INFO] -----
[INFO]
[INFO] --- exec-maven-plugin:1.4.0:java (default-cli) @ trigger-reboot ---
Starting sample...
Invoke reboot direct method
Invoked reboot on device
Status for device: 200
Message from device: Started reboot
Press ENTER to exit.
Get reported properties from device twin

Get reported properties from device twin
Reported Properties{lastReboot={date={year=2017.0, month=5.0, day=15.0}, time={hour=11.0, minute=25.0, second=40.0, nano=2.64E8}}}
```

3. The simulated device responds to the reboot direct method call:



```
c:\repos\samples\JavaDMGetStarted>simulated-device
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building simulated-device 1.0-SNAPSHOT
[INFO] -----
[INFO]
[INFO] --- exec-maven-plugin:1.4.0:java (default-cli) @ simulated-device ---
Starting device sample...
log4j:WARN No appenders could be found for logger (com.microsoft.azure.sdk.iot.device.DeviceClientConfig).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
Subscribed to direct methods and polling for reported properties. Waiting...
IoT Hub responded to device method operation with status OK_EMPTY
Press any key to exit...
IoT Hub responded to device twin operation with status OK_EMPTY
IoT Hub responded to device twin operation with status OK
Received reboot request
Rebooting...
IoT Hub responded to device method operation with status OK_EMPTY
Rebooted
IoT Hub responded to device twin operation with status OK_EMPTY
IoT Hub responded to device twin operation with status OK_EMPTY
```

Customize and extend the device management actions

Your IoT solutions can expand the defined set of device management patterns or enable custom patterns by using the device twin and cloud-to-device method primitives. Other examples of device management actions include factory reset, firmware update, software update, power management, network and connectivity management, and data encryption.

Device maintenance windows

Typically, you configure devices to perform actions at a time that minimizes interruptions and downtime. Device maintenance windows are a commonly used pattern to define the time when a device should update its configuration. Your back-end solutions can use the desired properties of the device twin to define and activate a policy on your device that enables a maintenance window. When a device receives the maintenance window policy, it can use the reported property of the device twin to report the status of the policy. The back-end app can then use device twin queries to attest to compliance of devices and each policy.

Next steps

In this tutorial, you used a direct method to trigger a remote reboot on a device. You used the reported properties to report the last reboot time from the device, and queried the device twin to discover the last reboot time of the device from the cloud.

To continue getting started with IoT Hub and device management patterns such as remote over the air firmware update, see [How to do a firmware update](#)

To learn how to extend your IoT solution and schedule method calls on multiple devices, see [Schedule and broadcast jobs](#).

Get started with device management (Python)

3/6/2019 • 9 minutes to read

Back-end apps can use Azure IoT Hub primitives, such as [device twin](#) and [direct methods](#), to remotely start and monitor device management actions on devices. This tutorial shows you how a back-end app and a device app can work together to initiate and monitor a remote device reboot using IoT Hub.

NOTE

The features described in this article are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

Use a direct method to initiate device management actions (such as reboot, factory reset, and firmware update) from a back-end app in the cloud. The device is responsible for:

- Handling the method request sent from IoT Hub.
- Initiating the corresponding device-specific action on the device.
- Providing status updates through *reported properties* to IoT Hub.

You can use a back-end app in the cloud to run device twin queries to report on the progress of your device management actions.

This tutorial shows you how to:

- Use the Azure portal to create an IoT Hub and create a device identity in your IoT hub.
- Create a simulated device app that contains a direct method that reboots that device. Direct methods are invoked from the cloud.
- Create a Python console app that calls the reboot direct method in the simulated device app through your IoT hub.

At the end of this tutorial, you have two Python console apps:

dmpatterns_getstarted_device.py, which connects to your IoT hub with the device identity created earlier, receives a reboot direct method, simulates a physical reboot, and reports the time for the last reboot.

dmpatterns_getstarted_service.py, which calls a direct method in the simulated device app, displays the response, and displays the updated reported properties.

To complete this tutorial, you need the following:

- [Python 2.x or 3.x](#). Make sure to use the 32-bit or 64-bit installation as required by your setup. When prompted during the installation, make sure to add Python to your platform-specific environment variable. If you are using Python 2.x, you may need to [install or upgrade pip, the Python package management system](#).
 - Install the [azure-iothub-device-client](#) package, using the command

```
pip install azure-iothub-device-client
```
 - Install the [azure-iothub-service-client](#) package, using the command

```
pip install azure-iothub-service-client
```
- If you are using Windows OS, then [Visual C++ redistributable package](#) to allow the use of native DLLs from Python.
- An active Azure account. (If you don't have an account, you can create a [free account](#) in just a couple of minutes.)

Create an IoT hub

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose +**Create a resource**, then choose **Internet of Things**.
3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Review + create

Create an IoT Hub to help you connect, monitor, and manage billions of your IoT assets. [Learn More](#)

PROJECT DETAILS

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

* Subscription [?](#) Microsoft Azure Internal Consumption

* Resource Group [?](#) Create new Use existing [contoso-hub-rgrp](#)

* Region [?](#) West US

* IoT Hub Name [?](#) contoso-test-hub

[Review + create](#) [Next: Size and scale >](#) Automation options

Fill in the fields.

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics **Size and scale** Review + create

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS

* Pricing and scale tier [?](#) S1: Standard tier [?](#) [Learn how to choose the right IoT Hub tier for your solution](#)

Number of S1 IoT Hub units [?](#) 1 [1](#) This determines your IoT Hub scale capability and can be changed as your need increases.

Device-to-cloud-messages [?](#) Enabled

Message routing [?](#) Enabled

Cloud-to-device commands [?](#) Enabled

IoT Edge [?](#) Enabled

Device management [?](#) Enabled

Advanced Settings

Device-to-cloud partitions [?](#) 4 [4](#)

Review + create [« Previous: Basics](#) [Automation options](#)

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: The number of messages allowed per unit per day depends on your hub's pricing tier. For example, if you want the IoT hub to support ingress of 700,000 messages, you choose two S1 tier units.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

Advanced / Device-to-cloud partitions: This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most IoT hubs only need four partitions.

5. Click **Review + create** to review your choices. You see something similar to this screen.

Basics

Subscription: Microsoft Azure Internal Consumption
Resource Group: contoso-hub-rgrp
Region: West US
IoT Hub Name: contoso-test-hub

SIZE AND SCALE

Pricing and scale tier: S1
Number of S1 IoT Hub units: 1
Messages per day: 400,000
Cost per month: 25.00 USD

Create

6. Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Retrieve connection string for IoT hub

After your hub has been created, retrieve the connection string for the hub. This is used to connect devices and applications to your hub.

1. Click on your hub to see the IoT Hub pane with Settings, and so on. Click **Shared access policies**.
2. In **Shared access policies**, select the **iothubowner** policy.
3. Under **Shared access keys**, copy the **Connection string -- primary key** to be used later.

ContosoHub - Shared access policies

iothubowner

Access policy name: iothubowner

Permissions:

- Registry read
- Registry write
- Service connect
- Device connect

Shared access keys:

Primary key:	HostName=ContosoHub.azure-devices.net;SharedAccessKey...
Secondary key:	HostName=ContosoHub.azure-devices.net;SharedAccessKey...

Connection string—primary key:

HostName=ContosoHub.azure-devices.net;SharedAccessKey...

For more information, see [Access control](#) in the "IoT Hub developer guide."

Create a simulated device app

In this section, you will:

- Create a Python console app that responds to a direct method called by the cloud
 - Simulate a device reboot
 - Use the reported properties to enable device twin queries to identify devices and when they last rebooted
1. Using a text editor, create a **dmpatterns_getstarted_device.py** file.
 2. Add the following `import` statements at the start of the **dmpatterns_getstarted_device.py** file.

```
import random
import time, datetime
import sys

import iothub_client
from iothub_client import IoTHubClient, IoTHubClientError, IoTHubTransportProvider, IoTHubClientResult,
IoTHubError, DeviceMethodReturnValue
```

3. Add variables including a **CONNECTION_STRING** variable and the client initialization. Replace the connection string with your device connection string.

```
CONNECTION_STRING = "{deviceConnectionString}"
PROTOCOL = IoTHubTransportProvider.MQTT

CLIENT = IoTHubClient(CONNECTION_STRING, PROTOCOL)

WAIT_COUNT = 5

SEND_REPORTED_STATE_CONTEXT = 0
METHOD_CONTEXT = 0

SEND_REPORTED_STATE_CALLBACKS = 0
METHOD_CALLBACKS = 0
```

4. Add the following function callbacks to implement the direct method on the device.

```

def send_reported_state_callback(status_code, user_context):
    global SEND_REPORTED_STATE_CALLBACKS

    print ( "Device twins updated." )

def device_method_callback(method_name, payload, user_context):
    global METHOD_CALLBACKS

    if method_name == "rebootDevice":
        print ( "Rebooting device..." )

        time.sleep(20)

        print ( "Device rebooted." )

        current_time = str(datetime.datetime.now())
        reported_state = "{\"rebootTime\":\"" + current_time + "\"}"
        CLIENT.send_reported_state(reported_state, len(reported_state), send_reported_state_callback,
SEND_REPORTED_STATE_CONTEXT)

        print ( "Updating device twins: rebootTime" )

    device_method_return_value = DeviceMethodReturnValue()
    device_method_return_value.response = "{ \"Response\": \"This is the response from the device\" }"
    device_method_return_value.status = 200

    return device_method_return_value

```

5. Start the direct method listener and wait.

```

def iothub_client_init():
    if CLIENT.protocol == IoTHubTransportProvider.MQTT or client.protocol ==
IoTHubTransportProvider.MQTT_WS:
        CLIENT.set_device_method_callback(device_method_callback, METHOD_CONTEXT)

def iothub_client_sample_run():
    try:
        iothub_client_init()

        while True:
            print ( "IoTHubClient waiting for commands, press Ctrl-C to exit" )

            status_counter = 0
            while status_counter <= WAIT_COUNT:
                time.sleep(10)
                status_counter += 1

    except IoTHubError as iothub_error:
        print ( "Unexpected error %s from IoTHub" % iothub_error )
        return
    except KeyboardInterrupt:
        print ( "IoTHubClient sample stopped" )

if __name__ == '__main__':
    print ( "Starting the IoT Hub Python sample..." )
    print ( "    Protocol %s" % PROTOCOL )
    print ( "    Connection string=%s" % CONNECTION_STRING )

    iothub_client_sample_run()

```

6. Save and close the **dmpatterns_getstarted_device.py** file.

NOTE

To keep things simple, this tutorial does not implement any retry policy. In production code, you should implement retry policies (such as an exponential backoff), as suggested in the article, [Transient Fault Handling](#).

Trigger a remote reboot on the device using a direct method

In this section, you create a Python console app that initiates a remote reboot on a device using a direct method. The app uses device twin queries to discover the last reboot time for that device.

1. Using a text editor, create a **dmpatterns_getstarted_service.py** file.
2. Add the following `import` statements at the start of the **dmpatterns_getstarted_service.py** file.

```
import sys, time
import iothub_service_client

from iothub_service_client import IoTHubDeviceMethod, IoTHubError, IoTHubDeviceTwin
```

3. Add the following variable declarations. Only replace placeholder values for *IoTHubConnectionString* and *deviceId*.

```
CONNECTION_STRING = "{IoTHubConnectionString}"
DEVICE_ID = "{deviceId}"

METHOD_NAME = "rebootDevice"
METHOD_PAYLOAD = "{\"method_number\": \"42\"}"
TIMEOUT = 60
WAIT_COUNT = 10
```

4. Add the following function to invoke the device method to reboot the target device, then query for the device twins and get the last reboot time.

```

def iothub_devicemethod_sample_run():
    try:
        iothub_twin_method = IoTHubDeviceTwin(CONNECTION_STRING)
        iothub_device_method = IoTHubDeviceMethod(CONNECTION_STRING)

        print( "" )
        print( "Invoking device to reboot..." )

        response = iothub_device_method.invoke(DEVICE_ID, METHOD_NAME, METHOD_PAYLOAD, TIMEOUT)

        print( "" )
        print( "Successfully invoked the device to reboot." )

        print( "" )
        print( response.payload )

    while True:
        print( "" )
        print( "IoTHubClient waiting for commands, press Ctrl-C to exit" )

        status_counter = 0
        while status_counter <= WAIT_COUNT:
            twin_info = iothub_twin_method.get_twin(DEVICE_ID)

            if twin_info.find("rebootTime") != -1:
                print( "Last reboot time: " +
twin_info[twin_info.find("rebootTime")+11:twin_info.find("rebootTime")+37])
            else:
                print( "Waiting for device to report last reboot time..." )

            time.sleep(5)
            status_counter += 1

    except IoTHubError as iothub_error:
        print( "" )
        print( "Unexpected error {0}".format(iothub_error) )
        return
    except KeyboardInterrupt:
        print( "" )
        print( "IoTHubDeviceMethod sample stopped" )

if __name__ == '__main__':
    print( "Starting the IoT Hub Service Client DeviceManagement Python sample..." )
    print( "    Connection string = {0}".format(CONNECTION_STRING) )
    print( "    Device ID          = {0}".format(DEVICE_ID) )

    iothub_devicemethod_sample_run()

```

5. Save and close the **dmpatterns_getstarted_service.py** file.

Run the apps

You are now ready to run the apps.

- At the command prompt, run the following command to begin listening for the reboot direct method.

```
python dmpatterns_getstarted_device.py
```

- At another command prompt, run the following command to trigger the remote reboot and query for the device twin to find the last reboot time.

```
python dmpatterns_getstarted_service.py
```

3. You see the device response to the direct method in the console.

Customize and extend the device management actions

Your IoT solutions can expand the defined set of device management patterns or enable custom patterns by using the device twin and cloud-to-device method primitives. Other examples of device management actions include factory reset, firmware update, software update, power management, network and connectivity management, and data encryption.

Device maintenance windows

Typically, you configure devices to perform actions at a time that minimizes interruptions and downtime. Device maintenance windows are a commonly used pattern to define the time when a device should update its configuration. Your back-end solutions can use the desired properties of the device twin to define and activate a policy on your device that enables a maintenance window. When a device receives the maintenance window policy, it can use the reported property of the device twin to report the status of the policy. The back-end app can then use device twin queries to attest to compliance of devices and each policy.

Next steps

In this tutorial, you used a direct method to trigger a remote reboot on a device. You used the reported properties to report the last reboot time from the device, and queried the device twin to discover the last reboot time of the device from the cloud.

To continue getting started with IoT Hub and device management patterns such as remote over the air firmware update, see [How to do a firmware update](#)

To learn how to extend your IoT solution and schedule method calls on multiple devices, see [Schedule and broadcast jobs](#).

Schedule and broadcast jobs (Node)

3/6/2019 • 9 minutes to read

Azure IoT Hub is a fully managed service that enables a back-end app to create and track jobs that schedule and update millions of devices. Jobs can be used for the following actions:

- Update desired properties
- Update tags
- Invoke direct methods

Conceptually, a job wraps one of these actions and tracks the progress of execution against a set of devices, which is defined by a device twin query. For example, a back-end app can use a job to invoke a reboot method on 10,000 devices, specified by a device twin query and scheduled at a future time. That application can then track progress as each of those devices receive and execute the reboot method.

Learn more about each of these capabilities in these articles:

- Device twin and properties: [Get started with device twins](#) and [Tutorial: How to use device twin properties](#)
- Direct methods: [IoT Hub developer guide - direct methods](#) and [Tutorial: direct methods](#)

NOTE

The features described in this article are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

This tutorial shows you how to:

- Create a Node.js simulated device app that has a direct method, which enables **lockDoor**, which can be called by the solution back end.
- Create a Node.js console app that calls the **lockDoor** direct method in the simulated device app using a job and updates the desired properties using a device job.

At the end of this tutorial, you have two Node.js apps:

simDevice.js, which connects to your IoT hub with the device identity and receives a **lockDoor** direct method.

scheduleJobService.js, which calls a direct method in the simulated device app and updates the device twin's desired properties using a job.

To complete this tutorial, you need the following:

- Node.js version 4.0.x or later,
[Prepare your development environment](#) describes how to install Node.js for this tutorial on either Windows or Linux.
- An active Azure account. (If you don't have an account, you can create a [free account](#) in just a couple of minutes.)

Create an IoT hub

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).

2. Choose +**Create a resource**, then choose **Internet of Things**.
3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Review + create

Create an IoT Hub to help you connect, monitor, and manage billions of your IoT assets. [Learn More](#)

PROJECT DETAILS

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

* Subscription: Microsoft Azure Internal Consumption

* Resource Group: contoso-hub-rgrp

* Region: West US

* IoT Hub Name: contoso-test-hub

Review + create **Next: Size and scale >** Automation options

Fill in the fields.

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics **Size and scale** Review + create

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS

* Pricing and scale tier [?](#) S1: Standard tier [▼](#) [Learn how to choose the right IoT Hub tier for your solution](#)

Number of S1 IoT Hub units [?](#) 1 This determines your IoT Hub scale capability and can be changed as your need increases.

Device-to-cloud-messages [?](#) Enabled

Message routing [?](#) Enabled

Cloud-to-device commands [?](#) Enabled

IoT Edge [?](#) Enabled

Device management [?](#) Enabled

Advanced Settings

Device-to-cloud partitions [?](#) 4

Review + create [« Previous: Basics](#) [Automation options](#)

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: The number of messages allowed per unit per day depends on your hub's pricing tier. For example, if you want the IoT hub to support ingress of 700,000 messages, you choose two S1 tier units.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

Advanced / Device-to-cloud partitions: This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most IoT hubs only need four partitions.

5. Click **Review + create** to review your choices. You see something similar to this screen.

The screenshot shows the 'Review + create' step of the IoT hub creation wizard. It displays basic information like subscription, resource group, region, and IoT Hub name, as well as size and scale details including pricing tier, number of units, messages per day, and cost per month. At the bottom, there are 'Create' and 'Automation options' buttons.

6. Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Retrieve connection string for IoT hub

After your hub has been created, retrieve the connection string for the hub. This is used to connect devices and applications to your hub.

1. Click on your hub to see the IoT Hub pane with Settings, and so on. Click **Shared access policies**.
2. In **Shared access policies**, select the **iothubowner** policy.
3. Under **Shared access keys**, copy the **Connection string -- primary key** to be used later.

The screenshot shows the 'Shared access policies' page for the 'ContosoHub' IoT hub. It lists policies like 'iothubowner', 'service', 'device', 'registryRead', and 'registryReadWrite'. The 'iothubowner' policy is selected. On the right, the 'Shared access keys' section shows the 'Primary key' and 'Secondary key' fields, with the 'Connection string--primary key' field highlighted by a red box. The connection string value is 'HostName=ContosoHub.azure-devices.net;SharedAccessKey...'. There are also 'Regen key' and 'Delete' buttons for the policy.

For more information, see [Access control](#) in the "IoT Hub developer guide."

Create a device identity

In this section, you use the Azure CLI to create a device identity for this tutorial. The Azure CLI is preinstalled in the [Azure Cloud Shell](#), or you can [install it locally](#). Device IDs are case sensitive.

1. Run the following command in the command-line environment where you are using the Azure CLI to install the IoT extension:

```
az extension add --name azure-cli-iot-ext
```

2. If you are running the Azure CLI locally, use the following command to sign in to your Azure account (if you are using the Cloud Shell, you are signed in automatically and you don't need to run this command):

```
az login
```

3. Finally, create a new device identity called `myDeviceId` and retrieve the device connection string with these commands:

```
az iot hub device-identity create --device-id myDeviceId --hub-name {Your IoT Hub name}  
az iot hub device-identity show-connection-string --device-id myDeviceId --hub-name {Your IoT Hub name} -o table
```

IMPORTANT

The device ID may be visible in the logs collected for customer support and troubleshooting, so make sure to avoid any sensitive information while naming it.

Make a note of the device connection string from the result. This device connection string is used by the device app to connect to your IoT Hub as a device.

Create a simulated device app

In this section, you create a Node.js console app that responds to a direct method called by the cloud, which triggers a simulated **lockDoor** method.

1. Create a new empty folder called **simDevice**. In the **simDevice** folder, create a package.json file using the following command at your command prompt. Accept all the defaults:

```
npm init
```

2. At your command prompt in the **simDevice** folder, run the following command to install the **azure-iot-device** Device SDK package and **azure-iot-device-mqtt** package:

```
npm install azure-iot-device azure-iot-device-mqtt --save
```

3. Using a text editor, create a new **simDevice.js** file in the **simDevice** folder.

4. Add the following 'require' statements at the start of the **simDevice.js** file:

```
'use strict';

var Client = require('azure-iot-device').Client;
var Protocol = require('azure-iot-device-mqtt').Mqtt;
```

5. Add a **connectionString** variable and use it to create a **Client** instance.

```
var connectionString = 'HostName={youriothostname};DeviceId={yourdeviceid};SharedAccessKey=
{yourdevicekey}';
var client = Client.fromConnectionString(connectionString, Protocol);
```

6. Add the following function to handle the **lockDoor** method.

```
var onLockDoor = function(request, response) {

    // Respond the cloud app for the direct method
    response.send(200, function(err) {
        if (!err) {
            console.error('An error occurred when sending a method response:\n' + err.toString());
        } else {
            console.log('Response to method \'' + request.methodName + '\' sent successfully.');
        }
    });

    console.log('Locking Door!');
};
```

7. Add the following code to register the handler for the **lockDoor** method.

```
client.open(function(err) {
    if (err) {
        console.error('Could not connect to IoT Hub client.');
    } else {
        console.log('Client connected to IoT Hub. Register handler for lockDoor direct method.');
        client.onDeviceMethod('lockDoor', onLockDoor);
    }
});
```

8. Save and close the **simDevice.js** file.

NOTE

To keep things simple, this tutorial does not implement any retry policy. In production code, you should implement retry policies (such as an exponential backoff), as suggested in the article, [Transient Fault Handling](#).

Schedule jobs for calling a direct method and updating a device twin's properties

In this section, you create a Node.js console app that initiates a remote **lockDoor** on a device using a direct method and update the device twin's properties.

1. Create a new empty folder called **scheduleJobService**. In the **scheduleJobService** folder, create a `package.json` file using the following command at your command prompt. Accept all the defaults:

```
npm init
```

2. At your command prompt in the **scheduleJobService** folder, run the following command to install the **azure-iothub** Device SDK package and **azure-iot-device-mqtt** package:

```
npm install azure-iothub uuid --save
```

3. Using a text editor, create a new **scheduleJobService.js** file in the **scheduleJobService** folder.

4. Add the following 'require' statements at the start of the **dmpatterns_gscheduleJobServiceetstarted_service.js** file:

```
'use strict';

var uuid = require('uuid');
var JobClient = require('azure-iothub').JobClient;
```

5. Add the following variable declarations and replace the placeholder values:

```
var connectionString = '{iothubconnectionstring}';
var queryCondition = "deviceId IN ['myDeviceId']";
var startTime = new Date();
var maxExecutionTimeInSeconds = 300;
var jobClient = JobClient.fromConnectionString(connectionString);
```

6. Add the following function that is used to monitor the execution of the job:

```
function monitorJob (jobId, callback) {
    var jobMonitorInterval = setInterval(function() {
        jobClient.getJob(jobId, function(err, result) {
            if (err) {
                console.error('Could not get job status: ' + err.message);
            } else {
                console.log('Job: ' + jobId + ' - status: ' + result.status);
                if (result.status === 'completed' || result.status === 'failed' || result.status ===
'recancelled') {
                    clearInterval(jobMonitorInterval);
                    callback(null, result);
                }
            }
        });
    }, 5000);
}
```

7. Add the following code to schedule the job that calls the device method:

```

var methodParams = {
    methodName: 'lockDoor',
    payload: null,
    responseTimeoutInSeconds: 15 // Timeout after 15 seconds if device is unable to process method
};

var methodJobId = uuid.v4();
console.log('scheduling Device Method job with id: ' + methodJobId);
jobClient.scheduleDeviceMethod(methodJobId,
    queryCondition,
    methodParams,
    startTime,
    maxExecutionTimeInSeconds,
    function(err) {
    if (err) {
        console.error('Could not schedule device method job: ' + err.message);
    } else {
        monitorJob(methodJobId, function(err, result) {
            if (err) {
                console.error('Could not monitor device method job: ' + err.message);
            } else {
                console.log(JSON.stringify(result, null, 2));
            }
        });
    }
});

```

- Add the following code to schedule the job to update the device twin:

```

var twinPatch = {
    etag: '*',
    properties: {
        desired: {
            building: '43',
            floor: 3
        }
    }
};

var twinJobId = uuid.v4();

console.log('scheduling Twin Update job with id: ' + twinJobId);
jobClient.scheduleTwinUpdate(twinJobId,
    queryCondition,
    twinPatch,
    startTime,
    maxExecutionTimeInSeconds,
    function(err) {
    if (err) {
        console.error('Could not schedule twin update job: ' + err.message);
    } else {
        monitorJob(twinJobId, function(err, result) {
            if (err) {
                console.error('Could not monitor twin update job: ' + err.message);
            } else {
                console.log(JSON.stringify(result, null, 2));
            }
        });
    }
});

```

- Save and close the **scheduleJobService.js** file.

Run the applications

You are now ready to run the applications.

1. At the command prompt in the **simDevice** folder, run the following command to begin listening for the reboot direct method.

```
node simDevice.js
```

2. At the command prompt in the **scheduleJobService** folder, run the following command to trigger the jobs to lock the door and update the twin

```
node scheduleJobService.js
```

3. You see the device response to the direct method in the console.

Next steps

In this tutorial, you used a job to schedule a direct method to a device and the update of the device twin's properties.

To continue getting started with IoT Hub and device management patterns such as remote over the air firmware update, see:

[Tutorial: How to do a firmware update](#)

To continue getting started with IoT Hub, see [Getting started with Azure IoT Edge](#).

Schedule and broadcast jobs (.NET/.NET)

2/28/2019 • 9 minutes to read

Use Azure IoT Hub to schedule and track jobs that update millions of devices. Use jobs to:

- Update desired properties
- Update tags
- Invoke direct methods

A job wraps one of these actions and tracks the execution against a set of devices that is defined by a device twin query. For example, a back-end app can use a job to invoke a direct method on 10,000 devices that reboots the devices. You specify the set of devices with a device twin query and schedule the job to run at a future time. The job tracks progress as each of the devices receive and execute the reboot direct method.

To learn more about each of these capabilities, see:

- Device twin and properties: [Get started with device twins](#) and [Tutorial: How to use device twin properties](#)
- Direct methods: [IoT Hub developer guide - direct methods](#) and [Tutorial: Use direct methods](#)

NOTE

The features described in this article are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

This tutorial shows you how to:

- Create a device app that implements a direct method called **LockDoor** that can be called by the back-end app.
- Create a back-end app that creates a job to call the **LockDoor** direct method on multiple devices. Another job sends desired property updates to multiple devices.

At the end of this tutorial, you have two .NET (C#) console apps:

SimulateDeviceMethods that connects to your IoT hub and implements the **LockDoor** direct method.

ScheduleJob that uses jobs to call the **LockDoor** direct method and update the device twin desired properties on multiple devices.

To complete this tutorial, you need the following:

- Visual Studio 2017.
- An active Azure account. If you don't have an account, you can create a [free account](#) in just a couple of minutes.

Create an IoT hub

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose +**Create a resource**, then choose **Internet of Things**.
3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Review + create

Create an IoT Hub to help you connect, monitor, and manage billions of your IoT assets. [Learn More](#)

PROJECT DETAILS

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

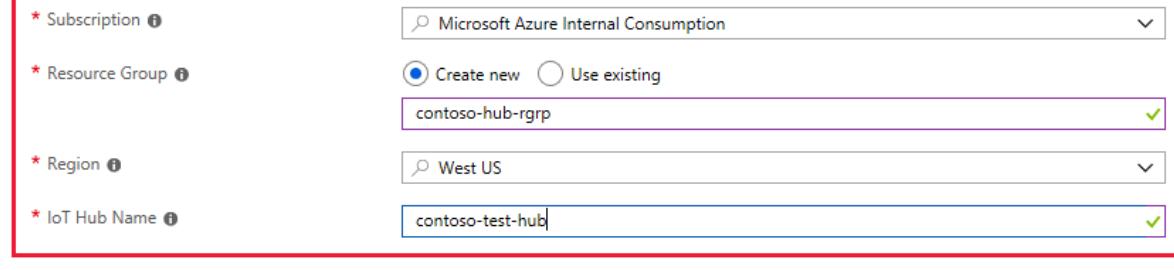
* Subscription

* Resource Group Create new Use existing

* Region

* IoT Hub Name

Review + create **Next: Size and scale »** Automation options



Fill in the fields.

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics **Size and scale** Review + create

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS

* Pricing and scale tier **S1: Standard tier** [Learn how to choose the right IoT Hub tier for your solution](#)

Number of S1 IoT Hub units **1** This determines your IoT Hub scale capability and can be changed as your need increases.

Device-to-cloud-messages **Enabled**
Message routing **Enabled**
Cloud-to-device commands **Enabled**
IoT Edge **Enabled**
Device management **Enabled**

Advanced Settings

Device-to-cloud partitions **4**

Review + create [« Previous: Basics](#) [Automation options](#)

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: The number of messages allowed per unit per day depends on your hub's pricing tier. For example, if you want the IoT hub to support ingress of 700,000 messages, you choose two S1 tier units.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

Advanced / Device-to-cloud partitions: This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most IoT hubs only need four partitions.

5. Click **Review + create** to review your choices. You see something similar to this screen.

The screenshot shows the 'Review + create' step of the IoT hub creation wizard. It displays basic information like subscription, resource group, region, and IoT hub name, along with size and scale details. The 'Create' button at the bottom left is highlighted with a red box.

6. Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Retrieve connection string for IoT hub

After your hub has been created, retrieve the connection string for the hub. This is used to connect devices and applications to your hub.

1. Click on your hub to see the IoT Hub pane with Settings, and so on. Click **Shared access policies**.
2. In **Shared access policies**, select the **iothubowner** policy.
3. Under **Shared access keys**, copy the **Connection string -- primary key** to be used later.

The screenshot shows the 'Shared access policies' section of the IoT Hub settings. It lists the 'iothubowner' policy with various permissions selected. The 'Shared access keys' section shows the primary and secondary connection strings, with the primary one highlighted with a red box.

For more information, see [Access control](#) in the "IoT Hub developer guide."

Register a new device in the IoT hub

In this section, you create a device identity in the identity registry in your IoT hub. A device cannot connect to IoT hub unless it has an entry in the identity registry. For more information, see the "Identity registry" section of the [IoT Hub developer guide](#)

1. In your IoT hub navigation menu, open **IoT Devices**, then click **Add** to register a new device in your IoT hub.

The screenshot shows the 'ContosoHub - IoT devices' blade in the Azure portal. The left sidebar contains several sections: 'Events', 'SETTINGS' (with options like Shared access policies, Pricing and scale, Operations monitoring, IP Filter, Certificates, Properties, Locks, and Automation script), 'EXPLORERS' (with Query explorer and IoT devices selected), and 'AUTOMATIC DEVICE MANAGEMENT' (with IoT Edge (preview)). The main area features a 'Query' tool with a sample SQL query: 'SELECT * FROM devices WHERE optional (e.g. tags.location='US')'. Below the query is an 'Execute' button. A table header is visible with columns: DEVICE ID, STATUS, LAST ACTIVITY, LAST STATUS..., AUTHENTICATI..., CLOUD TO DEV... . A message at the top states: 'You can use this tool to view, create, update, and delete devices on your IoT Hub.'

2. Provide a name for your new device, such as **myDeviceId**, and click **Save**. This action creates a new device identity for your IoT hub.

Create a device

Learn more about creating devices

* Device ID ⓘ
myDeviceId

Authentication type ⓘ
Symmetric key X.509 Self-Signed X.509 CA Signed

* Primary key ⓘ
Enter your primary key

* Secondary key ⓘ
Enter your secondary key

Auto-generate keys ⓘ

Connect this device to an IoT hub ⓘ
Enable Disable

Parent device (Preview) ⓘ
No parent device
Set a parent device

Save

IMPORTANT

The device ID may be visible in the logs collected for customer support and troubleshooting, so make sure to avoid any sensitive information while naming it.

3. After the device is created, open the device from the list in the **IoT devices** pane. Copy the **Connection string---primary key** to use later.

Device details
myDeviceId

Device Id: myDeviceId

Primary key: <Primary Key>

Secondary key: <Secondary Key>

Connection string (primary key):
HostName=ContosoHub.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=<Primary Key>

Connection string (secondary key):
HostName=ContosoHub.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=<Secondary Key>

Connect this device to an IoT hub: [Enable](#) [Disable](#)

Parent device (Preview): [No parent device](#) [Set a parent device](#)

NOTE

The IoT Hub identity registry only stores device identities to enable secure access to the IoT hub. It stores device IDs and keys to use as security credentials, and an enabled/disabled flag that you can use to disable access for an individual device. If your application needs to store other device-specific metadata, it should use an application-specific store. For more information, see [IoT Hub developer guide](#).

Create a simulated device app

In this section, you create a .NET console app that responds to a direct method called by the solution back end.

1. In Visual Studio, add a Visual C# Windows Classic Desktop project to the current solution by using the **Console Application** project template. Name the project **SimulateDeviceMethods**.

Add New Project

.NET Framework 4.5.2 Sort by: Default

Search Installed Templates (Ctrl+E)

Type: Visual C#

Console App (.NET Framework) Visual C# A project for creating a command-line application

Windows Universal Visual C# Windows Universal

Windows Classic Desktop Visual C# Windows Classic Desktop

Web Visual C# Web

.NET Core Visual C# .NET Core

.NET Standard Visual C# .NET Standard

Android Visual C# Android

Cloud Visual C# Cloud

Cross-Platform Visual C# Cross-Platform

Extensibility Visual C# Extensibility

iOS Visual C# iOS

Test Visual C# Test

tvOS Visual C# tvOS

WCF Visual C# WCF

Visual Basic Visual Basic

Visual C++ Visual C++

Visual F# Visual F#

SQL Server SQL Server

Azure Data Lake Azure Data Lake

JavaScript JavaScript

TypeScript TypeScript

Dependency Validation Dependency Validation

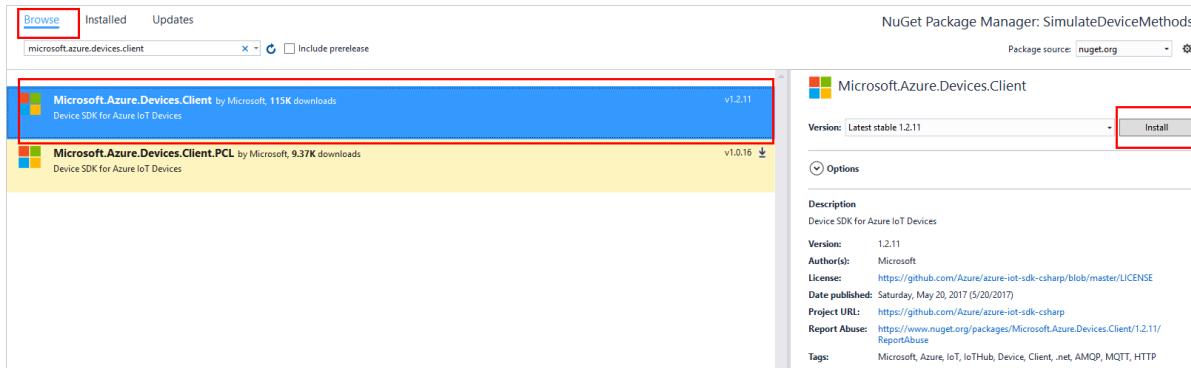
Name: SimulateDeviceMethods

Location: C:\code\iotDevice

OK Cancel

2. In Solution Explorer, right-click the **SimulateDeviceMethods** project, and then click **Manage NuGet Packages....**
3. In the **NuGet Package Manager** window, select **Browse** and search for

Microsoft.Azure.Devices.Client. Select **Install** to install the **Microsoft.Azure.Devices.Client** package, and accept the terms of use. This procedure downloads, installs, and adds a reference to the [Azure IoT device SDK](#) NuGet package and its dependencies.



4. Add the following `using` statements at the top of the **Program.cs** file:

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

5. Add the following fields to the **Program** class. Replace the placeholder value with the device connection string that you noted in the previous section:

```
static string DeviceConnectionString = "<yourDeviceConnectionString>";
static DeviceClient Client = null;
```

6. Add the following to implement the direct method on the device:

```
static Task<MethodResponse> LockDoor(MethodRequest methodRequest, object userContext)
{
    Console.WriteLine();
    Console.WriteLine("Locking Door!");
    Console.WriteLine("\nReturning response for method {0}", methodRequest.Name);

    string result = "Door was locked.";
    return Task.FromResult(new MethodResponse(Encoding.UTF8.GetBytes(result), 200));
}
```

7. Add the following to implement the device twins listener on the device:

```
private static async Task OnDesiredPropertyChanged(TwinCollection desiredProperties,
    object userContext)
{
    Console.WriteLine("Desired property change:");
    Console.WriteLine(JsonConvert.SerializeObject(desiredProperties));
}
```

8. Finally, add the following code to the **Main** method to open the connection to your IoT hub and initialize the method listener:

```
try
{
    Console.WriteLine("Connecting to hub");
    Client = DeviceClient.CreateFromConnectionString(DeviceConnectionString,
        TransportType.Mqtt);

    Client.SetMethodHandlerAsync("LockDoor", LockDoor, null);
    Client.SetDesiredPropertyUpdateCallbackAsync(OnDesiredPropertyChanged, null);

    Console.WriteLine("Waiting for direct method call and device twin update\n Press enter to exit.");
    Console.ReadLine();

    Console.WriteLine("Exiting...");

    Client.SetMethodHandlerAsync("LockDoor", null, null);
    Client.CloseAsync().Wait();
}
catch (Exception ex)
{
    Console.WriteLine();
    Console.WriteLine("Error in sample: {0}", ex.Message);
}
```

9. Save your work and build your solution.

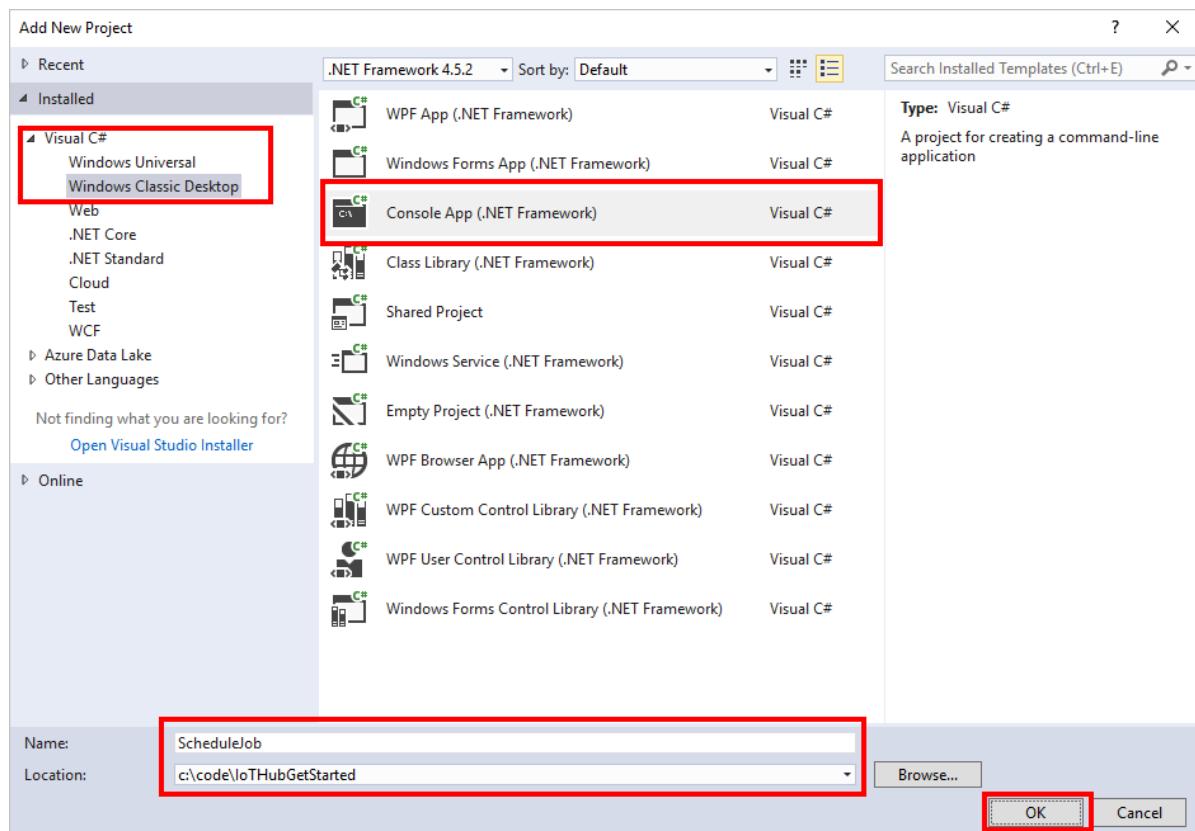
NOTE

To keep things simple, this tutorial does not implement any retry policy. In production code, you should implement retry policies (such as connection retry), as suggested in the article, [Transient Fault Handling](#).

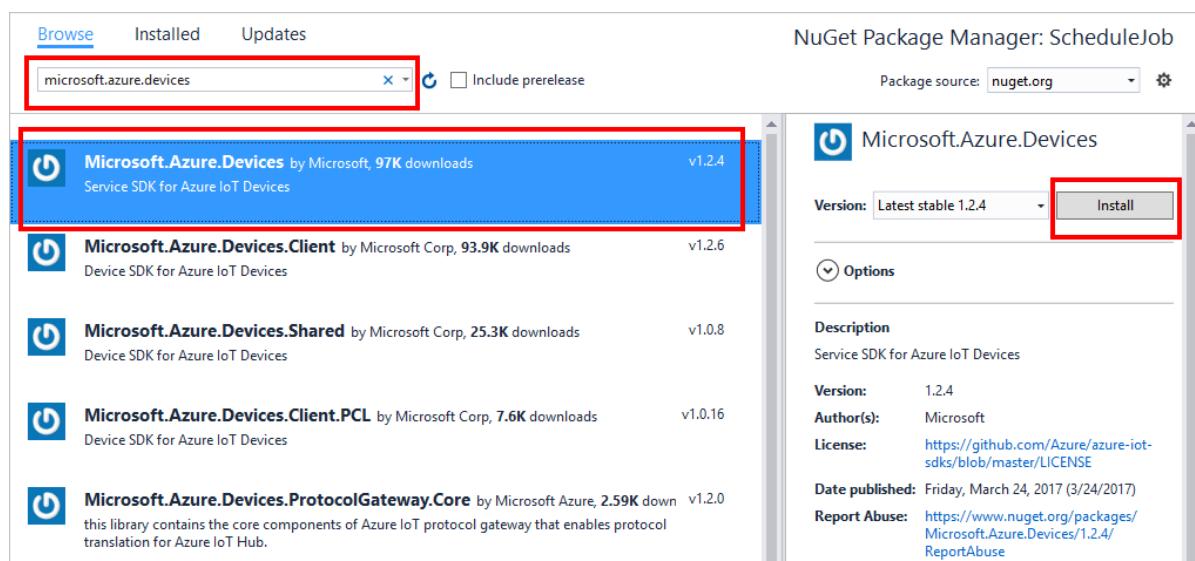
Schedule jobs for calling a direct method and sending device twin updates

In this section, you create a .NET console app (using C#) that uses jobs to call the **LockDoor** direct method and send desired property updates to multiple devices.

1. In Visual Studio, add a Visual C# Windows Classic Desktop project to the current solution by using the **Console Application** project template. Name the project **ScheduleJob**.



2. In Solution Explorer, right-click the **ScheduleJob** project, and then click **Manage NuGet Packages....**
3. In the **NuGet Package Manager** window, select **Browse**, search for **Microsoft.Azure.Devices**, select **Install** to install the **Microsoft.Azure.Devices** package, and accept the terms of use. This step downloads, installs, and adds a reference to the [Azure IoT service SDK](#) NuGet package and its dependencies.



4. Add the following `using` statements at the top of the **Program.cs** file:

```
using Microsoft.Azure.Devices;
using Microsoft.Azure.Devices.Shared;
```

5. Add the following `using` statement if not already present in the default statements.

```
using System.Threading;
using System.Threading.Tasks;
```

6. Add the following fields to the **Program** class. Replace the placeholders with the IoT Hub connection string for the hub that you created in the previous section and the name of your device.

```
static JobClient jobClient;
static string connString = "<yourIoTHubConnectionString>";
static string deviceId = "<yourDeviceId>";
```

7. Add the following method to the **Program** class:

```
public static async Task MonitorJob(string jobId)
{
    JobResponse result;
    do
    {
        result = await jobClient.GetJobAsync(jobId);
        Console.WriteLine("Job Status : " + result.Status.ToString());
        Thread.Sleep(2000);
    } while ((result.Status != JobStatus.Completed) &&
             (result.Status != JobStatus.Failed));
}
```

8. Add the following method to the **Program** class:

```
public static async Task StartMethodJob(string jobId)
{
    CloudToDeviceMethod directMethod =
        new CloudToDeviceMethod("LockDoor", TimeSpan.FromSeconds(5),
                               TimeSpan.FromSeconds(5));

    JobResponse result = await jobClient.ScheduleDeviceMethodAsync(jobId,
        $"DeviceId IN ['{deviceId}']",
        directMethod,
        DateTime.UtcNow,
        (long)TimeSpan.FromMinutes(2).TotalSeconds);

    Console.WriteLine("Started Method Job");
}
```

9. Add another method to the **Program** class:

```
public static async Task StartTwinUpdateJob(string jobId)
{
    Twin twin = new Twin(deviceId);
    twin.Tags = new TwinCollection();
    twin.Tags["Building"] = "43";
    twin.Tags["Floor"] = "3";
    twin.ETag = "*";

    twin.Properties.Desired["LocationUpdate"] = DateTime.UtcNow;

    JobResponse createJobResponse = jobClient.ScheduleTwinUpdateAsync(
        jobId,
        $"DeviceId IN ['{deviceId}']",
        twin,
        DateTime.UtcNow,
        (long)TimeSpan.FromMinutes(2).TotalSeconds).Result;

    Console.WriteLine("Started Twin Update Job");
}
```

NOTE

For more information about query syntax, see [IoT Hub query language](#).

- Finally, add the following lines to the **Main** method:

```
Console.WriteLine("Press ENTER to start running jobs.");
Console.ReadLine();

jobClient = JobClient.CreateFromConnectionString(connString);

string methodJobId = Guid.NewGuid().ToString();

StartMethodJob(methodJobId);
MonitorJob(methodJobId).Wait();
Console.WriteLine("Press ENTER to run the next job.");
Console.ReadLine();

string twinUpdateJobId = Guid.NewGuid().ToString();

StartTwinUpdateJob(twinUpdateJobId);
MonitorJob(twinUpdateJobId).Wait();
Console.WriteLine("Press ENTER to exit.");
Console.ReadLine();
```

- Save your work and build your solution.

Run the apps

You are now ready to run the apps.

- In the Visual Studio Solution Explorer, right-click your solution, and then click **Build. Multiple startup projects**. Make sure `SimulateDeviceMethods` is at the top of the list followed by `ScheduleJob`. Set both their actions to **Start** and click **OK**.
- Run the projects by clicking **Start** or go to the **Debug** menu and click **Start Debugging**.
- You see the output from both device and back-end apps.

The image shows two terminal windows side-by-side. The left window, titled 'C:\Users\v-masebo\Desktop\working\LockDoor\SimulateLockDoor\bin\Debug\SimulateLockDoor.exe', displays the following text:
Connecting to hub
Waiting for direct method call and device twins update
Press enter to exit.
Locking Door!
Returning response for method LockDoor
Desired property change:
{"LocationUpdate":"2018-02-16T20:58:37.4504092Z", "\$version":14}
The right window, titled 'C:\Users\v-masebo\Desktop\working\LockDoor\Job\bin\Debug\Job.exe', displays the following text:
Press ENTER to start running jobs.
Job Status : Unknown
Started Method Job
Job Status : Running
Job Status : Running
Job Status : Completed
Press ENTER to run the next job.
Started Twin Update Job
Job Status : Queued
Job Status : Running
Job Status : Completed
Press ENTER to exit.

Next steps

In this tutorial, you used a job to schedule a direct method to a device and the update of the device twin's properties.

To continue getting started with IoT Hub and device management patterns such as remote over the air firmware update, read [Tutorial: How to do a firmware update](#).

To learn about deploying AI to edge devices with Azure IoT Edge, see [Getting started with IoT Edge](#).

Schedule and broadcast jobs (Java)

3/6/2019 • 14 minutes to read

Use Azure IoT Hub to schedule and track jobs that update millions of devices. Use jobs to:

- Update desired properties
- Update tags
- Invoke direct methods

A job wraps one of these actions and tracks the execution against a set of devices. A device twin query defines the set of devices the job executes against. For example, a back-end app can use a job to invoke a direct method on 10,000 devices that reboots the devices. You specify the set of devices with a device twin query and schedule the job to run at a future time. The job tracks progress as each of the devices receive and execute the reboot direct method.

To learn more about each of these capabilities, see:

- Device twin and properties: [Get started with device twins](#)
- Direct methods: [IoT Hub developer guide - direct methods](#) and [Tutorial: Use direct methods](#)

NOTE

The features described in this article are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

This tutorial shows you how to:

- Create a device app that implements a direct method called **lockDoor**. The device app also receives desired property changes from the back-end app.
- Create a back-end app that creates a job to call the **lockDoor** direct method on multiple devices. Another job sends desired property updates to multiple devices.

At the end of this tutorial, you have a java console device app and a java console back-end app:

simulated-device that connects to your IoT hub, implements the **lockDoor** direct method, and handles desired property changes.

schedule-jobs that use jobs to call the **lockDoor** direct method and update the device twin desired properties on multiple devices.

NOTE

The article [Azure IoT SDKs](#) provides information about the Azure IoT SDKs that you can use to build both device and back-end apps.

Prerequisites

To complete this tutorial, you need:

- The latest [Java SE Development Kit 8](#)

- Maven 3
- An active Azure account. (If you don't have an account, you can create a [free account](#) in just a couple of minutes.)

Create an IoT hub

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose +**Create a resource**, then choose **Internet of Things**.
3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.

The screenshot shows the 'Basics' step of the IoT Hub creation wizard. It includes fields for Subscription (set to Microsoft Azure Internal Consumption), Resource Group (set to contoso-hub-rgrp), Region (set to West US), and IoT Hub Name (set to contoso-test-hub). The 'Next: Size and scale' button is highlighted with a red box.

Fill in the fields.

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

IoT hub

Microsoft

[Basics](#) [Size and scale](#) [Review + create](#)

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS* Pricing and scale tier [?](#)

S1: Standard tier

[Learn how to choose the right IoT Hub tier for your solution](#)Number of S1 IoT Hub units [?](#)

This determines your IoT Hub scale capability and can be changed as your need increases.

Device-to-cloud-messages [?](#) EnabledMessage routing [?](#) EnabledCloud-to-device commands [?](#) EnabledIoT Edge [?](#) EnabledDevice management [?](#) Enabled[Advanced Settings](#)Device-to-cloud partitions [?](#)[Review + create](#)[« Previous: Basics](#)[Automation options](#)

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: The number of messages allowed per unit per day depends on your hub's pricing tier. For example, if you want the IoT hub to support ingress of 700,000 messages, you choose two S1 tier units.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

Advanced / Device-to-cloud partitions: This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most IoT hubs only need four partitions.

- Click **Review + create** to review your choices. You see something similar to this screen.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale **Review + create**

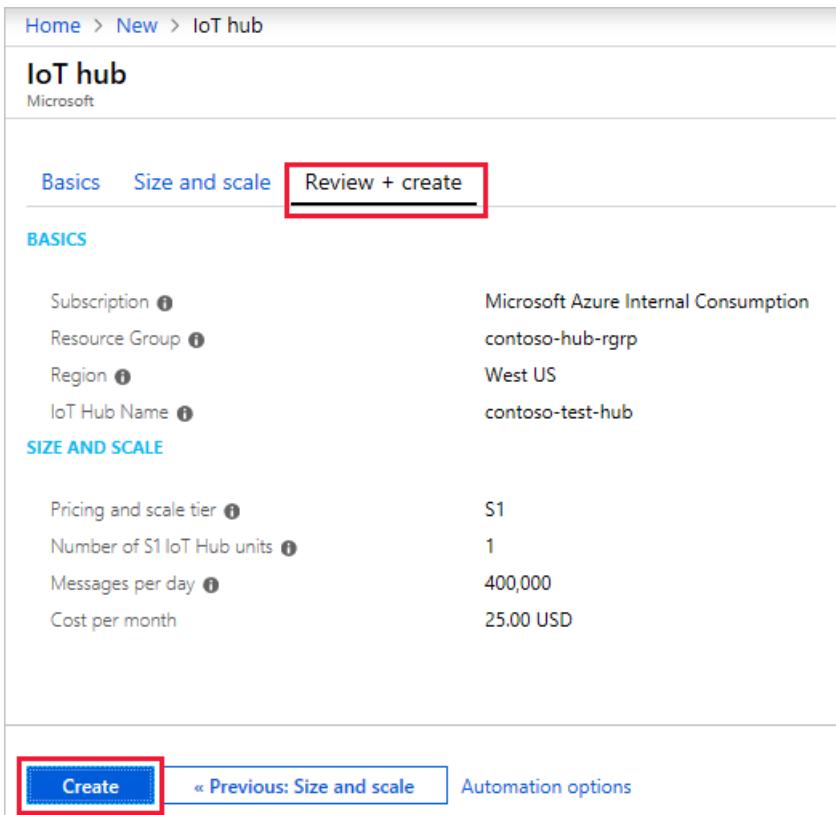
BASICS

Subscription	Microsoft Azure Internal Consumption
Resource Group	contoso-hub-rgrp
Region	West US
IoT Hub Name	contoso-test-hub

SIZE AND SCALE

Pricing and scale tier	S1
Number of S1 IoT Hub units	1
Messages per day	400,000
Cost per month	25.00 USD

Create [« Previous: Size and scale](#) [Automation options](#)



6. Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Retrieve connection string for IoT hub

After your hub has been created, retrieve the connection string for the hub. This is used to connect devices and applications to your hub.

1. Click on your hub to see the IoT Hub pane with Settings, and so on. Click **Shared access policies**.
2. In **Shared access policies**, select the **iothubowner** policy.
3. Under **Shared access keys**, copy the **Connection string -- primary key** to be used later.

Home > IoT Hub > ContosoHub - Shared access policies

ContosoHub - Shared access policies

iothubowner
ContosoHub

Add

IoT Hub uses permissions to grant access to each functionality.

Search to filter items...

POLICY

iothubowner
service
device
registryRead
registryReadWrite

Save **Discard** **Regen key** **Delete**

Access policy name: iothubowner

Permissions:

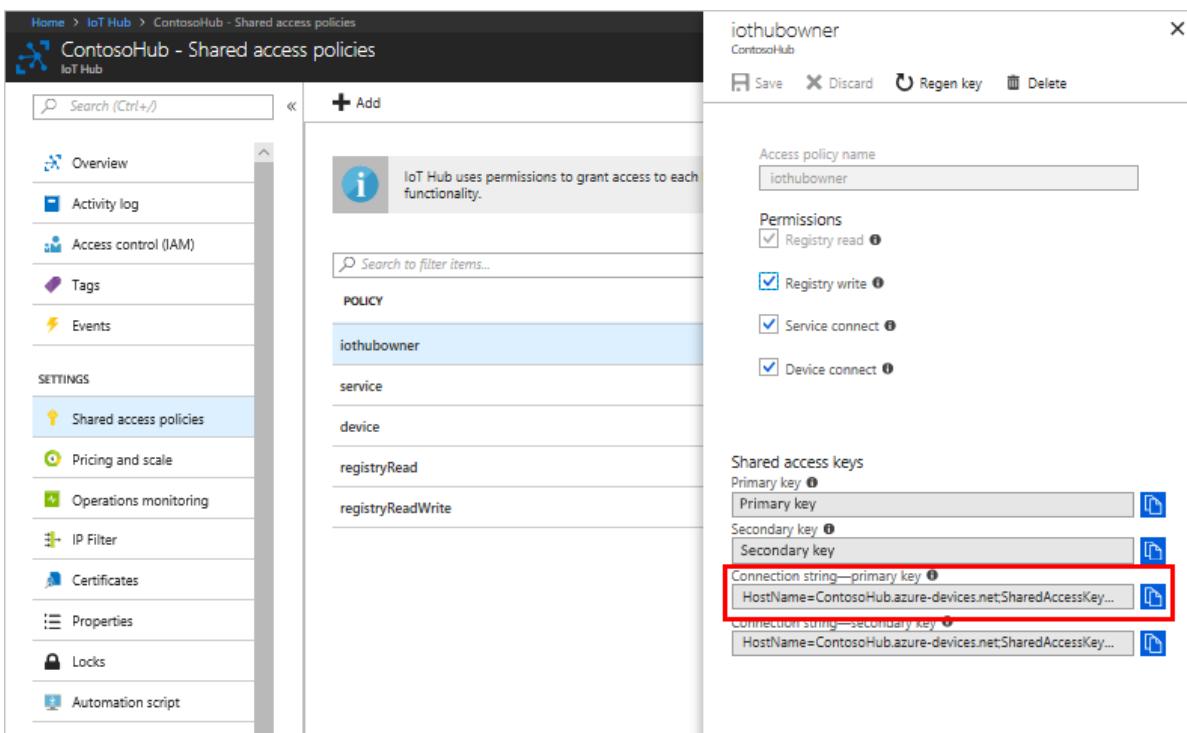
- Registry read
- Registry write
- Service connect
- Device connect

Shared access keys

Primary key	<input type="text" value="HostName=ContosoHub.azure-devices.net;SharedAccessKey..."/>
Secondary key	<input type="text" value="HostName=ContosoHub.azure-devices.net;SharedAccessKey..."/>

Connection string--primary key

Connection string--secondary key



For more information, see [Access control](#) in the "IoT Hub developer guide."

Register a new device in the IoT hub

In this section, you create a device identity in the identity registry in your IoT hub. A device cannot connect to IoT hub unless it has an entry in the identity registry. For more information, see the "Identity registry" section of the [IoT Hub developer guide](#)

1. In your IoT hub navigation menu, open **IoT Devices**, then click **Add** to register a new device in your IoT hub.

The screenshot shows the 'ContosoHub - IoT devices' blade in the Azure portal. The left sidebar contains various settings and management options. The 'IoT devices' item in the 'EXPLORERS' section is highlighted with a red box. The main area displays a query editor with a placeholder query and an 'Execute' button. Below the query editor is a table header with columns: DEVICE ID, STATUS, LAST ACTIVITY, LAST STATUS..., AUTHENTICATI..., and CLOUD TO DEV... . The message 'No results' is displayed below the table.

2. Provide a name for your new device, such as **myDeviceId**, and click **Save**. This action creates a new device identity for your IoT hub.

Create a device

Learn more about creating devices

* Device ID [i](#)
myDeviceId ✓

Authentication type [i](#)
[Symmetric key](#) [X.509 Self-Signed](#) [X.509 CA Signed](#)

* Primary key [i](#)
Enter your primary key

* Secondary key [i](#)
Enter your secondary key

Auto-generate keys [i](#)

Connect this device to an IoT hub [i](#)
[Enable](#) [Disable](#)

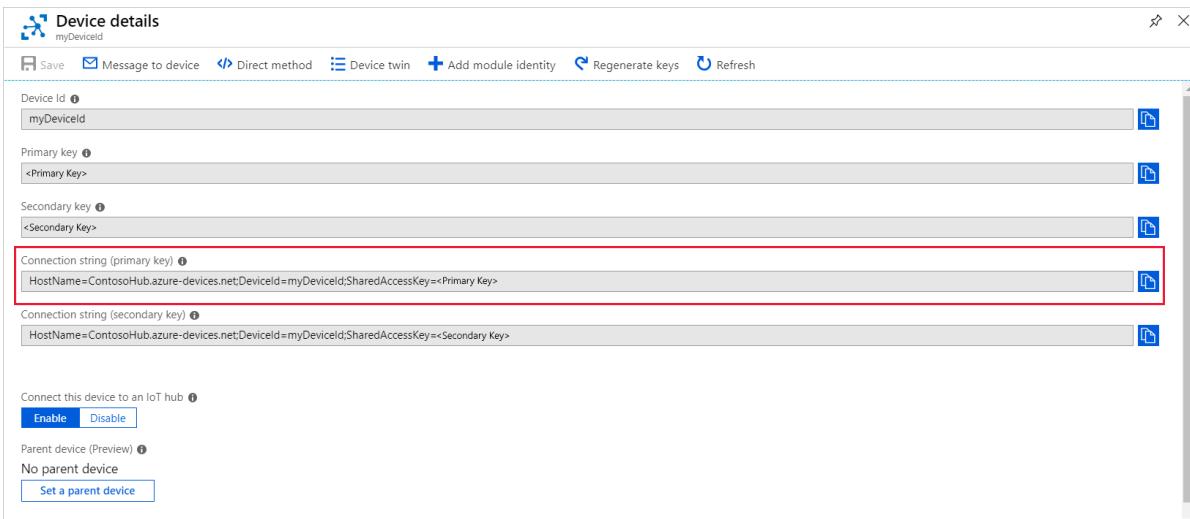
Parent device (Preview) [i](#)
No parent device
[Set a parent device](#)

[Save](#)

IMPORTANT

The device ID may be visible in the logs collected for customer support and troubleshooting, so make sure to avoid any sensitive information while naming it.

3. After the device is created, open the device from the list in the **IoT devices** pane. Copy the **Connection string---primary key** to use later.



NOTE

The IoT Hub identity registry only stores device identities to enable secure access to the IoT hub. It stores device IDs and keys to use as security credentials, and an enabled/disabled flag that you can use to disable access for an individual device. If your application needs to store other device-specific metadata, it should use an application-specific store. For more information, see [IoT Hub developer guide](#).

You can also use the [IoT extension for Azure CLI](#) tool to add a device to your IoT hub.

Create the service app

In this section, you create a Java console app that uses jobs to:

- Call the **lockDoor** direct method on multiple devices.
- Send desired properties to multiple devices.

To create the app:

1. On your development machine, create an empty folder called `iot-java-schedule-jobs`.
2. In the `iot-java-schedule-jobs` folder, create a Maven project called **schedule-jobs** using the following command at your command prompt. Note this is a single, long command:

```
mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=schedule-jobs -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```
3. At your command prompt, navigate to the `schedule-jobs` folder.
4. Using a text editor, open the `pom.xml` file in the `schedule-jobs` folder and add the following dependency to the **dependencies** node. This dependency enables you to use the **iot-service-client** package in your app to communicate with your IoT hub:

```
<dependency>
  <groupId>com.microsoft.azure.sdk.iot</groupId>
  <artifactId>iot-service-client</artifactId>
  <version>1.7.23</version>
  <type>jar</type>
</dependency>
```

NOTE

You can check for the latest version of **iot-service-client** using [Maven search](#).

5. Add the following **build** node after the **dependencies** node. This configuration instructs Maven to use Java 1.8 to build the app:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.3</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

6. Save and close the `pom.xml` file.
7. Using a text editor, open the `schedule-jobs\src\main\java\com\mycompany\app\App.java` file.
8. Add the following **import** statements to the file:

```
import com.microsoft.azure.sdk.iot.service.devicetwin.DeviceTwinDevice;
import com.microsoft.azure.sdk.iot.service.devicetwin.Pair;
import com.microsoft.azure.sdk.iot.service.devicetwin.Query;
import com.microsoft.azure.sdk.iot.service.devicetwin.SqlQuery;
import com.microsoft.azure.sdk.iot.service.jobs.JobClient;
import com.microsoft.azure.sdk.iot.service.jobs.JobResult;
import com.microsoft.azure.sdk.iot.service.jobs.JobStatus;

import java.util.Date;
import java.time.Instant;
import java.util.HashSet;
import java.util.Set;
import java.util.UUID;
```

9. Add the following class-level variables to the **App** class. Replace `{youriothubconnectionstring}` with your IoT hub connection string you noted in the *Create an IoT Hub* section:

```
public static final String iothubConnectionString = "{youriothubconnectionstring}";
public static final String deviceId = "myDeviceId";

// How long the job is permitted to run without
// completing its work on the set of devices
private static final long maxExecutionTimeInSeconds = 30;
```

10. Add the following method to the **App** class to schedule a job to update the **Building** and **Floor** desired properties in the device twin:

```

private static JobResult scheduleJobSetDesiredProperties(JobClient jobClient, String jobId) {
    DeviceTwinDevice twin = new DeviceTwinDevice(deviceId);
    Set<Pair> desiredProperties = new HashSet<Pair>();
    desiredProperties.add(new Pair("Building", 43));
    desiredProperties.add(new Pair("Floor", 3));
    twin.setDesiredProperties(desiredProperties);
    // Optimistic concurrency control
    twin.setETag("*");

    // Schedule the update twin job to run now
    // against a single device
    System.out.println("Schedule job " + jobId + " for device " + deviceId);
    try {
        JobResult jobResult = jobClient.scheduleUpdateTwin(jobId,
            "deviceId=" + deviceId + '',
            twin,
            new Date(),
            maxExecutionTimeInSeconds);
        return jobResult;
    } catch (Exception e) {
        System.out.println("Exception scheduling desired properties job: " + jobId);
        System.out.println(e.getMessage());
        return null;
    }
}

```

11. To schedule a job to call the **lockDoor** method, add the following method to the **App** class:

```

private static JobResult scheduleJobCallDirectMethod(JobClient jobClient, String jobId) {
    // Schedule a job now to call the lockDoor direct method
    // against a single device. Response and connection
    // timeouts are set to 5 seconds.
    System.out.println("Schedule job " + jobId + " for device " + deviceId);
    try {
        JobResult jobResult = jobClient.scheduleDeviceMethod(jobId,
            "deviceId=" + deviceId + '',
            "lockDoor",
            5L, 5L, null,
            new Date(),
            maxExecutionTimeInSeconds);
        return jobResult;
    } catch (Exception e) {
        System.out.println("Exception scheduling direct method job: " + jobId);
        System.out.println(e.getMessage());
        return null;
    }
}

```

12. To monitor a job, add the following method to the **App** class:

```

private static void monitorJob(JobClient jobClient, String jobId) {
    try {
        JobResult jobResult = jobClient.getJob(jobId);
        if(jobResult == null)
        {
            System.out.println("No JobResult for: " + jobId);
            return;
        }
        // Check the job result until it's completed
        while(jobResult.getJobStatus() != JobStatus.completed)
        {
            Thread.sleep(100);
            jobResult = jobClient.getJob(jobId);
            System.out.println("Status " + jobResult.getJobStatus() + " for job " + jobId);
        }
        System.out.println("Final status " + jobResult.getJobStatus() + " for job " + jobId);
    } catch (Exception e) {
        System.out.println("Exception monitoring job: " + jobId);
        System.out.println(e.getMessage());
        return;
    }
}

```

13. To query for the details of the jobs you ran, add the following method:

```

private static void queryDeviceJobs(JobClient jobClient, String start) throws Exception {
    System.out.println("\nQuery device jobs since " + start);

    // Create a jobs query using the time the jobs started
    Query deviceJobQuery = jobClient
        .queryDeviceJob(SqlQuery.createSqlQuery("*", SqlQuery.FromType.JOBS, "devices.jobs.startTimeUtc > "
        + start + "'", null).getQuery());

    // Iterate over the list of jobs and print the details
    while (jobClient.hasNextJob(deviceJobQuery)) {
        System.out.println(jobClient.getNextJob(deviceJobQuery));
    }
}

```

14. Update the **main** method signature to include the following `throws` clause:

```
public static void main( String[] args ) throws Exception
```

15. To run and monitor two jobs sequentially, add the following code to the **main** method:

```

// Record the start time
String start = Instant.now().toString();

// Create JobClient
JobClient jobClient = JobClient.createFromConnectionString(iotHubConnectionString);
System.out.println("JobClient created with success");

// Schedule twin job desired properties
// Maximum concurrent jobs is 1 for Free and S1 tiers
String desiredPropertiesJobId = "DPCMD" + UUID.randomUUID();
scheduleJobSetDesiredProperties(jobClient, desiredPropertiesJobId);
monitorJob(jobClient, desiredPropertiesJobId);

// Schedule twin job direct method
String directMethodJobId = "DMCMD" + UUID.randomUUID();
scheduleJobCallDirectMethod(jobClient, directMethodJobId);
monitorJob(jobClient, directMethodJobId);

// Run a query to show the job detail
queryDeviceJobs(jobClient, start);

System.out.println("Shutting down schedule-jobs app");

```

16. Save and close the `schedule-jobs\src\main\java\com\mycompany\app\App.java` file
17. Build the **schedule-jobs** app and correct any errors. At your command prompt, navigate to the `schedule-jobs` folder and run the following command:

```
mvn clean package -DskipTests
```

Create a device app

In this section, you create a Java console app that handles the desired properties sent from IoT Hub and implements the direct method call.

1. In the `iot-java-schedule-jobs` folder, create a Maven project called **simulated-device** using the following command at your command prompt. Note this is a single, long command:

```
mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=simulated-device -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

2. At your command prompt, navigate to the `simulated-device` folder.
3. Using a text editor, open the `pom.xml` file in the `simulated-device` folder and add the following dependencies to the **dependencies** node. This dependency enables you to use the **iot-device-client** package in your app to communicate with your IoT hub:

```

<dependency>
  <groupId>com.microsoft.azure.sdk.iot</groupId>
  <artifactId>iot-device-client</artifactId>
  <version>1.3.32</version>
</dependency>

```

NOTE

You can check for the latest version of **iot-device-client** using [Maven search](#).

4. Add the following **build** node after the **dependencies** node. This configuration instructs Maven to use Java 1.8 to build the app:

```

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.3</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
  </plugins>
</build>

```

5. Save and close the `pom.xml` file.
6. Using a text editor, open the `simulated-device\src\main\java\com\mycompany\app\App.java` file.
7. Add the following **import** statements to the file:

```

import com.microsoft.azure.sdk.iot.device.*;
import com.microsoft.azure.sdk.iot.device.DeviceTwin.*;

import java.io.IOException;
import java.net.URISyntaxException;
import java.util.Scanner;

```

8. Add the following class-level variables to the **App** class. Replacing `{youriothubname}` with your IoT hub name, and `{yourdevicekey}` with the device key value you generated in the *Create a device identity* section:

```

private static String connString = "HostName={youriothubname}.azure-devices.net;DeviceId=myDeviceID;SharedAccessKey={yourdevicekey}";
private static IoTHubClientProtocol protocol = IoTHubClientProtocol.MQTT;
private static final int METHOD_SUCCESS = 200;
private static final int METHOD_NOT_DEFINED = 404;

```

This sample app uses the **protocol** variable when it instantiates a **DeviceClient** object.

9. To print device twin notifications to the console, add the following nested class to the **App** class:

```

// Handler for device twin operation notifications from IoT Hub
protected static class DeviceTwinStatusCallBack implements IoTHubEventCallback {
    public void execute(IoTHubStatusCode status, Object context) {
        System.out.println("IoT Hub responded to device twin operation with status " + status.name());
    }
}

```

10. To print direct method notifications to the console, add the following nested class to the **App** class:

```

// Handler for direct method notifications from IoT Hub
protected static class DirectMethodStatusCallback implements IoTHubEventCallback {
    public void execute(IoTHubStatusCode status, Object context) {
        System.out.println("IoT Hub responded to direct method operation with status " + status.name());
    }
}

```

11. To handle direct method calls from IoT Hub, add the following nested class to the **App** class:

```

// Handler for direct method calls from IoT Hub
protected static class DirectMethodCallback
    implements DeviceMethodCallback {
    @Override
    public DeviceMethodData call(String methodName, Object methodData, Object context) {
        DeviceMethodData deviceMethodData;
        switch (methodName) {
            case "lockDoor": {
                System.out.println("Executing direct method: " + methodName);
                deviceMethodData = new DeviceMethodData(METHOD_SUCCESS, "Executed direct method " +
methodName);
                break;
            }
            default: {
                deviceMethodData = new DeviceMethodData(METHOD_NOT_DEFINED, "Not defined direct method " +
methodName);
            }
        }
        // Notify IoT Hub of result
        return deviceMethodData;
    }
}

```

12. Update the **main** method signature to include the following `throws` clause:

```
public static void main( String[] args ) throws IOException, URISyntaxException
```

13. Add the following code to the **main** method to:

- Create a device client to communicate with IoT Hub.
- Create a **Device** object to store the device twin properties.

```

// Create a device client
DeviceClient client = new DeviceClient(connString, protocol);

// An object to manage device twin desired and reported properties
Device dataCollector = new Device() {
    @Override
    public void PropertyCall(String propertyKey, Object PropertyValue, Object context)
    {
        System.out.println("Received desired property change: " + propertyKey + " " + PropertyValue);
    }
};

```

14. To start the device client services, add the following code to the **main** method:

```

try {
    // Open the DeviceClient
    // Start the device twin services
    // Subscribe to direct method calls
    client.open();
    client.startDeviceTwin(new DeviceTwinStatusCallBack(), null, dataCollector, null);
    client.subscribeToDeviceMethod(new DirectMethodCallback(), null, new DirectMethodStatusCallback(),
null);
} catch (Exception e) {
    System.out.println("Exception, shutting down \n" + " Cause: " + e.getCause() + " \n" +
e.getMessage());
    dataCollector.clean();
    client.closeNow();
    System.out.println("Shutting down...");
}

```

15. To wait for the user to press the **Enter** key before shutting down, add the following code to the end of the **main** method:

```

// Close the app
System.out.println("Press any key to exit...");
Scanner scanner = new Scanner(System.in);
scanner.nextLine();
dataCollector.clean();
client.closeNow();
scanner.close();

```

16. Save and close the `simulated-device\src\main\java\com\mycompany\app\App.java` file.

17. Build the **simulated-device** app and correct any errors. At your command prompt, navigate to the `simulated-device` folder and run the following command:

```
mvn clean package -DskipTests
```

Run the apps

You are now ready to run the console apps.

- At a command prompt in the `simulated-device` folder, run the following command to start the device app listening for desired property changes and direct method calls:

```
mvn exec:java -Dexec.mainClass="com.mycompany.app.App"
```

```
c:\repos\samples\iot-java-schedule-jobs\simulated-device>mvn exec:java -Dexec.mainClass="com.mycompany.app.App"
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building simulated-device 1.0-SNAPSHOT
[INFO] -----
[INFO] --- exec-maven-plugin:1.4.0:java (default-cli) @ simulated-device ---
log4j:WARN No appenders could be found for logger (com.microsoft.azure.sdk.iot.device.IotHubConnectionString).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
IoT Hub responded to device twin operation with status OK_EMPTY
Press any key to exit...
IoT Hub responded to direct method operation with status OK_EMPTY
IoT Hub responded to device twin operation with status OK
```

- At a command prompt in the `schedule-jobs` folder, run the following command to run the **schedule-jobs** service app to run two jobs. The first sets the desired property values, the second calls the direct method:

```
mvn exec:java -Dexec.mainClass="com.mycompany.app.App"
```

```
c:\repos\samples\iot-java-schedule-jobs\schedule-jobs>
Status completed for job DPCMDcf2ff5d-b990-46d7-9c33-e1a6fc2ac9af
Final status completed for job DPCMDcf2ff5d-b990-46d7-9c33-e1a6fc2ac9af
Schedule job DMCMD49d40cc3-f3ac-49c3-82bb-4c2371d9afe9 for device myFirstJavaDevice
Status running for job DMCMD49d40cc3-f3ac-49c3-82bb-4c2371d9afe9
Status completed for job DMCMD49d40cc3-f3ac-49c3-82bb-4c2371d9afe9
Final status completed for job DMCMD49d40cc3-f3ac-49c3-82bb-4c2371d9afe9
Shutting down schedule-jobs app
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 12.081 s
[INFO] Finished at: 2017-07-11T14:23:46+01:00
[INFO] Final Memory: 14M/363M
[INFO] -----
```

- The device app handles the desired property change and the direct method call:



```
c:\repos\samples\iot-java-schedule-jobs\simulated-device>mvn exec:java -Dexec.mainClass="com.mycompany.app.App"
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building simulated-device 1.0-SNAPSHOT
[INFO] -----
[INFO] [INFO] --- exec-maven-plugin:1.4.0:java (default-cli) @ simulated-device ---
[INFO] log4j:WARN No appenders could be found for logger (com.microsoft.azure.sdk.iot.device.IotHubConnectionString).
[INFO] log4j:WARN Please initialize the log4j system properly.
[INFO] log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
[INFO] IoT Hub responded to device twin operation with status OK_EMPTY
[INFO] Press any key to exit...
[INFO] IoT Hub responded to direct method operation with status OK_EMPTY
[INFO] IoT Hub responded to device twin operation with status OK
[INFO] Received desired property change: Building 43.0
[INFO] Received desired property change: Floor 3.0
[INFO] Executing direct method: lockDoor
[INFO] IoT Hub responded to direct method operation with status OK_EMPTY
```

Next steps

In this tutorial, you configured a new IoT hub in the Azure portal, and then created a device identity in the IoT hub's identity registry. You created a back-end app to run two jobs. The first job set desired property values, and the second job called a direct method.

Use the following resources to learn how to:

- Send telemetry from devices with the [Get started with IoT Hub](#) tutorial.
- Control devices interactively (such as turning on a fan from a user-controlled app) with the [Use direct methods](#) tutorial.s

Schedule and broadcast jobs (Python)

3/6/2019 • 10 minutes to read

Azure IoT Hub is a fully managed service that enables a back-end app to create and track jobs that schedule and update millions of devices. Jobs can be used for the following actions:

- Update desired properties
- Update tags
- Invoke direct methods

Conceptually, a job wraps one of these actions and tracks the progress of execution against a set of devices, which is defined by a device twin query. For example, a back-end app can use a job to invoke a reboot method on 10,000 devices, specified by a device twin query and scheduled at a future time. That application can then track progress as each of those devices receive and execute the reboot method.

Learn more about each of these capabilities in these articles:

- Device twin and properties: [Get started with device twins](#) and [Tutorial: How to use device twin properties](#)
- Direct methods: [IoT Hub developer guide - direct methods](#) and [Tutorial: direct methods](#)

NOTE

The features described in this article are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

This tutorial shows you how to:

- Create a Python simulated device app that has a direct method, which enables **lockDoor**, which can be called by the solution back end.
- Create a Python console app that calls the **lockDoor** direct method in the simulated device app using a job and updates the desired properties using a device job.

At the end of this tutorial, you have two Python apps:

simDevice.py, which connects to your IoT hub with the device identity and receives a **lockDoor** direct method.

scheduleJobService.py, which calls a direct method in the simulated device app and updates the device twin's desired properties using a job.

To complete this tutorial, you need the following:

- [Python 2.x or 3.x](#). Make sure to use the 32-bit or 64-bit installation as required by your setup. When prompted during the installation, make sure to add Python to your platform-specific environment variable. If you are using Python 2.x, you may need to [install or upgrade pip, the Python package management system](#).
- If you are using Windows OS, then [Visual C++ redistributable package](#) to allow the use of native DLLs from Python.
- An active Azure account. (If you don't have an account, you can create a [free account](#) in just a couple of minutes.)

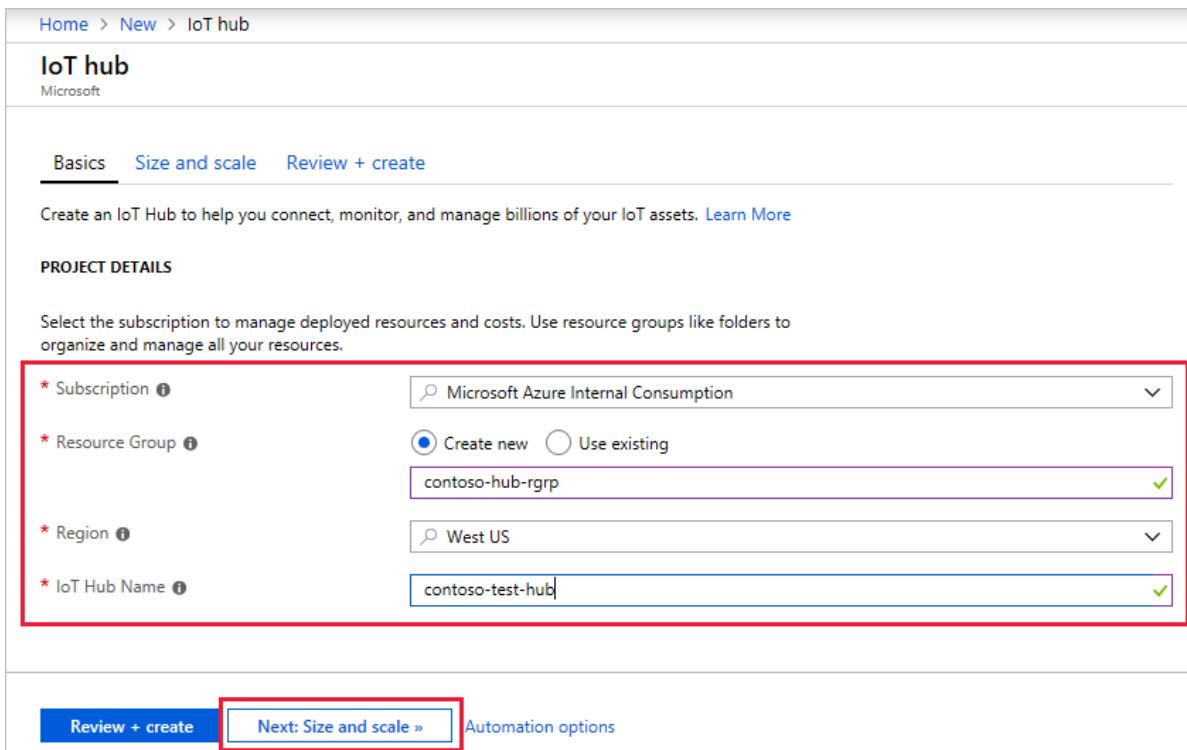
NOTE

The **Azure IoT SDK for Python** does not directly support **Jobs** functionality. Instead this tutorial offers an alternate solution utilizing asynchronous threads and timers. For further updates, see the **Service Client SDK** feature list on the [Azure IoT SDK for Python](#) page.

Create an IoT hub

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose +**Create a resource**, then choose **Internet of Things**.
3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.



Fill in the fields.

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

The screenshot shows the 'Size and scale' configuration step for creating an IoT hub. At the top, there are tabs for 'Basics', 'Size and scale' (which is selected), and 'Review + create'. A note below the tabs states: 'Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)'.

SCALE TIER AND UNITS

* Pricing and scale tier: S1: Standard tier. A link to 'Learn how to choose the right IoT Hub tier for your solution' is provided.

Number of S1 IoT Hub units: 1. A note below the slider says: 'This determines your IoT Hub scale capability and can be changed as your need increases.'

Enabled Features:

- Device-to-cloud-messages: Enabled
- Message routing: Enabled
- Cloud-to-device commands: Enabled
- IoT Edge: Enabled
- Device management: Enabled

Advanced Settings:

Device-to-cloud partitions: 4. A note below the slider says: 'This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most IoT hubs only need four partitions.'

At the bottom, there are buttons for 'Review + create', '<< Previous: Basics', and 'Automation options'.

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: The number of messages allowed per unit per day depends on your hub's pricing tier. For example, if you want the IoT hub to support ingress of 700,000 messages, you choose two S1 tier units.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

Advanced / Device-to-cloud partitions: This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most IoT hubs only need four partitions.

5. Click **Review + create** to review your choices. You see something similar to this screen.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale **Review + create**

BASICS

Subscription	Microsoft Azure Internal Consumption
Resource Group	contoso-hub-rgrp
Region	West US
IoT Hub Name	contoso-test-hub

SIZE AND SCALE

Pricing and scale tier	S1
Number of S1 IoT Hub units	1
Messages per day	400,000
Cost per month	25.00 USD

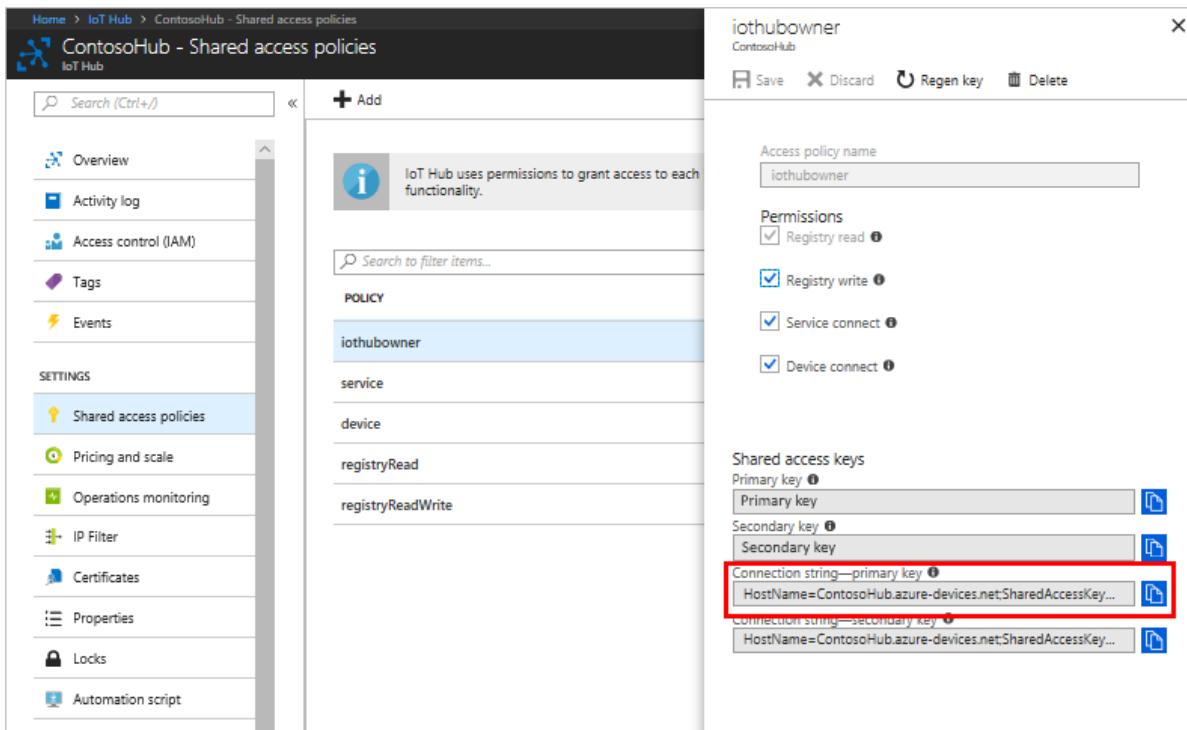
Create « Previous: Size and scale Automation options

6. Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Retrieve connection string for IoT hub

After your hub has been created, retrieve the connection string for the hub. This is used to connect devices and applications to your hub.

1. Click on your hub to see the IoT Hub pane with Settings, and so on. Click **Shared access policies**.
2. In **Shared access policies**, select the **iothubowner** policy.
3. Under **Shared access keys**, copy the **Connection string -- primary key** to be used later.



Home > IoT Hub > ContosoHub - Shared access policies

ContosoHub - Shared access policies

IoT Hub

Overview Activity log Access control (IAM) Tags Events

SETTINGS

- Shared access policies **Selected**
- Pricing and scale
- Operations monitoring
- IP Filter
- Certificates
- Properties
- Locks
- Automation script

Add

IoT Hub uses permissions to grant access to each functionality.

Search to filter items...

POLICY

iothubowner

service

device

registryRead

registryReadWrite

iothubowner

Save Discard Regen key Delete

Access policy name: iothubowner

Permissions:

- Registry read
- Registry write
- Service connect
- Device connect

Shared access keys:

Primary key: **HostName=ContosoHub.azure-devices.net;SharedAccessKey...**

Secondary key: **HostName=ContosoHub.azure-devices.net;SharedAccessKey...**

Connection string—primary key: **HostName=ContosoHub.azure-devices.net;SharedAccessKey...**

Connection string—secondary key: **HostName=ContosoHub.azure-devices.net;SharedAccessKey...**

For more information, see [Access control](#) in the "IoT Hub developer guide."

Register a new device in the IoT hub

In this section, you create a device identity in the identity registry in your IoT hub. A device cannot connect to IoT hub unless it has an entry in the identity registry. For more information, see the "Identity registry" section of the [IoT Hub developer guide](#)

1. In your IoT hub navigation menu, open **IoT Devices**, then click **Add** to register a new device in your IoT hub.

The screenshot shows the 'ContosoHub - IoT devices' blade in the Azure portal. On the left, there's a navigation menu with items like 'Events', 'Shared access policies', 'Pricing and scale', 'Operations monitoring', 'IP Filter', 'Certificates', 'Properties', 'Locks', and 'Automation script'. The 'IoT devices' item is highlighted with a red box. At the top, there's a search bar, a '+ Add' button (also highlighted with a red box), a 'Refresh' button, and a 'Delete' button. The main area has a message: 'You can use this tool to view, create, update, and delete devices on your IoT Hub.' Below it is a 'Query' section with a text input field containing 'optional (e.g. tags.location='US')' and an 'Execute' button. A table below shows columns for DEVICE ID, STATUS, LAST ACTIVITY, LAST STATUS..., AUTHENTICATI..., and CLOUD TO DEV... with a single row labeled 'No results'.

2. Provide a name for your new device, such as **myDeviceId**, and click **Save**. This action creates a new device identity for your IoT hub.



Create a device

□ X



Learn more about creating devices



* Device ID i

myDeviceId



Authentication type i

Symmetric key

X.509 Self-Signed

X.509 CA Signed

* Primary key i

Enter your primary key

* Secondary key i

Enter your secondary key

Auto-generate keys i



Connect this device to an IoT hub i

Enable

Disable

Parent device (Preview) i

No parent device

[Set a parent device](#)

[Save](#)

IMPORTANT

The device ID may be visible in the logs collected for customer support and troubleshooting, so make sure to avoid any sensitive information while naming it.

3. After the device is created, open the device from the list in the **IoT devices** pane. Copy the **Connection string---primary key** to use later.

The screenshot shows the 'Device details' page for a device named 'myDeviceId'. It includes fields for Device Id, Primary key, Secondary key, Connection string (primary key), and Connection string (secondary key). The 'Connection string (primary key)' field contains the value 'HostName=ContosoHub.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=<Primary Key>'. The 'Connection string (primary key)' and 'Connection string (secondary key)' fields are highlighted with a red border. Below these fields are buttons for 'Enable' and 'Disable' access, and a section for setting a parent device.

NOTE

The IoT Hub identity registry only stores device identities to enable secure access to the IoT hub. It stores device IDs and keys to use as security credentials, and an enabled/disabled flag that you can use to disable access for an individual device. If your application needs to store other device-specific metadata, it should use an application-specific store. For more information, see [IoT Hub developer guide](#).

Create a simulated device app

In this section, you create a Python console app that responds to a direct method called by the cloud, which triggers a simulated **lockDoor** method.

1. At your command prompt, run the following command to install the **azure-iot-device-client** package:

```
pip install azure-iotdevice-client
```

2. Using a text editor, create a new **simDevice.py** file in your working directory.

3. Add the following `import` statements and variables at the start of the **simDevice.py** file. Replace `deviceConnectionString` with the connection string of the device you created above:

```
import time
import sys

import iothub_client
from iothub_client import IoTHubClient, IoTHubClientError, IoTHubTransportProvider, IoTHubClientResult
from iothub_client import IoTHubError, DeviceMethodReturnValue

METHOD_CONTEXT = 0
TWIN_CONTEXT = 0
WAIT_COUNT = 10

PROTOCOL = IoTHubTransportProvider.MQTT
CONNECTION_STRING = "{deviceConnectionString}"
```

4. Add the following function callback to handle the **lockDoor** method:

```

def device_method_callback(method_name, payload, user_context):
    if method_name == "lockDoor":
        print ( "Locking Door!" )

    device_method_return_value = DeviceMethodReturnValue()
    device_method_return_value.response = "{ \"Response\": \"lockDoor called successfully\" }"
    device_method_return_value.status = 200
    return device_method_return_value

```

5. Add another function callback to handle device twins updates:

```

def device_twin_callback(update_state, payload, user_context):
    print ( "" )
    print ( "Twin callback called with:" )
    print ( "payload: %s" % payload )

```

6. Add the following code to register the handler for the **lockDoor** method. Also include the `main` routine:

```

def iothub_jobs_sample_run():
    try:
        client = IoTHubClient(CONNECTION_STRING, PROTOCOL)
        client.set_device_method_callback(device_method_callback, METHOD_CONTEXT)
        client.set_device_twin_callback(device_twin_callback, TWIN_CONTEXT)

        print ( "Direct method initialized." )
        print ( "Device twin callback initialized." )
        print ( "IoTHubClient waiting for commands, press Ctrl-C to exit" )

        while True:
            status_counter = 0
            while status_counter <= WAIT_COUNT:
                time.sleep(10)
                status_counter += 1

    except IoTHubError as iothub_error:
        print ( "Unexpected error %s from IoTHub" % iothub_error )
        return
    except KeyboardInterrupt:
        print ( "IoTHubClient sample stopped" )

    if __name__ == '__main__':
        print ( "Starting the IoT Hub Python jobs sample..." )
        print ( "    Protocol %s" % PROTOCOL )
        print ( "    Connection string=%s" % CONNECTION_STRING )

    iothub_jobs_sample_run()

```

7. Save and close the **simDevice.py** file.

NOTE

To keep things simple, this tutorial does not implement any retry policy. In production code, you should implement retry policies (such as an exponential backoff), as suggested in the article, [Transient Fault Handling](#).

Schedule jobs for calling a direct method and updating a device twin's properties

In this section, you create a Python console app that initiates a remote **lockDoor** on a device using a direct method and update the device twin's properties.

1. At your command prompt, run the following command to install the **azure-iot-service-client** package:

```
pip install azure-iothub-service-client
```

2. Using a text editor, create a new **scheduleJobService.py** file in your working directory.

3. Add the following `import` statements and variables at the start of the **scheduleJobService.py** file:

```
import sys
import time
import threading
import uuid

import iothub_service_client
from iothub_service_client import IoTHubRegistryManager, IoTHubRegistryManagerAuthMethod
from iothub_service_client import IoTHubDeviceTwin, IoTHubDeviceMethod, IoTHubError

CONNECTION_STRING = "{IoTHubConnectionString}"
DEVICE_ID = "{deviceId}"

METHOD_NAME = "lockDoor"
METHOD_PAYLOAD = "{\"lockTime\":\"10m\"}"
UPDATE_JSON = "{\"properties\":{\"desired\":{\"building\":43,\"floor\":3}}}"
TIMEOUT = 60
WAIT_COUNT = 5
```

4. Add the following function that is used to query for devices:

```
def query_condition(device_id):
    iothub_registry_manager = IoTHubRegistryManager(CONNECTION_STRING)

    number_of_devices = 10
    dev_list = iothub_registry_manager.get_device_list(number_of_devices)

    for device in range(0, number_of_devices):
        if dev_list[device].deviceId == device_id:
            return 1

    print ( "Device not found" )
    return 0
```

5. Add the following methods to run the jobs that call the direct method and device twin:

```
def device_method_job(job_id, device_id, wait_time, execution_time):
    print ( "" )
    print ( "Scheduling job: " + str(job_id) )
    time.sleep(wait_time)

    if query_condition(device_id):
        iothub_device_method = IoTHubDeviceMethod(CONNECTION_STRING)

        response = iothub_device_method.invoke(device_id, METHOD_NAME, METHOD_PAYLOAD, TIMEOUT)

        print ( "" )
        print ( "Direct method " + METHOD_NAME + " called." )

def device_twin_job(job_id, device_id, wait_time, execution_time):
    print ( "" )
    print ( "Scheduling job " + str(job_id) )
    time.sleep(wait_time)

    if query_condition(device_id):
        iothub_twin_method = IoTHubDeviceTwin(CONNECTION_STRING)

        twin_info = iothub_twin_method.update_twin(DEVICE_ID, UPDATE_JSON)

        print ( "" )
        print ( "Device twin updated." )
```

6. Add the following code to schedule the jobs and update job status. Also include the `main` routine:

```

def iothub_jobs_sample_run():
    try:
        method_thr_id = uuid.uuid4()
        method_thr = threading.Thread(target=device_method_job, args=(method_thr_id, DEVICE_ID, 20, TIMEOUT), kwargs={})
        method_thr.start()

        print ( "" )
        print ( "Direct method called with Job Id: " + str(method_thr_id) )

        twin_thr_id = uuid.uuid4()
        twin_thr = threading.Thread(target=device_twin_job, args=(twin_thr_id, DEVICE_ID, 10, TIMEOUT), kwargs={})
        twin_thr.start()

        print ( "" )
        print ( "Device twin called with Job Id: " + str(twin_thr_id) )

    while True:
        print ( "" )

        if method_thr.is_alive():
            print ( "...job " + str(method_thr_id) + " still running." )
        else:
            print ( "...job " + str(method_thr_id) + " complete." )

        if twin_thr.is_alive():
            print ( "...job " + str(twin_thr_id) + " still running." )
        else:
            print ( "...job " + str(twin_thr_id) + " complete." )

        print ( "Job status posted, press Ctrl-C to exit" )

        status_counter = 0
        while status_counter <= WAIT_COUNT:
            time.sleep(1)
            status_counter += 1

    except IoTHubError as iothub_error:
        print ( "" )
        print ( "Unexpected error {0}" % iothub_error )
        return
    except KeyboardInterrupt:
        print ( "" )
        print ( "IoTHubService sample stopped" )

if __name__ == '__main__':
    print ( "Starting the IoT Hub jobs Python sample..." )
    print ( "    Connection string = {0}".format(CONNECTION_STRING) )
    print ( "    Device ID         = {0}".format(DEVICE_ID) )

    iothub_jobs_sample_run()

```

7. Save and close the **scheduleJobService.py** file.

Run the applications

You are now ready to run the applications.

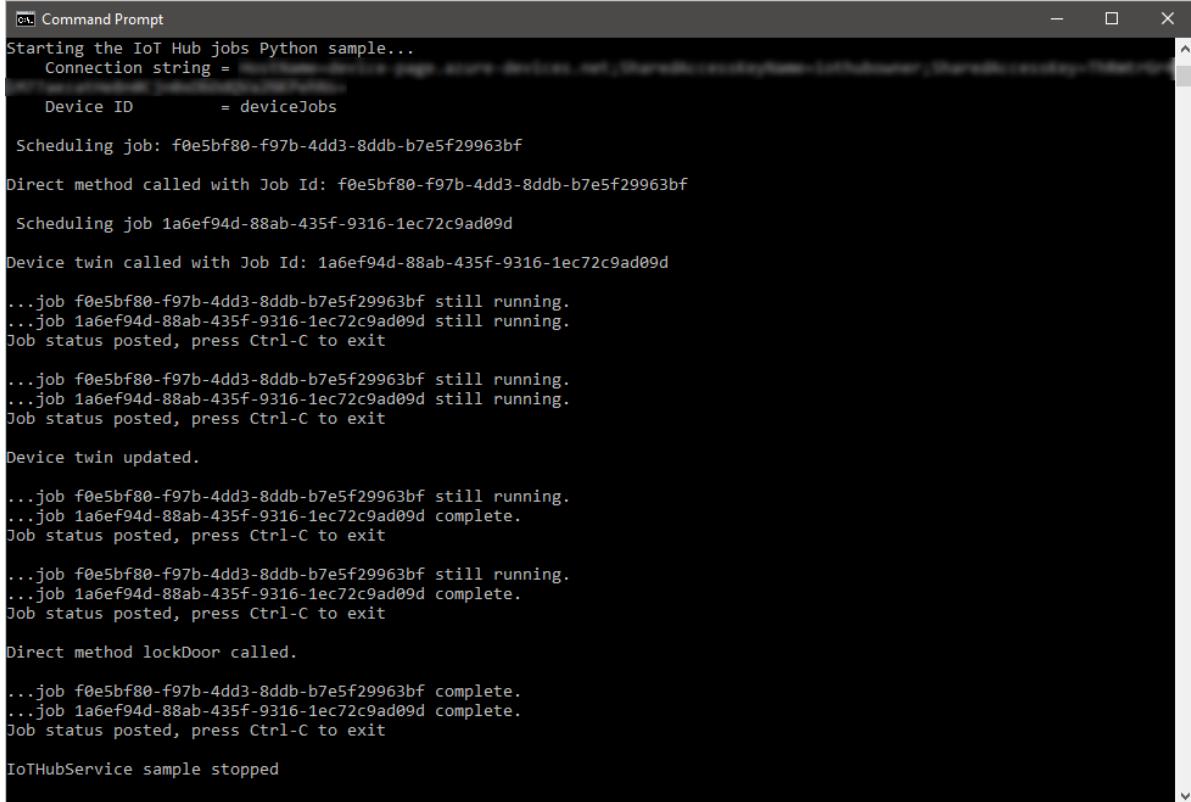
- At the command prompt in your working directory, run the following command to begin listening for the reboot direct method:

```
python simDevice.py
```

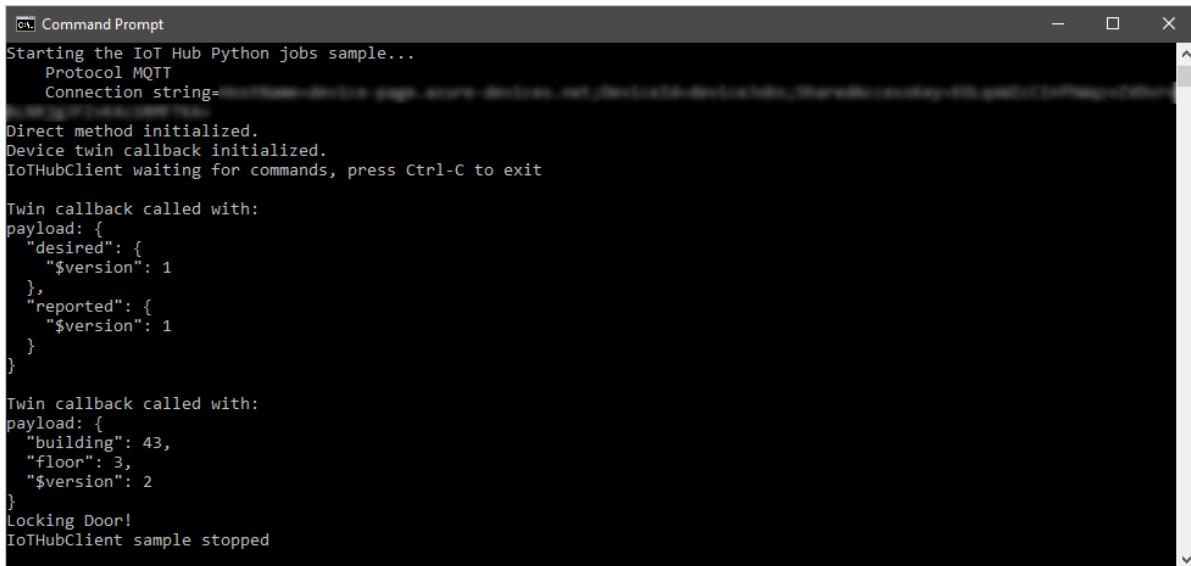
2. At another command prompt in your working directory, run the following command to trigger the jobs to lock the door and update the twin:

```
python scheduleJobService.py
```

3. You see the device responses to the direct method and device twins update in the console.



```
Starting the IoT Hub jobs Python sample...
Connection string =
Device ID      = deviceJobs
Scheduling job: f0e5bf80-f97b-4dd3-8ddb-b7e5f29963bf
Direct method called with Job Id: f0e5bf80-f97b-4dd3-8ddb-b7e5f29963bf
Scheduling job 1a6ef94d-88ab-435f-9316-1ec72c9ad09d
Device twin called with Job Id: 1a6ef94d-88ab-435f-9316-1ec72c9ad09d
...job f0e5bf80-f97b-4dd3-8ddb-b7e5f29963bf still running.
...job 1a6ef94d-88ab-435f-9316-1ec72c9ad09d still running.
Job status posted, press Ctrl-C to exit
...job f0e5bf80-f97b-4dd3-8ddb-b7e5f29963bf still running.
...job 1a6ef94d-88ab-435f-9316-1ec72c9ad09d still running.
Job status posted, press Ctrl-C to exit
Device twin updated.
...job f0e5bf80-f97b-4dd3-8ddb-b7e5f29963bf still running.
...job 1a6ef94d-88ab-435f-9316-1ec72c9ad09d complete.
Job status posted, press Ctrl-C to exit
...job f0e5bf80-f97b-4dd3-8ddb-b7e5f29963bf still running.
...job 1a6ef94d-88ab-435f-9316-1ec72c9ad09d complete.
Job status posted, press Ctrl-C to exit
Direct method lockDoor called.
...job f0e5bf80-f97b-4dd3-8ddb-b7e5f29963bf complete.
...job 1a6ef94d-88ab-435f-9316-1ec72c9ad09d complete.
Job status posted, press Ctrl-C to exit
IoTHubService sample stopped
```



```
Starting the IoT Hub Python jobs sample...
Protocol MQTT
Connection string=
Direct method initialized.
Device twin callback initialized.
IoTHubClient waiting for commands, press Ctrl-C to exit
Twin callback called with:
payload: {
  "desired": {
    "$version": 1
  },
  "reported": {
    "$version": 1
  }
}
Twin callback called with:
payload: {
  "building": 43,
  "floor": 3,
  "$version": 2
}
Locking Door!
IoTHubClient sample stopped
```

Next steps

In this tutorial, you used a job to schedule a direct method to a device and the update of the device twin's properties.

To continue getting started with IoT Hub and device management patterns such as remote over the air firmware update, see [How to do a firmware update](#).

Create an IoT hub using the Azure portal

11/9/2018 • 6 minutes to read

This article describes how to create and manage IoT hubs using the [Azure portal](#).

To use the steps in this tutorial, you need an Azure subscription. If you don't have an Azure subscription, create a [free account](#) before you begin.

Create an IoT hub

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose +**Create a resource**, then choose **Internet of Things**.
3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.

The screenshot shows the 'Basics' step of the IoT Hub creation wizard. The 'PROJECT DETAILS' section is highlighted with a red box. It includes fields for Subscription (selected: Microsoft Azure Internal Consumption), Resource Group (selected: Create new, value: contoso-hub-rgrp), Region (selected: West US), and IoT Hub Name (value: contoso-test-hub). At the bottom, there are 'Review + create' and 'Next: Size and scale >' buttons, with 'Next: Size and scale >' being the one highlighted by a red box.

Fill in the fields.

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

The screenshot shows the 'Size and scale' configuration step for creating an IoT hub. At the top, there are three tabs: 'Basics' (selected), 'Size and scale' (underlined), and 'Review + create'. A note below the tabs states: 'Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)'.

SCALE TIER AND UNITS

* Pricing and scale tier: A dropdown menu is set to 'S1: Standard tier'. Below it is a link: 'Learn how to choose the right IoT Hub tier for your solution'.

Number of S1 IoT Hub units: A slider is set to '1'. Below the slider is a note: 'This determines your IoT Hub scale capability and can be changed as your need increases.'

Enabled Features

- Device-to-cloud-messages: Enabled
- Message routing: Enabled
- Cloud-to-device commands: Enabled
- IoT Edge: Enabled
- Device management: Enabled

Advanced Settings

Device-to-cloud partitions: A slider is set to '4'.

At the bottom, there are three buttons: 'Review + create' (highlighted in blue), '< Previous: Basics', and 'Automation options'.

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: The number of messages allowed per unit per day depends on your hub's pricing tier. For example, if you want the IoT hub to support ingress of 700,000 messages, you choose two S1 tier units.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

Advanced / Device-to-cloud partitions: This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most IoT hubs only need four partitions.

5. Click **Review + create** to review your choices. You see something similar to this screen.

The screenshot shows the 'Review + create' step of the IoT hub creation wizard. It displays the following configuration details:

Setting	Value
Subscription	Microsoft Azure Internal Consumption
Resource Group	contoso-hub-rgrp
Region	West US
IoT Hub Name	contoso-test-hub
Pricing and scale tier	S1
Number of S1 IoT Hub units	1
Messages per day	400,000
Cost per month	25.00 USD

At the bottom, there are three buttons: 'Create' (highlighted with a red box), '< Previous: Size and scale', and 'Automation options'.

6. Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Change the settings of the IoT hub

You can change the settings of an existing IoT hub after it's created from the IoT Hub pane.

The screenshot shows the 'Settings' pane for an existing IoT hub. The left sidebar lists various settings categories:

- Settings
 - Shared access policies
 - Pricing and scale (highlighted with a blue box)
 - Operations monitoring
 - IP Filter
 - Certificates
 - Built-in endpoints
 - Properties
 - Locks
 - Automation script
- Explorers
 - Query explorer
 - IoT devices
- Automatic Device Management
 - IoT Edge
 - IoT device configuration
- Messaging
 - File upload
 - Message routing

The main pane displays the following properties for the IoT hub:

Property	Value
Resource group (change)	contoso-hub-rgrp
Status	Active
Location	West US
Subscription (change)	Microsoft Azure Internal Consumption
Subscription ID	<Your Subscription ID>

Below the properties, there are two promotional callouts:

- Need a way to provision millions of devices?**
IoT Hub Device Provisioning Service enables zero-touch, just-in-time provisioning to the right IoT hub without requiring human intervention.
- Want to learn more about IoT Hub?**
Check out IoT Hub documentation. Learn how to use IoT Hub to connect, monitor, and control billions of Internet of Things assets.

Here are some of the properties you can set for an IoT hub:

Pricing and scale: You can use this property to migrate to a different tier or set the number of IoT Hub units.

Operations monitoring: Turn the different monitoring categories on or off, such as logging for events related to device-to-cloud messages or cloud-to-device messages.

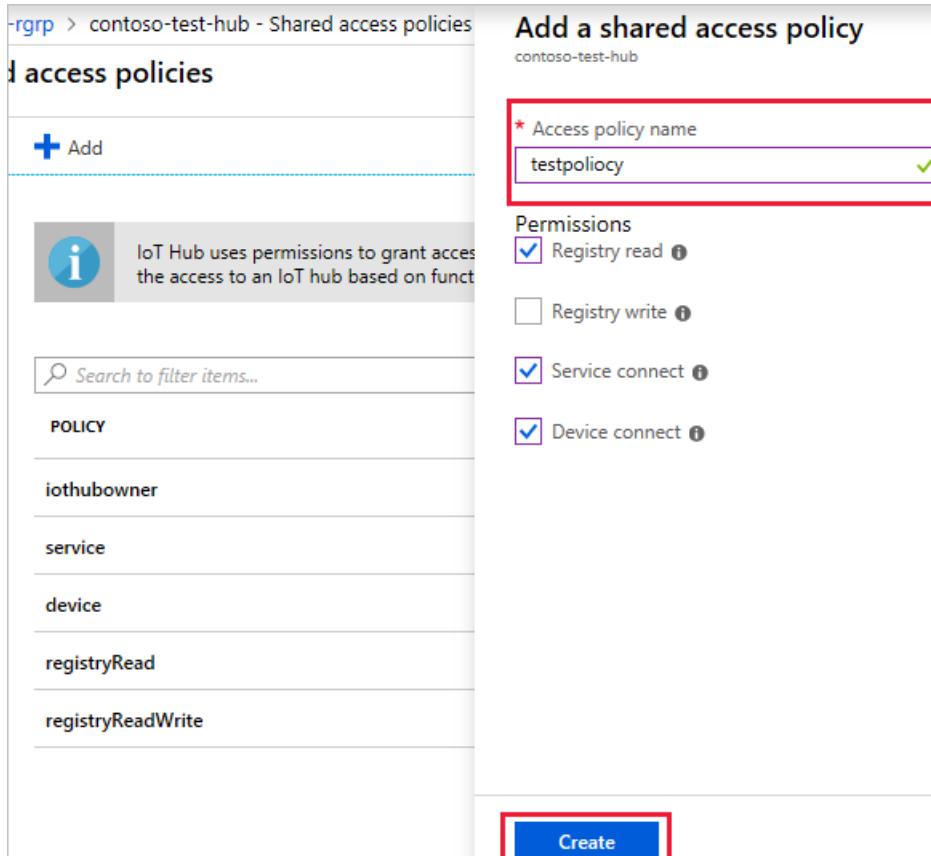
IP Filter: Specify a range of IP addresses that will be accepted or rejected by the IoT hub.

Properties: Provides the list of properties that you can copy and use elsewhere, such as the resource ID, resource group, location, and so on.

Shared access policies

You can also view or modify the list of shared access policies by clicking **Shared access policies** in the **Settings** section. These policies define the permissions for devices and services to connect to IoT Hub.

Click **Add** to open the **Add a shared access policy** blade. You can enter the new policy name and the permissions that you want to associate with this policy, as shown in the following figure:



- The **Registry read** and **Registry write** policies grant read and write access rights to the identity registry. Choosing the write option automatically chooses the read option.
- The **Service connect** policy grants permission to access service endpoints such as **Receive device-to-cloud**.
- The **Device connect** policy grants permissions for sending and receiving messages using the IoT Hub device-side endpoints.

Click **Create** to add this newly created policy to the existing list.

Message Routing for an IoT hub

Click **Message Routing** under **Messaging** to see the Message Routing pane, where you define routes and custom endpoints for the hub. **Message routing** enables you to manage how data is sent from your devices to your endpoints. The first step is to add a new route. Then you can add an existing endpoint to the route, or create a new one of the types supported, such as blob storage.

Send data from your devices to endpoints that you choose.

Routes [Custom endpoints](#)

Create an endpoint, and then add a route (you can add up to 100 from each IoT hub). Messages that don't match a query are automatically sent to messages/events if you've enabled the fallback route.

[Disable fallback route](#)

+ Add [Test all routes](#) [Delete](#)

<input checked="" type="checkbox"/> NAME	DATA SOURCE	ROUTING QUERY	ENDPOINT	ENABLED
No results				

Routes

Routes is the first tab on the Message Routing pane. To add a new route, click **+Add**. You see the following screen.

Add a route

* Name [?](#)

* Endpoint [?](#)

* Data source [?](#)
 [?](#)

* Enable route [?](#)

Create a query to filter messages before data is routed to an endpoint. [Learn more](#)

Routing query [?](#)

Name your hub. The name must be unique within the list of routes for that hub.

For **Endpoint**, you can select one from the dropdown list, or add a new one. In this example, a storage account and container are already available. To add them as an endpoint, click **+Add** next to the Endpoint dropdown and select **Blob Storage**. The following screen shows where the storage account and container are specified.



Add a storage endpoint

Route your telemetry and device messages to Azure Storage as blobs.

* Endpoint name ?

Azure Storage account and container

Create a new container, or choose an existing one that shares a subscription with this IoT hub.

Azure Storage container

Pick a container

Batch frequency ?



Chunk size window ?



* Blob file name format ?

The format must contain {iothub}, {partition}, {YYYY}, {MM}, {DD}, {HH} and {mm} in any order.

If multiple files are created within the same minute, the filename format would be contoso-test-hub/0/2018/09/13/10/44-01.

Click **Pick a container** to select the storage account and container. When you have selected those fields, it returns to the Endpoint pane. Use the defaults for the rest of the fields and **Create** to create the endpoint for the storage account and add it to the routing rules.

For **Data source**, select Device Telemetry Messages.

Next, add a routing query. In this example, the messages that have an application property called `level` with a value equal to `critical` are routed to the storage account.

 Add a route

* Name [?](#)
storage-route ✓

* Endpoint [?](#)
storage-ep ▼ 

* Data source [?](#)
Device Telemetry Messages ▼

* Enable route [?](#)
 Enable Disable

Create a query to filter messages before data is routed to an endpoint. [Learn more](#)

Routing query [?](#)
1 level="critical"

▼ Test

Save

Click **Save** to save the routing rule. You return to the Message Routing pane, and your new routing rule is displayed.

Custom endpoints

Click the **Custom endpoints** tab. You see any custom endpoints already created. From here, you can add new endpoints or delete existing endpoints.

NOTE

If you delete a route, it does not delete the endpoints assigned to that route. To delete an endpoint, click the Custom endpoints tab, select the endpoint you want to delete, and click Delete.

You can read more about custom endpoints in [Reference - IoT hub endpoints](#).

You can define up to 10 custom endpoints for an IoT hub.

To see a full example of how to use custom endpoints with routing, see [Message routing with IoT Hub](#).

Find a specific IoT hub

Here are two ways to find a specific IoT hub in your subscription:

1. If you know the resource group to which the IoT hub belongs, click **Resource groups**, then select the resource group from the list. The resource group screen shows all of the resources in that group, including the IoT hubs. Click on the hub for which you're looking.
2. Click **All resources**. On the **All resources** pane, there is a dropdown list that defaults to **All types**. Click on the dropdown list, uncheck **Select all**. Find **IoT Hub** and check it. Click on the dropdown list box to close it, and the entries will be filtered, showing only your IoT hubs.

Delete the IoT hub

To delete an IoT hub, find the IoT hub you want to delete, then click the **Delete** button below the IoT hub name.

Next steps

Follow these links to learn more about managing Azure IoT Hub:

- [Message routing with IoT Hub](#)
- [IoT Hub metrics](#)
- [Operations monitoring](#)

Create an IoT hub using the Azure IoT Tools for Visual Studio Code

2/22/2019 • 2 minutes to read

This article shows you how to use the [Azure IoT Tools for Visual Studio Code](#) to create an Azure IoT hub.

NOTE

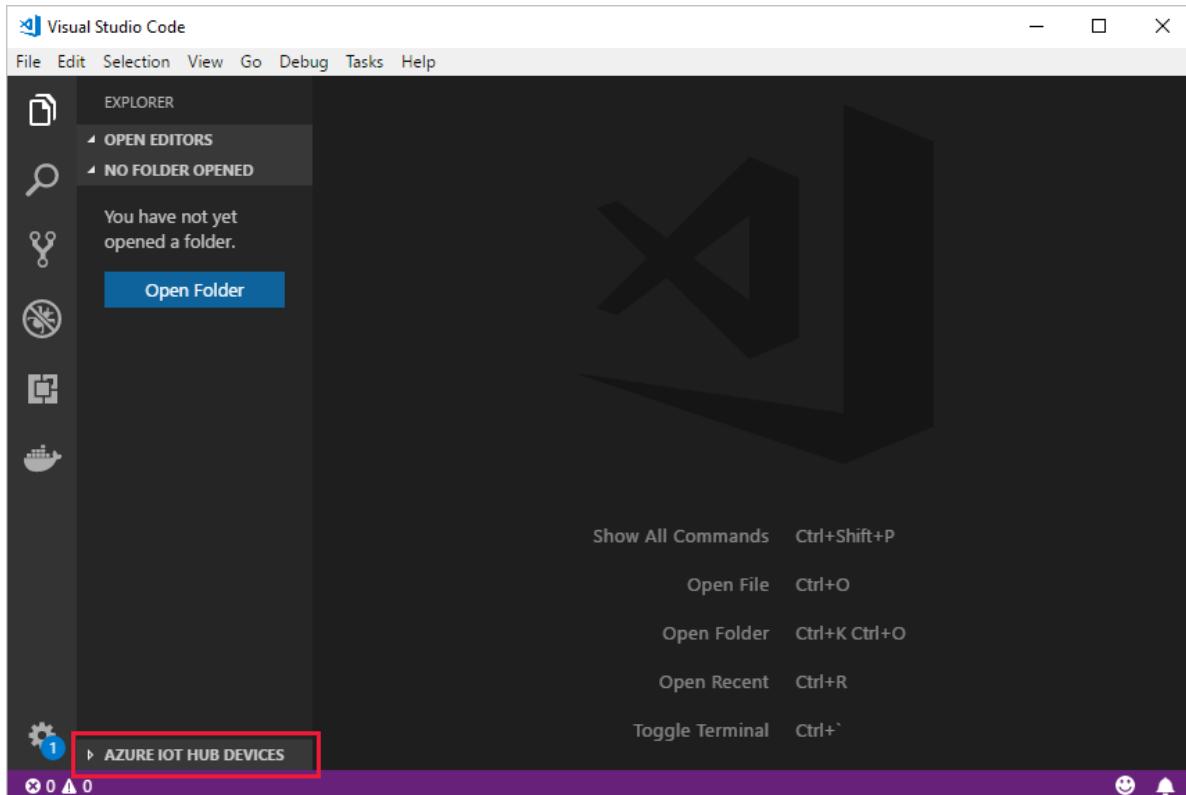
This article has been updated to use the new Azure PowerShell Az module. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For installation instructions, see [Install Azure PowerShell](#).

To complete this article, you need the following:

- An Azure subscription. If you don't have an Azure subscription, create a [free account](#) before you begin.
- [Visual Studio Code](#)
- [Azure IoT Tools](#) for Visual Studio Code.

Create an IoT hub

1. In Visual Studio Code, open the **Explorer** view.
2. At the bottom of the Explorer, expand the **Azure IoT Hub Devices** section.



3. Click on the ... in the **Azure IoT Hub Devices** section header. If you don't see the ellipsis, hover over the header.
4. Choose **Create IoT Hub**.

5. A pop-up will show in the bottom right corner to let you sign in to Azure for the first time.
6. Select Azure subscription.
7. Select resource group.
8. Select location.
9. Select pricing tier.
10. Enter a globally unique name for your IoT Hub.
11. Wait a few minutes until the IoT Hub is created.

Next steps

Now you have deployed an IoT hub using the Azure IoT Tools for Visual Studio Code. To explore further, check out the following articles:

- [Use the Azure IoT Tools for Visual Studio Code to send and receive messages between your device and an IoT Hub.](#)
- [Use the Azure IoT Tools for Visual Studio Code for Azure IoT Hub device management](#)
- [See the Azure IoT Hub Toolkit wiki page.](#)

Create an IoT hub using the New-AzIoTHub cmdlet

2/22/2019 • 2 minutes to read

Introduction

You can use Azure PowerShell cmdlets to create and manage Azure IoT hubs. This tutorial shows you how to create an IoT hub with PowerShell.

To complete this how-to, you need an Azure subscription. If you don't have an Azure subscription, create a [free account](#) before you begin.

NOTE

This article has been updated to use the new Azure PowerShell Az module. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For installation instructions, see [Install Azure PowerShell](#).

Open Azure Cloud Shell

Azure Cloud Shell is a free, interactive shell that you can use to run the steps in this article. Common Azure tools are preinstalled and configured in Cloud Shell for you to use with your account. Just select the **Copy** button to copy the code, paste it in Cloud Shell, and then press Enter to run it. There are a few ways to open Cloud Shell:

Select **Try It** in the upper-right corner of a code block.



Open Cloud Shell in your browser.

[Launch Cloud Shell](#)

Select the **Cloud Shell** button on the menu in the upper-right corner of the [Azure portal](#).



Connect to your Azure subscription

If you are using the Cloud Shell, you are already logged in to your subscription. If you are running PowerShell locally instead, enter the following command to sign in to your Azure subscription:

```
# Log into Azure account.  
Login-AzAccount
```

Create a resource group

You need a resource group to deploy an IoT hub. You can use an existing resource group or create a new one.

To create a resource group for your IoT hub, use the [New-AzResourceGroup](#) command. This example creates a resource group called **MyIoTRG1** in the **East US** region:

```
New-AzResourceGroup -Name MyIoTRG1 -Location "East US"
```

Create an IoT hub

To create an IoT hub in the resource group you created in the previous step, use the [New-AzIoTHub](#) command.

This example creates an **S1** hub called **MyTestIoTHub** in the **East US** region:

```
New-AzIoTHub `  
-ResourceGroupName MyIoTRG1 `  
-Name MyTestIoTHub `  
-SkuName S1 -Units 1 `  
-Location "East US"
```

The name of the IoT hub must be globally unique.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

You can list all the IoT hubs in your subscription using the [Get-AzIoTHub](#) command:

```
Get-AzIoTHub
```

This example shows the S1 Standard IoT Hub you created in the previous step.

You can delete the IoT hub using the [Remove-AzIoTHub](#) command:

```
Remove-AzIoTHub `  
-ResourceGroupName MyIoTRG1 `  
-Name MyTestIoTHub
```

Alternatively, you can remove a resource group and all the resources it contains using the [Remove-AzResourceGroup](#) command:

```
Remove-AzResourceGroup -Name MyIoTRG1
```

Next steps

Now you have deployed an IoT hub using a PowerShell cmdlet, if you want to explore further, check out the following articles:

- [PowerShell cmdlets for working with your IoT hub.](#)
- [IoT Hub resource provider REST API.](#)

To learn more about developing for IoT Hub, see the following articles:

- [Introduction to C SDK](#)
- [Azure IoT SDKs](#)

To further explore the capabilities of IoT Hub, see:

- Deploying AI to edge devices with Azure IoT Edge

Create an IoT hub using the Azure CLI

1/29/2019 • 2 minutes to read

This article shows you how to create an IoT hub using Azure CLI.

Prerequisites

To complete this how-to, you need an Azure subscription. If you don't have an Azure subscription, create a [free account](#) before you begin.

Open Azure Cloud Shell

Azure Cloud Shell is a free, interactive shell that you can use to run the steps in this article. Common Azure tools are preinstalled and configured in Cloud Shell for you to use with your account. Just select the **Copy** button to copy the code, paste it in Cloud Shell, and then press Enter to run it. There are a few ways to open Cloud Shell:

Select Try It in the upper-right corner of a code block.	
Open Cloud Shell in your browser.	
Select the Cloud Shell button on the menu in the upper-right corner of the Azure portal .	

Sign in and set your Azure account

If you are running Azure CLI locally instead of using Cloud Shell, you need to sign in to your Azure account.

At the command prompt, run the [login command](#):

```
az login
```

Follow the instructions to authenticate using the code and sign in to your Azure account through a web browser.

Create an IoT Hub

Use the Azure CLI to create a resource group and then add an IoT hub.

- When you create an IoT hub, you must create it in a resource group. Either use an existing resource group, or run the following [command to create a resource group](#):

```
az group create --name {your resource group name} --location westus
```

TIP

The previous example creates the resource group in the West US location. You can view a list of available locations by running this command:

```
az account list-locations -o table
```

- Run the following [command to create an IoT hub](#) in your resource group, using a globally unique name for your IoT hub:

```
az iot hub create --name {your iot hub name} \
--resource-group {your resource group name} --sku S1
```

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

The previous command creates an IoT hub in the S1 pricing tier for which you are billed. For more information, see [Azure IoT Hub pricing](#).

Remove an IoT Hub

You can use Azure CLI to [delete an individual resource](#), such as an IoT hub, or delete a resource group and all its resources, including any IoT hubs.

To [delete an IoT hub](#), run the following command:

```
az iot hub delete --name {your iot hub name} -\
--resource-group {your resource group name}
```

To [delete a resource group](#) and all its resources, run the following command:

```
az group delete --name {your resource group name}
```

Next steps

To learn more about using an IoT hub, see the following articles:

- [IoT Hub developer guide](#)
- [Using the Azure portal to manage IoT Hub](#)

Create an IoT hub using the resource provider REST API (.NET)

2/28/2019 • 7 minutes to read

You can use the [IoT Hub resource provider REST API](#) to create and manage Azure IoT hubs programmatically. This tutorial shows you how to use the IoT Hub resource provider REST API to create an IoT hub from a C# program.

NOTE

Azure has two different deployment models for creating and working with resources: [Azure Resource Manager](#) and [classic](#). This article covers using the Azure Resource Manager deployment model.

NOTE

This article has been updated to use the new Azure PowerShell Az module. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For installation instructions, see [Install Azure PowerShell](#).

To complete this tutorial, you need the following:

- Visual Studio 2015 or Visual Studio 2017.
- An active Azure account.
If you don't have an account, you can create a [free account](#) in just a couple of minutes.
- [Azure PowerShell 1.0](#) or later.

Prepare to authenticate Azure Resource Manager requests

You must authenticate all the operations that you perform on resources using the [Azure Resource Manager](#) with Azure Active Directory (AD). The easiest way to configure this is to use PowerShell or Azure CLI.

Install the [Azure PowerShell cmdlets](#) before you continue.

The following steps show how to set up password authentication for an AD application using PowerShell. You can run these commands in a standard PowerShell session.

1. Sign in to your Azure subscription using the following command:

```
Connect-AzureRmAccount
```

2. If you have multiple Azure subscriptions, signing in to Azure grants you access to all the Azure subscriptions associated with your credentials. Use the following command to list the Azure subscriptions available for you to use:

```
Get-AzureRMSubscription
```

Use the following command to select subscription that you want to use to run the commands to manage your IoT hub. You can use either the subscription name or ID from the output of the previous command:

```
Select-AzureRMSubscription ` 
    -SubscriptionName "{your subscription name}"
```

3. Make a note of your **TenantId** and **SubscriptionId**. You need them later.
4. Create a new Azure Active Directory application using the following command, replacing the place holders:
 - **{Display name}**: a display name for your application such as **MySampleApp**
 - **{Home page URL}**: the URL of the home page of your app such as **http://mysampleapp/home**. This URL does not need to point to a real application.
 - **{Application identifier}**: A unique identifier such as **http://mysampleapp**. This URL does not need to point to a real application.
 - **{Password}**: A password that you use to authenticate with your app.

```
$SecurePassword=ConvertTo-SecureString {password} -asplaintext -force
New-AzureRmADApplication -DisplayName {Display name} -HomePage {Home page URL} -IdentifierUris
{Application identifier} -Password $SecurePassword
```

5. Make a note of the **ApplicationId** of the application you created. You need this later.
6. Create a new service principal using the following command, replacing **{MyApplicationId}** with the **ApplicationId** from the previous step:

```
New-AzureRmADServicePrincipal -ApplicationId {MyApplicationId}
```

7. Set up a role assignment using the following command, replacing **{MyApplicationId}** with your **ApplicationId**.

```
New-AzureRmRoleAssignment -RoleDefinitionName Owner -ServicePrincipalName {MyApplicationId}
```

You have now finished creating the Azure AD application that enables you to authenticate from your custom C# application. You need the following values later in this tutorial:

- TenantId
- SubscriptionId
- ApplicationId
- Password

Prepare your Visual Studio project

1. In Visual Studio, create a Visual C# Windows Classic Desktop project using the **Console App (.NET Framework)** project template. Name the project **CreateIoTHubREST**.
2. In Solution Explorer, right-click on your project and then click **Manage NuGet Packages**.
3. In NuGet Package Manager, check **Include prerelease**, and on the **Browse** page search for **Microsoft.Azure.Management.ResourceManager**. Select the package, click **Install**, in **Review Changes** click **OK**, then click **I Accept** to accept the licenses.
4. In NuGet Package Manager, search for **Microsoft.IdentityModel.Clients.ActiveDirectory**. Click **Install**, in **Review Changes** click **OK**, then click **I Accept** to accept the license.
5. In Program.cs, replace the existing **using** statements with the following code:

```
using System;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Text;
using Microsoft.Azure.Management.ResourceManager;
using Microsoft.Azure.Management.ResourceManager.Models;
using Microsoft.IdentityModel.Clients.ActiveDirectory;
using Newtonsoft.Json;
using Microsoft.Rest;
using System.Linq;
using System.Threading;
```

6. In Program.cs, add the following static variables replacing the placeholder values. You made a note of **ApplicationId**, **SubscriptionId**, **TenantId**, and **Password** earlier in this tutorial. **Resource group name** is the name of the resource group you use when you create the IoT hub. You can use a pre-existing or a new resource group. **IoT Hub name** is the name of the IoT Hub you create, such as **MyIoTHub**. The name of your IoT hub must be globally unique. **Deployment name** is a name for the deployment, such as **Deployment_01**.

```
static string applicationId = "{Your ApplicationId}";
static string subscriptionId = "{Your SubscriptionId}";
static string tenantId = "{Your TenantId}";
static string password = "{Your application Password}";

static string rgName = "{Resource group name}";
static string iotHubName = "{IoT Hub name including your initials}";
```

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

Obtain an Azure Resource Manager token

Azure Active Directory must authenticate all the tasks that you perform on resources using the Azure Resource Manager. The example shown here uses password authentication, for other approaches see [Authenticating Azure Resource Manager requests](#).

1. Add the following code to the **Main** method in Program.cs to retrieve a token from Azure AD using the application id and password.

```
var authContext = new AuthenticationContext(string.Format
    ("https://login.microsoftonline.com/{0}", tenantId));
var credential = new ClientCredential(applicationId, password);
AuthenticationResult token = authContext.AcquireTokenAsync
    ("https://management.core.windows.net/", credential).Result;

if (token == null)
{
    Console.WriteLine("Failed to obtain the token");
    return;
}
```

2. Create a **ResourceManagementClient** object that uses the token by adding the following code to the end of the **Main** method:

```
var creds = new TokenCredentials(token.AccessToken);
var client = new ResourceManagementClient(creds);
client.SubscriptionId = subscriptionId;
```

3. Create, or obtain a reference to, the resource group you are using:

```
var rgResponse = client.ResourceGroups.CreateOrUpdate(rgName,
    new ResourceGroup("East US"));
if (rgResponse.Properties.ProvisioningState != "Succeeded")
{
    Console.WriteLine("Problem creating resource group");
    return;
}
```

Use the resource provider REST API to create an IoT hub

Use the [IoT Hub resource provider REST API](#) to create an IoT hub in your resource group. You can also use the resource provider REST API to make changes to an existing IoT hub.

1. Add the following method to Program.cs:

```
static void CreateIoTHub(string token)
{
}
```

2. Add the following code to the **CreateIoTHub** method. This code creates an **HttpClient** object with the authentication token in the headers:

```
HttpClient client = new HttpClient();
client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", token);
```

3. Add the following code to the **CreateIoTHub** method. This code describes the IoT hub to create and generates a JSON representation. For the current list of locations that support IoT Hub see [Azure Status](#):

```
var description = new
{
    name = iotHubName,
    location = "East US",
    sku = new
    {
        name = "S1",
        tier = "Standard",
        capacity = 1
    }
};

var json = JsonConvert.SerializeObject(description, Formatting.Indented);
```

4. Add the following code to the **CreateIoTHub** method. This code submits the REST request to Azure. The code then checks the response and retrieves the URL you can use to monitor the state of the deployment task:

```

var content = new StringContent(JsonConvert.SerializeObject(description), Encoding.UTF8,
    "application/json");
var requestUri =
    string.Format("https://management.azure.com/subscriptions/{0}/resourcegroups/{1}/providers/Microsoft.de
vices/IotHubs/{2}?api-version=2016-02-03", subscriptionId, rgName, iotHubName);
var result = client.PutAsync(requestUri, content).Result;

if (!result.IsSuccessStatusCode)
{
    Console.WriteLine("Failed {0}", result.Content.ReadAsStringAsync().Result);
    return;
}

var asyncStatusUri = result.Headers.GetValues("Azure-AsyncOperation").First();

```

- Add the following code to the end of the **CreateIoTHub** method. This code uses the **asyncStatusUri** address retrieved in the previous step to wait for the deployment to complete:

```

string body;
do
{
    Thread.Sleep(10000);
    HttpResponseMessage deploymentstatus = client.GetAsync(asyncStatusUri).Result;
    body = deploymentstatus.Content.ReadAsStringAsync().Result;
} while (body == "{\"status\":\"Running\"}");

```

- Add the following code to the end of the **CreateIoTHub** method. This code retrieves the keys of the IoT hub you created and prints them to the console:

```

var listKeysUri =
    string.Format("https://management.azure.com/subscriptions/{0}/resourceGroups/{1}/providers/Microsoft.De
vices/IotHubs/{2}/IoTHubKeys/listkeys?api-version=2016-02-03", subscriptionId, rgName, iotHubName);
var keysresults = client.PostAsync(listKeysUri, null).Result;

Console.WriteLine("Keys: {0}", keysresults.Content.ReadAsStringAsync().Result);

```

Complete and run the application

You can now complete the application by calling the **CreateIoTHub** method before you build and run it.

- Add the following code to the end of the **Main** method:

```

CreateIoTHub(token.AccessToken);
Console.ReadLine();

```

- Click **Build** and then **Build Solution**. Correct any errors.
- Click **Debug** and then **Start Debugging** to run the application. It may take several minutes for the deployment to run.
- To verify that your application added the new IoT hub, visit the [Azure portal](#) and view your list of resources. Alternatively, use the **Get-AzResource** PowerShell cmdlet.

NOTE

This example application adds an S1 Standard IoT Hub for which you are billed. When you are finished, you can delete the IoT hub through the [Azure portal](#) or by using the **Remove-AzResource** PowerShell cmdlet when you are finished.

Next steps

Now you have deployed an IoT hub using the resource provider REST API, you may want to explore further:

- Read about the capabilities of the [IoT Hub resource provider REST API](#).
- Read [Azure Resource Manager overview](#) to learn more about the capabilities of Azure Resource Manager.

To learn more about developing for IoT Hub, see the following articles:

- [Introduction to C SDK](#)
- [Azure IoT SDKs](#)

To further explore the capabilities of IoT Hub, see:

- [Deploying AI to edge devices with Azure IoT Edge](#)

Create an IoT hub using Azure Resource Manager template (PowerShell)

2/28/2019 • 4 minutes to read

You can use Azure Resource Manager to create and manage Azure IoT hubs programmatically. This tutorial shows you how to use an Azure Resource Manager template to create an IoT hub with PowerShell.

NOTE

Azure has two different deployment models for creating and working with resources: [Azure Resource Manager and classic](#). This article covers using the Azure Resource Manager deployment model.

NOTE

This article has been updated to use the new Azure PowerShell Az module. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For installation instructions, see [Install Azure PowerShell](#).

To complete this tutorial, you need the following:

- An active Azure account.
If you don't have an account, you can create a [free account](#) in just a couple of minutes.
- [Azure PowerShell 1.0 or later](#).

TIP

The article [Using Azure PowerShell with Azure Resource Manager](#) provides more information about how to use PowerShell and Azure Resource Manager templates to create Azure resources.

Connect to your Azure subscription

In a PowerShell command prompt, enter the following command to sign in to your Azure subscription:

```
Connect-AzAccount
```

If you have multiple Azure subscriptions, signing in to Azure grants you access to all the Azure subscriptions associated with your credentials. Use the following command to list the Azure subscriptions available for you to use:

```
Get-AzSubscription
```

Use the following command to select subscription that you want to use to run the commands to create your IoT hub. You can use either the subscription name or ID from the output of the previous command:

```
Select-AzSubscription `
```

```
-SubscriptionName "{your subscription name}"
```

You can use the following commands to discover where you can deploy an IoT hub and the currently supported API versions:

```
((Get-AzResourceProvider -ProviderNamespace Microsoft.Devices).ResourceTypes | Where-Object ResourceTypeName -eq IoTHubs).Locations  
((Get-AzResourceProvider -ProviderNamespace Microsoft.Devices).ResourceTypes | Where-Object ResourceTypeName -eq IoTHubs).ApiVersions
```

Create a resource group to contain your IoT hub using the following command in one of the supported locations for IoT Hub. This example creates a resource group called **MyIoTRG1**:

```
New-AzResourceGroup -Name MyIoTRG1 -Location "East US"
```

Submit a template to create an IoT hub

Use a JSON template to create an IoT hub in your resource group. You can also use an Azure Resource Manager template to make changes to an existing IoT hub.

1. Use a text editor to create an Azure Resource Manager template called **template.json** with the following resource definition to create a new standard IoT hub. This example adds the IoT Hub in the **East US** region, creates two consumer groups (**cg1** and **cg2**) on the Event Hub-compatible endpoint, and uses the **2016-02-03** API version. This template also expects you to pass in the IoT hub name as a parameter called **hubName**. For the current list of locations that support IoT Hub see [Azure Status](#).

```
{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "hubName": {
            "type": "string"
        }
    },
    "resources": [
    {
        "apiVersion": "2016-02-03",
        "type": "Microsoft.Devices/IotHubs",
        "name": "[parameters('hubName')]",
        "location": "East US",
        "sku": {
            "name": "S1",
            "tier": "Standard",
            "capacity": 1
        },
        "properties": {
            "location": "East US"
        }
    },
    {
        "apiVersion": "2016-02-03",
        "type": "Microsoft.Devices/IotHubs/eventhubEndpoints/ConsumerGroups",
        "name": "[concat(parameters('hubName'), '/events/cg1')]",
        "dependsOn": [
            "[concat('Microsoft.Devices/Iothubs/', parameters('hubName'))]"
        ]
    },
    {
        "apiVersion": "2016-02-03",
        "type": "Microsoft.Devices/IotHubs/eventhubEndpoints/ConsumerGroups",
        "name": "[concat(parameters('hubName'), '/events/cg2')]",
        "dependsOn": [
            "[concat('Microsoft.Devices/Iothubs/', parameters('hubName'))]"
        ]
    }
    ],
    "outputs": {
        "hubKeys": {
            "value": "[listKeys(resourceId('Microsoft.Devices/IotHubs', parameters('hubName')), '2016-02-03')]",
            "type": "object"
        }
    }
}
```

2. Save the Azure Resource Manager template file on your local machine. This example assumes you save it in a folder called **c:\templates**.
3. Run the following command to deploy your new IoT hub, passing the name of your IoT hub as a parameter. In this example, the name of the IoT hub is `abcmyiothub`. The name of your IoT hub must be globally unique:

```
New-AzResourceGroupDeployment -ResourceGroupName MyIoTRG1 -TemplateFile C:\templates\template.json -hubName abcmyiothub
```

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. The output displays the keys for the IoT hub you created.
5. To verify your application added the new IoT hub, visit the [Azure portal](#) and view your list of resources.
Alternatively, use the **Get-AzResource** PowerShell cmdlet.

NOTE

This example application adds an S1 Standard IoT Hub for which you are billed. You can delete the IoT hub through the [Azure portal](#) or by using the **Remove-AzResource** PowerShell cmdlet when you are finished.

Next steps

Now you have deployed an IoT hub using an Azure Resource Manager template with PowerShell, you may want to explore further:

- Read about the capabilities of the [IoT Hub resource provider REST API](#).
- Read [Azure Resource Manager overview](#) to learn more about the capabilities of Azure Resource Manager.
- For the JSON syntax and properties to use in templates, see [Microsoft.Devices resource types](#).

To learn more about developing for IoT Hub, see the following articles:

- [Introduction to C SDK](#)
- [Azure IoT SDKs](#)

To further explore the capabilities of IoT Hub, see:

- [Deploying AI to edge devices with Azure IoT Edge](#)

Create an IoT hub using Azure Resource Manager template (.NET)

2/28/2019 • 8 minutes to read

You can use Azure Resource Manager to create and manage Azure IoT hubs programmatically. This tutorial shows you how to use an Azure Resource Manager template to create an IoT hub from a C# program.

NOTE

Azure has two different deployment models for creating and working with resources: [Azure Resource Manager and classic](#). This article covers using the Azure Resource Manager deployment model.

NOTE

This article has been updated to use the new Azure PowerShell Az module. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For installation instructions, see [Install Azure PowerShell](#).

To complete this tutorial, you need the following:

- Visual Studio 2015 or Visual Studio 2017.
- An active Azure account.
If you don't have an account, you can create a [free account](#) in just a couple of minutes.
- An [Azure Storage account](#) where you can store your Azure Resource Manager template files.
- [Azure PowerShell 1.0](#) or later.

Prepare to authenticate Azure Resource Manager requests

You must authenticate all the operations that you perform on resources using the [Azure Resource Manager](#) with Azure Active Directory (AD). The easiest way to configure this is to use PowerShell or Azure CLI.

Install the [Azure PowerShell cmdlets](#) before you continue.

The following steps show how to set up password authentication for an AD application using PowerShell. You can run these commands in a standard PowerShell session.

1. Sign in to your Azure subscription using the following command:

```
Connect-AzureRmAccount
```

2. If you have multiple Azure subscriptions, signing in to Azure grants you access to all the Azure subscriptions associated with your credentials. Use the following command to list the Azure subscriptions available for you to use:

```
Get-AzureRMSubscription
```

Use the following command to select subscription that you want to use to run the commands to manage your IoT hub. You can use either the subscription name or ID from the output of the previous command:

```
Select-AzureRMSubscription  
-SubscriptionName "{your subscription name}"
```

3. Make a note of your **TenantId** and **SubscriptionId**. You need them later.
4. Create a new Azure Active Directory application using the following command, replacing the place holders:
 - **{Display name}**: a display name for your application such as **MySampleApp**
 - **{Home page URL}**: the URL of the home page of your app such as **http://mysampleapp/home**. This URL does not need to point to a real application.
 - **{Application identifier}**: A unique identifier such as **http://mysampleapp**. This URL does not need to point to a real application.
 - **{Password}**: A password that you use to authenticate with your app.

```
$SecurePassword=ConvertTo-SecureString {password} -asplaintext -force  
New-AzureRmADApplication -DisplayName {Display name} -HomePage {Home page URL} -IdentifierUris  
{Application identifier} -Password $SecurePassword
```

5. Make a note of the **ApplicationId** of the application you created. You need this later.
6. Create a new service principal using the following command, replacing **{MyApplicationId}** with the **ApplicationId** from the previous step:

```
New-AzureRmADServicePrincipal -ApplicationId {MyApplicationId}
```

7. Set up a role assignment using the following command, replacing **{MyApplicationId}** with your **ApplicationId**.

```
New-AzureRmRoleAssignment -RoleDefinitionName Owner -ServicePrincipalName {MyApplicationId}
```

You have now finished creating the Azure AD application that enables you to authenticate from your custom C# application. You need the following values later in this tutorial:

- TenantId
- SubscriptionId
- ApplicationId
- Password

Prepare your Visual Studio project

1. In Visual Studio, create a Visual C# Windows Classic Desktop project using the **Console App (.NET Framework)** project template. Name the project **CreateIoTHub**.
2. In Solution Explorer, right-click on your project and then click **Manage NuGet Packages**.
3. In NuGet Package Manager, check **Include prerelease**, and on the **Browse** page search for **Microsoft.Azure.Management.ResourceManager**. Select the package, click **Install**, in **Review Changes** click **OK**, then click **I Accept** to accept the licenses.
4. In NuGet Package Manager, search for **Microsoft.IdentityModel.Clients.ActiveDirectory**. Click **Install**, in **Review Changes** click **OK**, then click **I Accept** to accept the license.
5. In Program.cs, replace the existing **using** statements with the following code:

```
using System;
using Microsoft.Azure.Management.ResourceManager;
using Microsoft.Azure.Management.ResourceManager.Models;
using Microsoft.IdentityModel.Clients.ActiveDirectory;
using Microsoft.Rest;
```

6. In Program.cs, add the following static variables replacing the placeholder values. You made a note of **ApplicationId**, **SubscriptionId**, **TenantId**, and **Password** earlier in this tutorial. **Your Azure Storage account name** is the name of the Azure Storage account where you store your Azure Resource Manager template files. **Resource group name** is the name of the resource group you use when you create the IoT hub. The name can be a pre-existing or new resource group. **Deployment name** is a name for the deployment, such as **Deployment_01**.

```
static string applicationId = "{Your ApplicationId}";
static string subscriptionId = "{Your SubscriptionId}";
static string tenantId = "{Your TenantId}";
static string password = "{Your application Password}";
static string storageAddress = "https://'{Your storage account name}.blob.core.windows.net";
static string rgName = "{Resource group name}";
static string deploymentName = "{Deployment name}";
```

Obtain an Azure Resource Manager token

Azure Active Directory must authenticate all the tasks that you perform on resources using the Azure Resource Manager. The example shown here uses password authentication, for other approaches see [Authenticating Azure Resource Manager requests](#).

1. Add the following code to the **Main** method in Program.cs to retrieve a token from Azure AD using the application id and password.

```
var authContext = new AuthenticationContext(string.Format
    ("https://login.microsoftonline.com/{0}", tenantId));
var credential = new ClientCredential(applicationId, password);
AuthenticationResult token = authContext.AcquireTokenAsync
    ("https://management.core.windows.net/", credential).Result;

if (token == null)
{
    Console.WriteLine("Failed to obtain the token");
    return;
}
```

2. Create a **ResourceManagementClient** object that uses the token by adding the following code to the end of the **Main** method:

```
var creds = new TokenCredentials(token.AccessToken);
var client = new ResourceManagementClient(creds);
client.SubscriptionId = subscriptionId;
```

3. Create, or obtain a reference to, the resource group you are using:

```

var rgResponse = client.ResourceGroups.CreateOrUpdate(rgName,
    new ResourceGroup("East US"));
if (rgResponse.Properties.ProvisioningState != "Succeeded")
{
    Console.WriteLine("Problem creating resource group");
    return;
}

```

Submit a template to create an IoT hub

Use a JSON template and parameter file to create an IoT hub in your resource group. You can also use an Azure Resource Manager template to make changes to an existing IoT hub.

1. In Solution Explorer, right-click on your project, click **Add**, and then click **New Item**. Add a JSON file called **template.json** to your project.
2. To add a standard IoT hub to the **East US** region, replace the contents of **template.json** with the following resource definition. For the current list of regions that support IoT Hub see [Azure Status](#):

```

{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "hubName": {
            "type": "string"
        }
    },
    "resources": [
        {
            "apiVersion": "2016-02-03",
            "type": "Microsoft.Devices/IotHubs",
            "name": "[parameters('hubName')]",
            "location": "East US",
            "sku": {
                "name": "S1",
                "tier": "Standard",
                "capacity": 1
            },
            "properties": {
                "location": "East US"
            }
        }
    ],
    "outputs": {
        "hubKeys": {
            "value": "[listKeys(resourceId('Microsoft.Devices/IotHubs', parameters('hubName')), '2016-02-03')]",
            "type": "object"
        }
    }
}

```

3. In Solution Explorer, right-click on your project, click **Add**, and then click **New Item**. Add a JSON file called **parameters.json** to your project.
4. Replace the contents of **parameters.json** with the following parameter information that sets a name for the new IoT hub such as **{your initials}mynewiothub**. The IoT hub name must be globally unique so it should include your name or initials:

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentParameters.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "hubName": { "value": "mynewiothub" }
  }
}
```

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

5. In **Server Explorer**, connect to your Azure subscription, and in your Azure Storage account create a container called **templates**. In the **Properties** panel, set the **Public Read Access** permissions for the **templates** container to **Blob**.
6. In **Server Explorer**, right-click on the **templates** container and then click **View Blob Container**. Click the **Upload Blob** button, select the two files, **parameters.json** and **templates.json**, and then click **Open** to upload the JSON files to the **templates** container. The URLs of the blobs containing the JSON data are:

```
https://{{Your storage account name}}.blob.core.windows.net/templates/parameters.json  
https://{{Your storage account name}}.blob.core.windows.net/templates/template.json
```

7. Add the following method to Program.cs:

```
static void CreateIoTHub(ResourceManagementClient client)
{
}
```

8. Add the following code to the **CreateIoTHub** method to submit the template and parameter files to the Azure Resource Manager:

```
var createResponse = client.Deployments.CreateOrUpdate(
  rgName,
  deploymentName,
  new Deployment()
{
  Properties = new DeploymentProperties
  {
    Mode = DeploymentMode.Incremental,
    TemplateLink = new TemplateLink
    {
      Uri = storageAddress + "/templates/template.json"
    },
    ParametersLink = new ParametersLink
    {
      Uri = storageAddress + "/templates/parameters.json"
    }
  });
}
```

9. Add the following code to the **CreateIoTHub** method that displays the status and the keys for the new IoT hub:

```
string state = createResponse.Properties.ProvisioningState;
Console.WriteLine("Deployment state: {0}", state);

if (state != "Succeeded")
{
    Console.WriteLine("Failed to create iothub");
}
Console.WriteLine(createResponse.Properties.Outputs);
```

Complete and run the application

You can now complete the application by calling the **CreateIoTHub** method before you build and run it.

1. Add the following code to the end of the **Main** method:

```
CreateIoTHub(client);
Console.ReadLine();
```

2. Click **Build** and then **Build Solution**. Correct any errors.
3. Click **Debug** and then **Start Debugging** to run the application. It may take several minutes for the deployment to run.
4. To verify your application added the new IoT hub, visit the [Azure portal](#) and view your list of resources.
Alternatively, use the **Get-AzResource** PowerShell cmdlet.

NOTE

This example application adds an S1 Standard IoT Hub for which you are billed. You can delete the IoT hub through the [Azure portal](#) or by using the **Remove-AzResource** PowerShell cmdlet when you are finished.

Next steps

Now you have deployed an IoT hub using an Azure Resource Manager template with a C# program, you may want to explore further:

- Read about the capabilities of the [IoT Hub resource provider REST API](#).
- Read [Azure Resource Manager overview](#) to learn more about the capabilities of Azure Resource Manager.
- For the JSON syntax and properties to use in templates, see [Microsoft.Devices resource types](#).

To learn more about developing for IoT Hub, see the following articles:

- [Introduction to C SDK](#)
- [Azure IoT SDKs](#)

To further explore the capabilities of IoT Hub, see:

- [Deploying AI to edge devices with Azure IoT Edge](#)

Configure IoT Hub file uploads using the Azure portal

2/28/2019 • 2 minutes to read

File upload

To use the [file upload functionality in IoT Hub](#), you must first associate an Azure Storage account with your hub. Select **File upload** to display a list of file upload properties for the IoT hub that is being modified.

The screenshot shows the 'File upload' configuration blade for an IoT hub named 'getStartedWithAnIoTHub'. The left sidebar lists various configuration sections: Pricing and scale, Operations monitoring, IP Filter, Properties, Locks, and Automation script. Below these are EXPLORERS (Device Explorer, Query Explorer), and MESSAGING (File upload, Endpoints, Routes). The 'MESSAGING' section is highlighted with a red box around the 'File upload' item. The main content area shows a summary message: 'Here you specify the storage container, file expiration, and retries for notifications when a file is uploaded.' It includes fields for 'Storage container' (set to 'uploadcontainer'), 'Receive notifications for uploaded files' (set to 'On'), 'SAS TTL' (set to '1 hr'), 'File notification settings' (Default TTL set to '1 hr'), and 'Maximum delivery count' (set to '10'). Top navigation buttons include Save and Discard.

- **Storage container:** Use the Azure portal to select a blob container in an Azure Storage account in your current Azure subscription to associate with your IoT Hub. If necessary, you can create an Azure Storage account on the **Storage accounts** blade and blob container on the **Containers** blade. IoT Hub automatically generates SAS URIs with write permissions to this blob container for devices to use when they upload files.

- Receive notifications for uploaded files:** Enable or disable file upload notifications via the toggle.
- SAS TTL:** This setting is the time-to-live of the SAS URIs returned to the device by IoT Hub. Set to one hour by default but can be customized to other values using the slider.
- File notification settings default TTL:** The time-to-live of a file upload notification before it is expired. Set to one day by default but can be customized to other values using the slider.
- File notification maximum delivery count:** The number of times the IoT Hub attempts to deliver a file upload notification. Set to 10 by default but can be customized to other values using the slider.

Next steps

For more information about the file upload capabilities of IoT Hub, see [Upload files from a device](#) in the IoT Hub developer guide.

Follow these links to learn more about managing Azure IoT Hub:

- [Bulk manage IoT devices](#)
- [IoT Hub metrics](#)
- [Operations monitoring](#)

To further explore the capabilities of IoT Hub, see:

- [IoT Hub developer guide](#)
- [Deploying AI to edge devices with Azure IoT Edge](#)
- [Secure your IoT solution from the ground up](#)

Configure IoT Hub file uploads using PowerShell

2/28/2019 • 3 minutes to read

To use the [file upload functionality in IoT Hub](#), you must first associate an Azure storage account with your IoT hub. You can use an existing storage account or create a new one.

NOTE

This article has been updated to use the new Azure PowerShell Az module. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For installation instructions, see [Install Azure PowerShell](#).

To complete this tutorial, you need the following:

- An active Azure account. If you don't have an account, you can create a [free account](#) in just a couple of minutes.
- [Azure PowerShell cmdlets](#).
- An Azure IoT hub. If you don't have an IoT hub, you can use the [New-AzIoTHub cmdlet](#) to create one or use the portal to [Create an IoT hub](#).
- An Azure storage account. If you don't have an Azure storage account, you can use the [Azure Storage PowerShell cmdlets](#) to create one or use the portal to [Create a storage account](#)

Sign in and set your Azure account

Sign in to your Azure account and select your subscription.

1. At the PowerShell prompt, run the **Connect-AzAccount** cmdlet:

```
Connect-AzAccount
```

2. If you have multiple Azure subscriptions, signing in to Azure grants you access to all the Azure subscriptions associated with your credentials. Use the following command to list the Azure subscriptions available for you to use:

```
Get-AzSubscription
```

Use the following command to select the subscription that you want to use to run the commands to manage your IoT hub. You can use either the subscription name or ID from the output of the previous command:

```
Select-AzSubscription ` 
-SubscriptionName "{your subscription name}"
```

Retrieve your storage account details

The following steps assume that you created your storage account using the **Resource Manager** deployment model, and not the **Classic** deployment model.

To configure file uploads from your devices, you need the connection string for an Azure storage account. The storage account must be in the same subscription as your IoT hub. You also need the name of a blob container in the storage account. Use the following command to retrieve your storage account keys:

```
Get-AzStorageAccountKey ` 
    -Name {your storage account name} ` 
    -ResourceGroupName {your storage account resource group}
```

Make a note of the **key1** storage account key value. You need it in the following steps.

You can either use an existing blob container for your file uploads or create new one:

- To list the existing blob containers in your storage account, use the following commands:

```
$ctx = New-AzStorageContext ` 
    -StorageAccountName {your storage account name} ` 
    -StorageAccountKey {your storage account key}
Get-AzStorageContainer -Context $ctx
```

- To create a blob container in your storage account, use the following commands:

```
$ctx = New-AzStorageContext ` 
    -StorageAccountName {your storage account name} ` 
    -StorageAccountKey {your storage account key}
New-AzStorageContainer ` 
    -Name {your new container name} ` 
    -Permission Off ` 
    -Context $ctx
```

Configure your IoT hub

You can now configure your IoT hub to [upload files to the IoT hub](#) using your storage account details.

The configuration requires the following values:

- **Storage container:** A blob container in an Azure storage account in your current Azure subscription to associate with your IoT hub. You retrieved the necessary storage account information in the preceding section. IoT Hub automatically generates SAS URIs with write permissions to this blob container for devices to use when they upload files.
- **Receive notifications for uploaded files:** Enable or disable file upload notifications.
- **SAS TTL:** This setting is the time-to-live of the SAS URIs returned to the device by IoT Hub. Set to one hour by default.
- **File notification settings default TTL:** The time-to-live of a file upload notification before it is expired. Set to one day by default.
- **File notification maximum delivery count:** The number of times the IoT Hub attempts to deliver a file upload notification. Set to 10 by default.

Use the following PowerShell cmdlet to configure the file upload settings on your IoT hub:

```
Set-AzIotHub ` 
    -ResourceGroupName "{your iot hub resource group}" ` 
    -Name "{your iot hub name}" ` 
    -FileUploadNotificationTtl "01:00:00" ` 
    -FileUploadSasUriTtl "01:00:00" ` 
    -EnableFileUploadNotifications $true ` 
    -FileUploadStorageConnectionString "DefaultEndpointsProtocol=https;AccountName={your storage account name};AccountKey={your storage account key};EndpointSuffix=core.windows.net" ` 
    -FileUploadContainerName "{your blob container name}" ` 
    -FileUploadNotificationMaxDeliveryCount 10
```

Next steps

For more information about the file upload capabilities of IoT Hub, see [Upload files from a device](#).

Follow these links to learn more about managing Azure IoT Hub:

- [Bulk manage IoT devices](#)
- [IoT Hub metrics](#)
- [Operations monitoring](#)

To further explore the capabilities of IoT Hub, see:

- [IoT Hub developer guide](#)
- [Deploying AI to edge devices with Azure IoT Edge](#)
- [Secure your IoT solution from the ground up](#)

Configure IoT Hub file uploads using Azure CLI

2/28/2019 • 4 minutes to read

To [upload files from a device](#), you must first associate an Azure Storage account with your IoT hub. You can use an existing storage account or create a new one.

To complete this tutorial, you need the following:

- An active Azure account. If you don't have an account, you can create a [free account](#) in just a couple of minutes.
- [Azure CLI](#).
- An Azure IoT hub. If you don't have an IoT hub, you can use the `az iot hub create` command to create one or [Create an IoT hub using the portal](#).
- An Azure Storage account. If you don't have an Azure Storage account, you can use the [Azure CLI - Manage storage accounts](#) to create one or use the portal to [Create a storage account](#).

Sign in and set your Azure account

Sign in to your Azure account and select your subscription.

1. At the command prompt, run the [login command](#):

```
az login
```

Follow the instructions to authenticate using the code and sign in to your Azure account through a web browser.

2. If you have multiple Azure subscriptions, signing in to Azure grants you access to all the Azure accounts associated with your credentials. Use the following [command to list the Azure accounts](#) available for you to use:

```
az account list
```

Use the following command to select the subscription that you want to use to run the commands to create your IoT hub. You can use either the subscription name or ID from the output of the previous command:

```
az account set --subscription {your subscription name or id}
```

Retrieve your storage account details

The following steps assume that you created your storage account using the **Resource Manager** deployment model, and not the **Classic** deployment model.

To configure file uploads from your devices, you need the connection string for an Azure storage account. The storage account must be in the same subscription as your IoT hub. You also need the name of a blob container in the storage account. Use the following command to retrieve your storage account keys:

```
az storage account show-connection-string --name {your storage account name} \  
--resource-group {your storage account resource group}
```

Make a note of the **ConnectionString** value. You need it in the following steps.

You can either use an existing blob container for your file uploads or create a new one:

- To list the existing blob containers in your storage account, use the following command:

```
az storage container list --connection-string "{your storage account connection string}"
```

- To create a blob container in your storage account, use the following command:

```
az storage container create --name {container name} \  
--connection-string "{your storage account connection string}"
```

File upload

You can now configure your IoT hub to enable the ability to [upload files to the IoT hub](#) using your storage account details.

The configuration requires the following values:

- **Storage container:** A blob container in an Azure storage account in your current Azure subscription to associate with your IoT hub. You retrieved the necessary storage account information in the preceding section. IoT Hub automatically generates SAS URIs with write permissions to this blob container for devices to use when they upload files.
- **Receive notifications for uploaded files:** Enable or disable file upload notifications.
- **SAS TTL:** This setting is the time-to-live of the SAS URIs returned to the device by IoT Hub. Set to one hour by default.
- **File notification settings default TTL:** The time-to-live of a file upload notification before it is expired. Set to one day by default.
- **File notification maximum delivery count:** The number of times the IoT Hub attempts to deliver a file upload notification. Set to 10 by default.

Use the following Azure CLI commands to configure the file upload settings on your IoT hub:

In a bash shell, use:

```
az iot hub update --name {your iot hub name} \
--set properties.storageEndpoints.'$default'.connectionString="{your storage account connection string}"

az iot hub update --name {your iot hub name} \
--set properties.storageEndpoints.'$default'.containerName="{your storage container name}"

az iot hub update --name {your iot hub name} \
--set properties.storageEndpoints.'$default'.sasTtlAsIso8601=PT1H0M0S

az iot hub update --name {your iot hub name} \
--set properties.enableFileUploadNotifications=true

az iot hub update --name {your iot hub name} \
--set properties.messagingEndpoints.fileNotifications.maxDeliveryCount=10

az iot hub update --name {your iot hub name} \
--set properties.messagingEndpoints.fileNotifications.ttlAsIso8601=PT1H0M0S
```

You can review the file upload configuration on your IoT hub using the following command:

```
az iot hub show --name {your iot hub name}
```

Next steps

For more information about the file upload capabilities of IoT Hub, see [Upload files from a device](#).

Follow these links to learn more about managing Azure IoT Hub:

- [Bulk manage IoT devices](#)
- [IoT Hub metrics](#)
- [Operations monitoring](#)

To further explore the capabilities of IoT Hub, see:

- [IoT Hub developer guide](#)
- [Deploying AI to edge devices with Azure IoT Edge](#)
- [Secure your IoT solution from the ground up](#)

Understand IoT Hub metrics

10/31/2018 • 8 minutes to read

IoT Hub metrics give you better data about the state of the Azure IoT resources in your Azure subscription. IoT Hub metrics enable you to assess the overall health of the IoT Hub service and the devices connected to it. User-facing statistics are important because they help you see what is going on with your IoT hub and help root-cause issues without needing to contact Azure support.

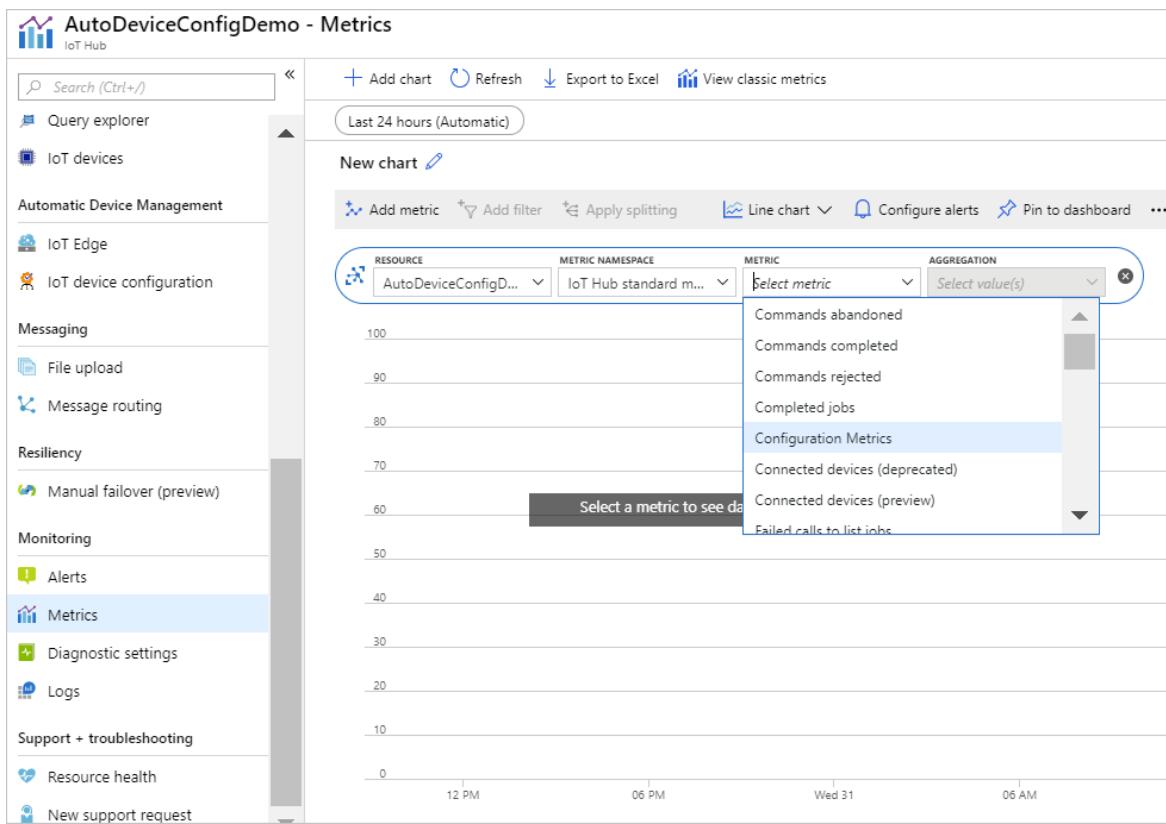
Metrics are enabled by default. You can view IoT Hub metrics from the Azure portal.

How to view IoT Hub metrics

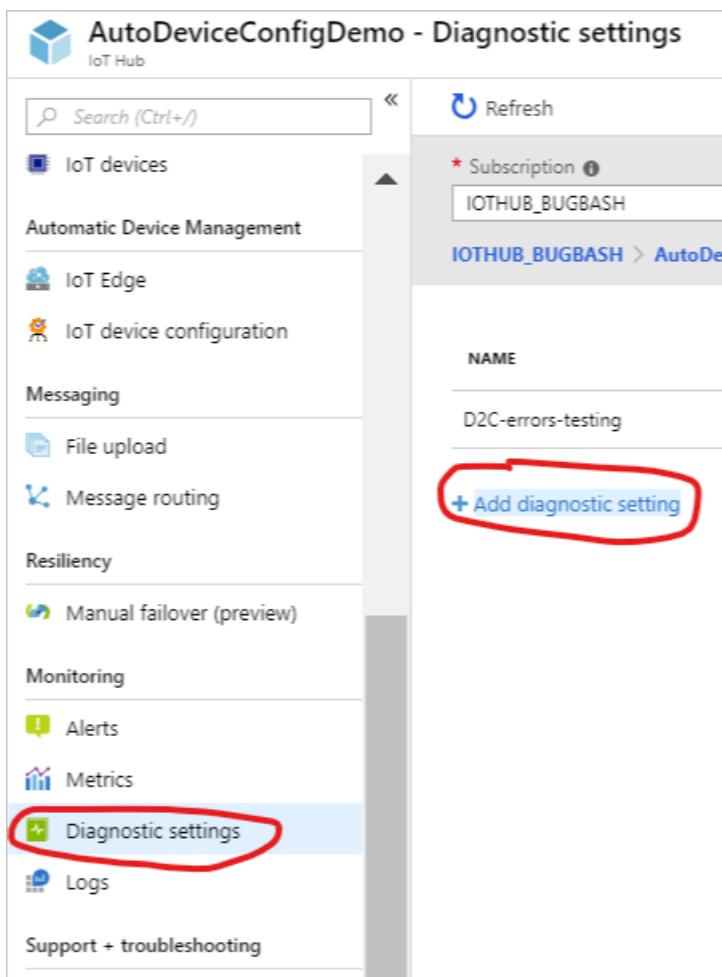
1. Create an IoT hub. You can find instructions on how to create an IoT hub in the [Send telemetry from a device to IoT Hub](#) guide.
2. Open the blade of your IoT hub. From there, click **Metrics**.

The screenshot shows the Azure IoT Hub blade for the resource group 'AutoDeviceConfigDemo'. On the left, a navigation menu lists 'Automatic Device Management' (IoT Edge, IoT device configuration), 'Messaging' (File upload, Message routing), 'Resiliency' (Manual failover (preview)), 'Monitoring' (Alerts, Metrics, Diagnostic settings, Logs), and 'Support + troubleshooting' (Resource health, New support request). The 'Metrics' item is highlighted with a red circle. The main pane displays basic resource information: Resource group (AutoDeviceConfigDemo), Hostname (AutoDeviceConfigDemo.azure-devices.net), Status (Active), Pricing and scale tier (S1 - Standard), Location (West Central US), Number of IoT Hub units (1), Subscription (change <subscription name>), Subscription ID (<subscription ID>), and Tags (change Click here to add tags). Below the main pane, two promotional cards are visible: one for IoT Hub Device Provisioning Service and another for learning more about IoT Hub.

3. From the metrics blade, you can view the metrics for your IoT hub and create custom views of your metrics.



4. You can choose to send your metrics data to an Event Hubs endpoint or an Azure Storage account by clicking **Diagnostics settings**, then **Add diagnostic setting**



IoT Hub metrics and how to use them

IoT Hub provides several metrics to give you an overview of the health of your hub and the total number of

connected devices. You can combine information from multiple metrics to paint a bigger picture of the state of the IoT hub. The following table describes the metrics each IoT hub tracks, and how each metric relates to the overall status of the IoT hub.

METRIC	METRIC DISPLAY NAME	UNIT	AGGREGATION TYPE	DESCRIPTION	DIMENSIONS
d2c .telemetry .ingress. allProtocol	Telemetry message send attempts	Count	Total	Number of device-to-cloud telemetry messages attempted to be sent to your IoT hub	No Dimensions
d2c .telemetry .ingress .success	Telemetry messages sent	Count	Total	Number of device-to-cloud telemetry messages sent successfully to your IoT hub	No Dimensions
c2d .commands .egress .complete .success	Commands completed	Count	Total	Number of cloud-to-device commands completed successfully by the device	No Dimensions
c2d .commands .egress .abandon .success	Commands abandoned	Count	Total	Number of cloud-to-device commands abandoned by the device	No Dimensions
c2d .commands .egress .reject .success	Commands rejected	Count	Total	Number of cloud-to-device commands rejected by the device	No Dimensions
devices .totalDevices	Total devices (deprecated)	Count	Total	Number of devices registered to your IoT hub	No Dimensions
devices .connectedDevices .allProtocol	Connected devices (deprecated)	Count	Total	Number of devices connected to your IoT hub	No Dimensions

METRIC	METRIC DISPLAY NAME	UNIT	AGGREGATION TYPE	DESCRIPTION	DIMENSIONS
d2c .telemetry .egress .success	Routing: telemetry messages delivered	Count	Total	The number of times messages were successfully delivered to all endpoints using IoT Hub routing. If a message is routed to multiple endpoints, this value increases by one for each successful delivery. If a message is delivered to the same endpoint multiple times, this value increases by one for each successful delivery.	No Dimensions
d2c .telemetry .egress .dropped	Routing: telemetry messages dropped	Count	Total	The number of times messages were dropped by IoT Hub routing due to dead endpoints. This value does not count messages delivered to fallback route as dropped messages are not delivered there.	No Dimensions
d2c .telemetry .egress .orphaned	Routing: telemetry messages orphaned	Count	Total	The number of times messages were orphaned by IoT Hub routing because they didn't match any routing rules (including the fallback rule).	No Dimensions

Metric	Metric Display Name	Unit	Aggregation Type	Description	Dimensions
d2c .telemetry .egress .invalid	Routing: telemetry messages incompatible	Count	Total	The number of times IoT Hub routing failed to deliver messages due to an incompatibility with the endpoint. This value does not include retries.	No Dimensions
d2c .telemetry .egress .fallback	Routing: messages delivered to fallback	Count	Total	The number of times IoT Hub routing delivered messages to the endpoint associated with the fallback route.	No Dimensions
d2c .endpoints .egress .eventHubs	Routing: messages delivered to Event Hub	Count	Total	The number of times IoT Hub routing successfully delivered messages to Event Hub endpoints.	No Dimensions
d2c .endpoints .latency .eventHubs	Routing: message latency for Event Hub	Milliseconds	Average	The average latency (milliseconds) between message ingress to IoT Hub and message ingress into an Event Hub endpoint.	No Dimensions
d2c .endpoints .egress .serviceBusQueues	Routing: messages delivered to Service Bus Queue	Count	Total	The number of times IoT Hub routing successfully delivered messages to Service Bus queue endpoints.	No Dimensions

METRIC	METRIC DISPLAY NAME	UNIT	AGGREGATION TYPE	DESCRIPTION	DIMENSIONS
d2c .endpoints .latency .serviceBusQueues	Routing: message latency for Service Bus Queue	Milliseconds	Average	The average latency (milliseconds) between message ingress to IoT Hub and telemetry message ingress into a Service Bus queue endpoint.	No Dimensions
d2c .endpoints .egress .serviceBusTopics	Routing: messages delivered to Service Bus Topic	Count	Total	The number of times IoT Hub routing successfully delivered messages to Service Bus topic endpoints.	No Dimensions
d2c .endpoints .latency .serviceBusTopics	Routing: message latency for Service Bus Topic	Milliseconds	Average	The average latency (milliseconds) between message ingress to IoT Hub and telemetry message ingress into a Service Bus topic endpoint.	No Dimensions
d2c .endpoints .egress .builtIn .events	Routing: messages delivered to messages/events	Count	Total	The number of times IoT Hub routing successfully delivered messages to the built-in endpoint (messages/events).	No Dimensions
d2c .endpoints .latency .builtIn.events	Routing: message latency for messages/events	Milliseconds	Average	The average latency (milliseconds) between message ingress to IoT Hub and telemetry message ingress into the built-in endpoint (messages/events).	No Dimensions

METRIC	METRIC DISPLAY NAME	UNIT	AGGREGATION TYPE	DESCRIPTION	DIMENSIONS
d2c .endpoints .egress .storage	Routing: messages delivered to storage	Count	Total	The number of times IoT Hub routing successfully delivered messages to storage endpoints.	No Dimensions
d2c .endpoints .latency .storage	Routing: message latency for storage	Milliseconds	Average	The average latency (milliseconds) between message ingress to IoT Hub and telemetry message ingress into a storage endpoint.	No Dimensions
d2c .endpoints .egress .storage .bytes	Routing: data delivered to storage	Bytes	Total	The amount of data (bytes) IoT Hub routing delivered to storage endpoints.	No Dimensions
d2c .endpoints .egress .storage .blobs	Routing: blobs delivered to storage	Count	Total	The number of times IoT Hub routing delivered blobs to storage endpoints.	No Dimensions
d2c .twin .read .success	Successful twin reads from devices	Count	Total	The count of all successful device-initiated twin reads.	No Dimensions
d2c .twin .read .failure	Failed twin reads from devices	Count	Total	The count of all failed device- initiated twin reads.	No Dimensions
d2c .twin .read .size	Response size of twin reads from devices	Bytes	Average	The average, min, and max of all successful device-initiated twin reads.	No Dimensions
d2c .twin .update .success	Successful twin updates from devices	Count	Total	The count of all successful device-initiated twin updates.	No Dimensions

METRIC	METRIC DISPLAY NAME	UNIT	AGGREGATION TYPE	DESCRIPTION	DIMENSIONS
d2c .twin .update .failure	Failed twin updates from devices	Count	Total	The count of all failed device-initiated twin updates.	No Dimensions
d2c .twin .update .size	Size of twin updates from devices	Bytes	Average	The average, min, and max size of all successful device-initiated twin updates.	No Dimensions
c2d .methods .success	Successful direct method invocations	Count	Total	The count of all successful direct method calls.	No Dimensions
c2d .methods .failure	Failed direct method invocations	Count	Total	The count of all failed direct method calls.	No Dimensions
c2d .methods .requestSize	Request size of direct method invocations	Bytes	Average	The average, min, and max of all successful direct method requests.	No Dimensions
c2d .methods .responseSize	Response size of direct method invocations	Bytes	Average	The average, min, and max of all successful direct method responses.	No Dimensions
c2d .twin .read .success	Successful twin reads from back end	Count	Total	The count of all successful back-end-initiated twin reads.	No Dimensions
c2d .twin .read .failure	Failed twin reads from back end	Count	Total	The count of all failed back-end-initiated twin reads.	No Dimensions
c2d .twin .read .size	Response size of twin reads from back end	Bytes	Average	The average, min, and max of all successful back-end-initiated twin reads.	No Dimensions
c2d .twin .update .success	Successful twin updates from back end	Count	Total	The count of all successful back-end-initiated twin updates.	No Dimensions

METRIC	METRIC DISPLAY NAME	UNIT	AGGREGATION TYPE	DESCRIPTION	DIMENSIONS
c2d .twin .update .failure	Failed twin updates from back end	Count	Total	The count of all failed back-end-initiated twin updates.	No Dimensions
c2d .twin .update .size	Size of twin updates from back end	Bytes	Average	The average, min, and max size of all successful back-end-initiated twin updates.	No Dimensions
twinQueries .success	Successful twin queries	Count	Total	The count of all successful twin queries.	No Dimensions
twinQueries .failure	Failed twin queries	Count	Total	The count of all failed twin queries.	No Dimensions
twinQueries .resultSize	Twin queries result size	Bytes	Average	The average, min, and max of the result size of all successful twin queries.	No Dimensions
jobs .createTwinUpdateJob .success	Successful creations of twin update jobs	Count	Total	The count of all successful creation of twin update jobs.	No Dimensions
jobs .createTwinUpdateJob .failure	Failed creations of twin update jobs	Count	Total	The count of all failed creation of twin update jobs.	No Dimensions
jobs .createDirectMethodJob .success	Successful creations of method invocation jobs	Count	Total	The count of all successful creation of direct method invocation jobs.	No Dimensions
jobs .createDirectMethodJob .failure	Failed creations of method invocation jobs	Count	Total	The count of all failed creation of direct method invocation jobs.	No Dimensions
jobs .listJobs .success	Successful calls to list jobs	Count	Total	The count of all successful calls to list jobs.	No Dimensions
jobs .listJobs .failure	Failed calls to list jobs	Count	Total	The count of all failed calls to list jobs.	No Dimensions

METRIC	METRIC DISPLAY NAME	UNIT	AGGREGATION TYPE	DESCRIPTION	DIMENSIONS
jobs.cancelJob.success	Successful job cancellations	Count	Total	The count of all successful calls to cancel a job.	No Dimensions
jobs.cancelJob.failure	Failed job cancellations	Count	Total	The count of all failed calls to cancel a job.	No Dimensions
jobs.queryJobs.success	Successful job queries	Count	Total	The count of all successful calls to query jobs.	No Dimensions
jobs.queryJobs.failure	Failed job queries	Count	Total	The count of all failed calls to query jobs.	No Dimensions
jobs.completed	Completed jobs	Count	Total	The count of all completed jobs.	No Dimensions
jobs.failed	Failed jobs	Count	Total	The count of all failed jobs.	No Dimensions
d2c.telemetry.ingress.sendThrottle	Number of throttling errors	Count	Total	Number of throttling errors due to device throughput throttles	No Dimensions
dailyMessage.QuotaUsed	Total number of messages used	Count	Average	Number of total messages used today. This is a cumulative value that is reset to zero at 00:00 UTC every day.	No Dimensions
deviceDataUsage	Total device data usage (deprecated)	Bytes	Total	Bytes transferred to and from any devices connected to IoT Hub	No Dimensions
deviceDataUsageV2	Total device data usage (preview)	Bytes	Total	Bytes transferred to and from any devices connected to IoT Hub	No Dimensions
totalDeviceCount	Total devices (preview)	Count	Average	Number of devices registered to your IoT hub	No Dimensions

METRIC	METRIC DISPLAY NAME	UNIT	AGGREGATION TYPE	DESCRIPTION	DIMENSIONS
connected DeviceCount	Connected devices (preview)	Count	Average	Number of devices connected to your IoT hub	No Dimensions
configurations	Configuration Metrics	Count	Total	Metrics for Configuration Operations	No Dimensions

Next steps

Now that you've seen an overview of IoT Hub metrics, follow this link to learn more about managing Azure IoT Hub:

- [Operations monitoring](#)

To further explore the capabilities of IoT Hub, see:

- [IoT Hub developer guide](#)
- [Deploying AI to edge devices with Azure IoT Edge](#)

Monitor the health of Azure IoT Hub and diagnose problems quickly

3/2/2019 • 11 minutes to read

Businesses that implement Azure IoT Hub expect reliable performance from their resources. To help you maintain a close watch on your operations, IoT Hub is fully integrated with [Azure Monitor](#) and [Azure Resource Health](#). These two services work to provide you with the data you need to keep your IoT solutions up and running in a healthy state.

Azure Monitor is a single source of monitoring and logging for all your Azure services. You can send the diagnostic logs that Azure Monitor generates to Azure Monitor logs, Event Hubs, or Azure Storage for custom processing. Azure Monitor's metrics and diagnostics settings give you visibility into the performance of your resources. Continue reading this article to learn how to [Use Azure Monitor](#) with your IoT hub.

IMPORTANT

The events emitted by the IoT Hub service using Azure Monitor diagnostic logs are not guaranteed to be reliable or ordered. Some events might be lost or delivered out of order. Diagnostic logs also aren't meant to be real-time, and it may take several minutes for events to be logged to your choice of destination.

Azure Resource Health helps you diagnose and get support when an Azure issue impacts your resources. A dashboard provides current and past health status for each of your IoT hubs. Continue to the section at the bottom of this article to learn how to [Use Azure Resource Health](#) with your IoT hub.

IoT Hub also provides its own metrics that you can use to understand the state of your IoT resources. To learn more, see [Understand IoT Hub metrics](#).

Use Azure Monitor

Azure Monitor provides diagnostics information for Azure resources, which means that you can monitor operations that take place within your IoT hub.

Azure Monitor's diagnostics settings replaces the IoT Hub operations monitor. If you currently use operations monitoring, you should migrate your workflows. For more information, see [Migrate from operations monitoring to diagnostics settings](#).

To learn more about the specific metrics and events that Azure Monitor watches, see [Supported metrics with Azure Monitor](#) and [Supported services, schemas, and categories for Azure Diagnostic Logs](#).

Enable logging with diagnostics settings

1. Sign in to the [Azure portal](#) and navigate to your IoT hub.
2. Select **Diagnostics settings**.
3. Select **Turn on diagnostics**.

Turn on diagnostics to collect the following data.

- Connections
- DeviceTelemetry
- C2DCommands
- DeviceIdentityOperations
- FileUploadOperations
- Routes
- D2CTwinOperations
- C2DTwinOperations
- Twin Queries
- JobsOperations
- DirectMethods
- AllMetrics

4. Give the diagnostic settings a name.

5. Choose where you want to send the logs. You can select any combination of the three options:

- Archive to a storage account
- Stream to an event hub
- Send to Log Analytics

6. Choose which operations you want to monitor, and enable logs for those operations. The operations that diagnostic settings can report on are:

- Connections
- Device telemetry
- Cloud-to-device messages
- Device identity operations
- File uploads
- Message routing
- Cloud-to-device twin operations
- Device-to-cloud twin operations
- Twin operations
- Job operations
- Direct methods
- Distributed tracing (preview)
- Configurations
- Device streams
- Device metrics

7. Save the new settings.

If you want to turn on diagnostics settings with PowerShell, use the following code:

```
Connect-AzureRmAccount  
Select-AzureRmSubscription -SubscriptionName <subscription that includes your IoT Hub>  
Set-AzureRmDiagnosticSetting -ResourceId <your resource Id> -ServiceBusRuleId <your service bus rule Id> -  
Enabled $true
```

New settings take effect in about 10 minutes. After that, logs appear in the configured archival target on the **Diagnostics settings** blade. For more information about configuring diagnostics, see [Collect and consume log](#)

data from your Azure resources.

Understand the logs

Azure Monitor tracks different operations that occur in IoT Hub. Each category has a schema that defines how events in that category are reported.

Connections

The connections category tracks device connect and disconnect events from an IoT hub as well as errors. This category is useful for identifying unauthorized connection attempts and or alerting when you lose connection to devices.

NOTE

For reliable connection status of devices check [Device heartbeat](#).

```
{  
  "records":  
  [  
    {  
      "time": " UTC timestamp",  
      "resourceId": "Resource Id",  
      "operationName": "deviceConnect",  
      "category": "Connections",  
      "level": "Information",  
      "properties": "{\"deviceId\":\"<deviceId>\",\"protocol\":\"<protocol>\",\"authType\":\"  
{\\\\\"scope\\\\\\\":\\\\\\\"device\\\\\\\",\\\\\\\"type\\\\\\\":\\\\\\\"sas\\\\\\\",\\\\\\\"issuer\\\\\\\":\\\\\\\"iothub\\\\\\\",\\\\\\\"acceptingIpFilterRule\\\\\\\":null\\\",\"maskedIpAddress\":\"<maskedIpAddress>\"}",  
      "location": "Resource location"  
    }  
  ]  
}
```

Cloud-to-device commands

The cloud-to-device commands category tracks errors that occur at the IoT hub and are related to the cloud-to-device message pipeline. This category includes errors that occur from:

- Sending cloud-to-device messages (like unauthorized sender errors),
- Receiving cloud-to-device messages (like delivery count exceeded errors), and
- Receiving cloud-to-device message feedback (like feedback expired errors).

This category does not catch errors when the cloud-to-device message is delivered successfully but then improperly handled by the device.

```
{
  "records":
  [
    {
      "time": " UTC timestamp",
      "resourceId": "Resource Id",
      "operationName": "messageExpired",
      "category": "C2DCommands",
      "level": "Error",
      "resultType": "Event status",
      "resultDescription": "MessageDescription",
      "properties": "{\"deviceId\":\"<deviceId>\",\" messageId\":[\"<messageId>\",\"<messageSizeInBytes>\",\"<protocol>\",\"Amqp\",\"<deliveryAcknowledgement>\":\"None, NegativeOnly, PositiveOnly, Full\",\"<deliveryCount>\":\"0\",\"<expiryTime>\":\"<timestamp>\",\"<timeInSystem>\":\"<timeInSystem>\",\"<ttl>\":<ttl>, \"<EventProcessedUtcTime>\":\"<UTC timestamp>\",\"<EventEnqueuedUtcTime>\":\"<UTC timestamp>\", \"<maskedIpAddress>\": \"<maskedIpAddress>\",\"<statusCode>\": \"4XX\"}"},
      "location": "Resource location"
    }
  ]
}
```

Device identity operations

The device identity operations category tracks errors that occur when you attempt to create, update, or delete an entry in your IoT hub's identity registry. Tracking this category is useful for provisioning scenarios.

```
{
  "records":
  [
    {
      "time": "UTC timestamp",
      "resourceId": "Resource Id",
      "operationName": "get",
      "category": "DeviceIdentityOperations",
      "level": "Error",
      "resultType": "Event status",
      "resultDescription": "MessageDescription",
      "properties": "{\"<maskedIpAddress>\":\"<maskedIpAddress>\",\"<deviceId>\":\"<deviceId>\",\"<statusCode>\":\"4XX\"}"},
      "location": "Resource location"
    }
  ]
}
```

Routes

The message routing category tracks errors that occur during message route evaluation and endpoint health as perceived by IoT Hub. This category includes events such as:

- A rule evaluates to "undefined",
- IoT Hub marks an endpoint as dead, or
- Any errors received from an endpoint.

This category does not include specific errors about the messages themselves (like device throttling errors), which are reported under the "device telemetry" category.

```
{
  "records": [
    {
      "time": "UTC timestamp",
      "resourceId": "Resource Id",
      "operationName": "endpointUnhealthy",
      "category": "Routes",
      "level": "Error",
      "properties": "{\"deviceId\": \"<deviceId>\", \"endpointName\": \"<endpointName>\", \"messageId\": <messageId>, \"details\": \"<errorDetails>\", \"routeName\": \"<routeName>\"}",
      "location": "Resource location"
    }
  ]
}
```

Device telemetry

The device telemetry category tracks errors that occur at the IoT hub and are related to the telemetry pipeline. This category includes errors that occur when sending telemetry events (such as throttling) and receiving telemetry events (such as unauthorized reader). This category cannot catch errors caused by code running on the device itself.

```
{
  "records": [
    {
      "time": "UTC timestamp",
      "resourceId": "Resource Id",
      "operationName": "ingress",
      "category": "DeviceTelemetry",
      "level": "Error",
      "resultType": "Event status",
      "resultDescription": "MessageDescription",
      "properties": "{\"deviceId\": \"<deviceId>\", \"batching\": \"0\", \"messageSizeInBytes\": \"<messageSizeInBytes>\", \"EventProcessedUtcTime\": \"<UTC timestamp>\", \"EventEnqueuedUtcTime\": \"<UTC timestamp>\", \"partitionId\": \"1\"}",
      "location": "Resource location"
    }
  ]
}
```

File upload operations

The file upload category tracks errors that occur at the IoT hub and are related to file upload functionality. This category includes:

- Errors that occur with the SAS URI, such as when it expires before a device notifies the hub of a completed upload.
- Failed uploads reported by the device.
- Errors that occur when a file is not found in storage during IoT Hub notification message creation.

This category cannot catch errors that directly occur while the device is uploading a file to storage.

```
{
  "records":
  [
    {
      "time": "UTC timestamp",
      "resourceId": "Resource Id",
      "operationName": "ingress",
      "category": "FileUploadOperations",
      "level": "Error",
      "resultType": "Event status",
      "resultDescription": "MessageDescription",
      "durationMs": "1",
      "properties": "{\"deviceId\":\"<deviceId>\",\"protocol\":\"<protocol>\",\"authType\":\"\\\n{\\\\\"scope\\\\\":\\\\\"device\\\\\",\\\\\"type\\\\\":\\\\\"sas\\\\\",\\\\\"issuer\\\\\":\\\\\"iothub\\\\\",\\\\\"acceptingIpFilterRule\\\\\":null}\",\"blobUri\":\"http://bloburi.com\"}",
      "location": "Resource location"
    }
  ]
}
```

Cloud-to-device twin operations

The cloud-to-device twin operations category tracks service-initiated events on device twins. These operations can include get twin, update or replace tags, and update or replace desired properties.

```
{
  "records":
  [
    {
      "time": "UTC timestamp",
      "resourceId": "Resource Id",
      "operationName": "read",
      "category": "C2DTwinOperations",
      "level": "Information",
      "durationMs": "1",
      "properties": "{\"deviceId\":\"<deviceId>\",\"sdkVersion\":\"<sdkVersion>\",\"messageSize\":\"<messageSize>\",\"authenticationType\":\"\\\n{\\\\\"scope\\\\\":\\\\\"device\\\\\",\\\\\"type\\\\\":\\\\\"sas\\\\\",\\\\\"issuer\\\\\":\\\\\"iothub\\\\\",\\\\\"acceptingIpFilterRule\\\\\":null}\",\"blobUri\":\"http://bloburi.com\"}",
      "location": "Resource location"
    }
  ]
}
```

Device-to-cloud twin operations

The device-to-cloud twin operations category tracks device-initiated events on device twins. These operations can include get twin, update reported properties, and subscribe to desired properties.

```
{
  "records":
  [
    {
      "time": "UTC timestamp",
      "resourceId": "Resource Id",
      "operationName": "update",
      "category": "D2CTwinOperations",
      "level": "Information",
      "durationMs": "1",
      "properties": "{\"deviceId\":\"<deviceId>\",\"protocol\":\"<protocol>\",\"authenticationType\":\"\\\n{\\\\\"scope\\\\\":\\\\\"device\\\\\",\\\\\"type\\\\\":\\\\\"sas\\\\\",\\\\\"issuer\\\\\":\\\\\"iothub\\\\\",\\\\\"acceptingIpFilterRule\\\\\":null}\",\"blobUri\":\"http://bloburi.com\"}",
      "location": "Resource location"
    }
  ]
}
```

Twin queries

The twin queries category reports on query requests for device twins that are initiated in the cloud.

```
{  
    "records":  
    [  
        {  
            "time": "UTC timestamp",  
            "resourceId": "Resource Id",  
            "operationName": "query",  
            "category": "TwinQueries",  
            "level": "Information",  
            "durationMs": "1",  
            "properties": "{\"query\":\"<twin query>\",\"sdkVersion\":\"<sdkVersion>\",\"messageSize\":\"<messageSize>\",\"pageSize\":\"<pageSize>\", \"continuation\":\"<true, false>\", \"resultSize\":\"<resultSize>\"}",  
            "location": "Resource location"  
        }  
    ]  
}
```

Jobs operations

The jobs operations category reports on job requests to update device twins or invoke direct methods on multiple devices. These requests are initiated in the cloud.

```
{  
    "records":  
    [  
        {  
            "time": "UTC timestamp",  
            "resourceId": "Resource Id",  
            "operationName": "jobCompleted",  
            "category": "JobsOperations",  
            "level": "Information",  
            "durationMs": "1",  
            "properties": "{\"jobId\":\"<jobId>\", \"sdkVersion\": \"<sdkVersion>\",\"messageSize\":<messageSize>, \"filter\":\"DeviceId IN ['1414ded9-b445-414d-89b9-e48e8c6285d5']\", \"startTimeUtc\":\"Wednesday, September 13, 2017\", \"duration\":\"0\"}",  
            "location": "Resource location"  
        }  
    ]  
}
```

Direct Methods

The direct methods category tracks request-response interactions sent to individual devices. These requests are initiated in the cloud.

```
{
  "records": [
    [
      {
        "time": "UTC timestamp",
        "resourceId": "Resource Id",
        "operationName": "send",
        "category": "DirectMethods",
        "level": "Information",
        "durationMs": "1",
        "properties": "{\"deviceId\":<messageSize>, \"RequestSize\": 1, \"ResponseSize\": 1,
\"sdkVersion\": \"2017-07-11\"}",
        "location": "Resource location"
      }
    ]
  }
}
```

Distributed Tracing (Preview)

The distributed tracing category tracks the correlation IDs for messages that carry the trace context header. To fully enable these logs, client-side code must be updated by following [Analyze and diagnose IoT applications end-to-end with IoT Hub distributed tracing \(preview\)](#).

Note that `correlationId` conforms to the [W3C Trace Context](#) proposal, where it contains a `trace-id` as well as a `span-id`.

IoT Hub D2C (device-to-cloud) logs

IoT Hub records this log when a message containing valid trace properties arrives at IoT Hub.

```
{
  "records": [
    [
      {
        "time": "UTC timestamp",
        "resourceId": "Resource Id",
        "operationName": "DiagnosticIoTHubD2C",
        "category": "DistributedTracing",
        "correlationId": "00-8cd869a412459a25f5b4f31311223344-0144d2590aacd909-01",
        "level": "Information",
        "resultType": "Success",
        "resultDescription": "Receive message success",
        "durationMs": "",
        "properties": "{\"messageSize\": 1, \"deviceId\":<deviceId>, \"callerLocalTimeUtc\": : \"2017-02-22T03:27:28.633Z\", \"calleeLocalTimeUtc\": \"2017-02-22T03:27:28.687Z\"}",
        "location": "Resource location"
      }
    ]
  }
}
```

Here, `durationMs` is not calculated as IoT Hub's clock might not be in sync with the device clock, and thus a duration calculation can be misleading. We recommend writing logic using the timestamps in the `properties` section to capture spikes in device-to-cloud latency.

PROPERTY	TYPE	DESCRIPTION
messageSize	Integer	The size of device-to-cloud message in bytes
deviceId	String of ASCII 7-bit alphanumeric characters	The identity of the device

PROPERTY	TYPE	DESCRIPTION
callerLocalTimeUtc	UTC timestamp	The creation time of the message as reported by the device local clock
calleeLocalTimeUtc	UTC timestamp	The time of message arrival at the IoT Hub's gateway as reported by IoT Hub service side clock

IoT Hub ingress logs

IoT Hub records this log when message containing valid trace properties writes to internal or built-in Event Hub.

```
{
  "records": [
    {
      "time": "UTC timestamp",
      "resourceId": "Resource Id",
      "operationName": "DiagnosticIoTHubIngress",
      "category": "DistributedTracing",
      "correlationId": "00-8cd869a412459a25f5b4f31311223344-349810a9bbd28730-01",
      "level": "Information",
      "resultType": "Success",
      "resultDescription": "Ingress message success",
      "durationMs": "10",
      "properties": "{\"isRoutingEnabled\": \"true\", \"parentSpanId\": \"0144d2590aacd909\"}",
      "location": "Resource location"
    }
  ]
}
```

In the `properties` section, this log contains additional information about message ingress

PROPERTY	TYPE	DESCRIPTION
isRoutingEnabled	String	Either true or false, indicates whether or not message routing is enabled in the IoT Hub
parentSpanId	String	The span-id of the parent message, which would be the D2C message trace in this case

IoT Hub egress logs

IoT Hub records this log when [routing](#) is enabled and the message is written to an [endpoint](#). If routing is not enabled, IoT Hub doesn't record this log.

```
{
  "records": [
    [
      {
        "time": "UTC timestamp",
        "resourceId": "Resource Id",
        "operationName": "DiagnosticIoTHubEgress",
        "category": "DistributedTracing",
        "correlationId": "00-8cd869a412459a25f5b4f31311223344-98ac3578922acd26-01",
        "level": "Information",
        "resultType": "Success",
        "resultDescription": "Egress message success",
        "durationMs": "10",
        "properties": "{\"endpointType\": \"EventHub\", \"endpointName\": \"myEventHub\", \"parentSpanId\": \"349810a9bbd28730\"}",
        "location": "Resource location"
      }
    ]
  }
}
```

In the `properties` section, this log contains additional information about message ingress

PROPERTY	TYPE	DESCRIPTION
<code>endpointName</code>	String	The name of the routing endpoint
<code>endpointType</code>	String	The type of the routing endpoint
<code>parentSpanId</code>	String	The span-id of the parent message, which would be the IoT Hub ingress message trace in this case

Read logs from Azure Event Hubs

After you set up event logging through diagnostics settings, you can create applications that read out the logs so that you can take action based on the information in them. This sample code retrieves logs from an event hub:

```

class Program
{
    static string connectionString = "{your AMS eventhub endpoint connection string}";
    static string monitoringEndpointName = "{your AMS event hub endpoint name}";
    static EventHubClient eventHubClient;
    //This is the Diagnostic Settings schema
    class AzureMonitorDiagnosticLog
    {
        string time { get; set; }
        string resourceId { get; set; }
        string operationName { get; set; }
        string category { get; set; }
        string level { get; set; }
        string resultType { get; set; }
        string resultDescription { get; set; }
        string durationMs { get; set; }
        string callerIpAddress { get; set; }
        string correlationId { get; set; }
        string identity { get; set; }
        string location { get; set; }
        Dictionary<string, string> properties { get; set; }
    };
    static void Main(string[] args)
    {
        Console.WriteLine("Monitoring. Press Enter key to exit.\n");
        eventHubClient = EventHubClient.CreateFromConnectionString(connectionString, monitoringEndpointName);
        var d2cPartitions = eventHubClient.GetRuntimeInformationAsync().PartitionIds;
        CancellationTokenSource cts = new CancellationTokenSource();
        var tasks = new List<Task>();
        foreach (string partition in d2cPartitions)
        {
            tasks.Add(ReceiveMessagesFromDeviceAsync(partition, cts.Token));
        }
        Console.ReadLine();
        Console.WriteLine("Exiting...");
        cts.Cancel();
        Task.WaitAll(tasks.ToArray());
    }
    private static async Task ReceiveMessagesFromDeviceAsync(string partition, CancellationToken ct)
    {
        var eventHubReceiver = eventHubClient.GetDefaultConsumerGroup().CreateReceiver(partition,
DateTime.UtcNow);
        while (true)
        {
            if (ct.IsCancellationRequested)
            {
                await eventHubReceiver.CloseAsync();
                break;
            }
            EventData eventData = await eventHubReceiver.ReceiveAsync(new TimeSpan(0,0,10));
            if (eventData != null)
            {
                string data = Encoding.UTF8.GetString(eventData.GetBytes());
                Console.WriteLine("Message received. Partition: {0} Data: '{1}'", partition, data);
                var deserializer = new JavaScriptSerializer();
                //deserialize json data to azure monitor object
                AzureMonitorDiagnosticLog message = new
JavaScriptSerializer().Deserialize<AzureMonitorDiagnosticLog>(result);

            }
        }
    }
}

```

Use Azure Resource Health

Use Azure Resource Health to monitor whether your IoT hub is up and running. You can also learn whether a regional outage is impacting the health of your IoT hub. To understand specific details about the health state of your Azure IoT Hub, we recommend that you [Use Azure Monitor](#).

Azure IoT Hub indicates health at a regional level. If a regional outage impacts your IoT hub, the health status shows as **Unknown**. To learn more, see [Resource types and health checks in Azure resource health](#).

To check the health of your IoT hubs, follow these steps:

1. Sign in to the [Azure portal](#).
2. Navigate to **Service Health > Resource health**.
3. From the drop-down boxes, select your subscription then select **IoT Hub** as the resource type.

To learn more about how to interpret health data, see [Azure resource health overview](#).

Next steps

- [Understand IoT Hub metrics](#)
- [IoT remote monitoring and notifications with Azure Logic Apps connecting your IoT hub and mailbox](#)

Set up X.509 security in your Azure IoT hub

2/28/2019 • 6 minutes to read

This tutorial simulates the steps you need to secure your Azure IoT hub using the *X.509 Certificate Authentication*. For the purpose of illustration, we will show how to use the open source tool OpenSSL to create certificates locally on your Windows machine. We recommend that you use this tutorial for test purposes only. For production environment, you should purchase the certificates from a *root certificate authority (CA)*.

Prerequisites

This tutorial requires that you have the following resources ready:

- You have created an IoT hub with your Azure subscription. See [Create an IoT hub through portal](#) for detailed steps.
- You have [Visual Studio 2015 or Visual Studio 2017](#) installed on your machine.

Get X.509 CA certificates

The X.509 certificate-based security in the IoT Hub requires you to start with an [X.509 certificate chain](#), which includes the root certificate as well as any intermediate certificates up until the leaf certificate.

You may choose either of the following ways to get your certificates:

- Purchase X.509 certificates from a *root certificate authority (CA)*. This is recommended for production environments. OR,
- Create your own X.509 certificates using a third party tool such as [OpenSSL](#). This will be fine for test and development purposes. See [Managing test CA certificates for samples and tutorials](#) for information about generating test CA certificates using PowerShell or Bash. The rest of this tutorial uses test CA certificates generated by following the instructions in [Managing test CA certificates for samples and tutorials](#).

Register X.509 CA certificates to your IoT hub

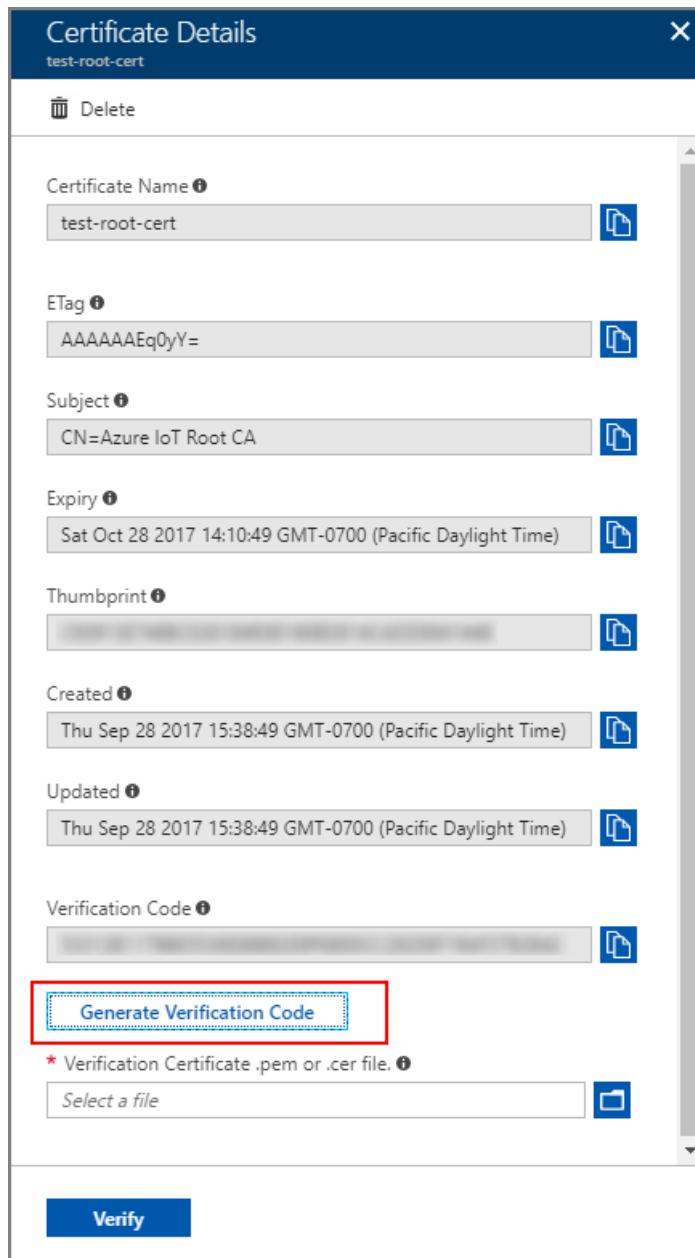
These steps show you how to add a new Certificate Authority to your IoT hub through the portal.

1. In the Azure portal, navigate to your IoT hub and open the **SETTINGS > Certificates** menu.
2. Click **Add** to add a new certificate.
3. Enter a friendly display name to your certificate. Select the root certificate file named *RootCA.cer* created in the previous section, from your machine. Click **Upload**.
4. Once you get a notification that your certificate is successfully uploaded, click **Save**.

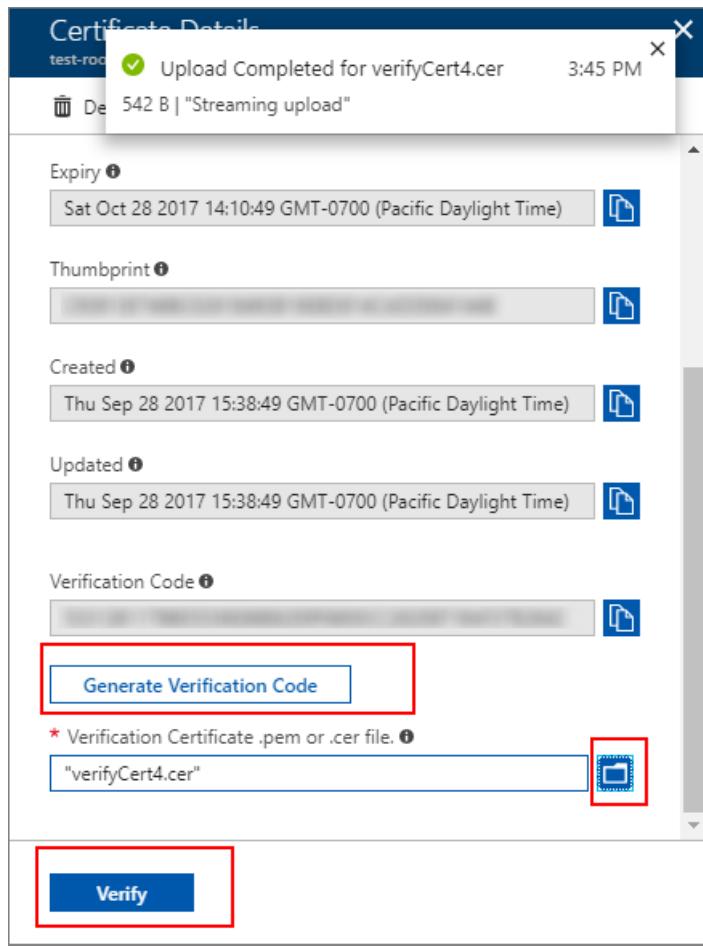
The screenshot shows the 'Certificates' blade in the Azure IoT Hub interface. On the left, a sidebar lists various management options like Overview, Activity log, Access control (IAM), Settings (Shared access policies, Pricing and scale, Operations monitoring, IP Filter, Certificates, Properties, Locks, Automation script), and Explorers (Device Explorer, Query Explorer). The 'Certificates' option is selected and highlighted in blue. The main area displays a table with columns: NAME, STATUS, EXPIRY, and SUBJECT. A message at the top states: 'You can use this tool to upload and manage your certificates.' Below the table, it says 'No results'. A modal window titled 'Add Certificate' is open in the top right corner, showing the status 'Upload Completed for RootCA.cer' and '449 B | "Streaming upload"'. It contains fields for 'Certificate Name' (set to 'test-root-cert') and 'Certificate .pem or .cer file' (set to '"RootCA.cer"'), both with green checkmarks indicating they are valid. A 'Save' button is at the bottom of the modal.

This will show your certificate in the **Certificate Explorer** list. Note the **STATUS** of this certificate is *Unverified*.

5. Click on the certificate that you added in the previous step.
6. In the **Certificate Details** blade, click **Generate Verification Code**.
7. It creates a **Verification Code** to validate the certificate ownership. Copy the code to your clipboard.



8. Now, you need to sign this *Verification Code* with the private key associate with your X.509 CA certificate, which generates a signature. There are tools available to perform this signing process, for example, OpenSSL. This is known as the [Proof of possession](#). Step 3 in [Managing test CA certificates for samples and tutorials](#) generates a verification code.
9. Upload the resulting signature from step 8 above to your IoT hub in the portal. In the **Certificate Details** blade on the Azure portal, navigate to the **Verification Certificate .pem or .cer file**, and select the signature, for example, *VerifyCert4.cer* created by the sample PowerShell command using the *File Explorer* icon besides it.
10. Once the certificate is successfully uploaded, click **Verify**. The **STATUS** of your certificate changes to **Verified** in the **Certificates** blade. Click **Refresh** if it does not update automatically.



Create an X.509 device for your IoT hub

1. In the Azure portal, navigate to your IoT hub's **Explorers > IoT devices** page.
2. Click **+ Add** to add a new device.
3. Give a friendly display name for the **Device ID**, and select **X.509 CA Signed** as the **Authentication Type**.
Click **Save**.

 Create a device □ X

 Learn more about creating devices ↗

* Device ID ⓘ
 ✓

Authentication type ⓘ
 Symmetric key X.509 Self-Signed X.509 CA Signed

Connect this device to an IoT hub ⓘ

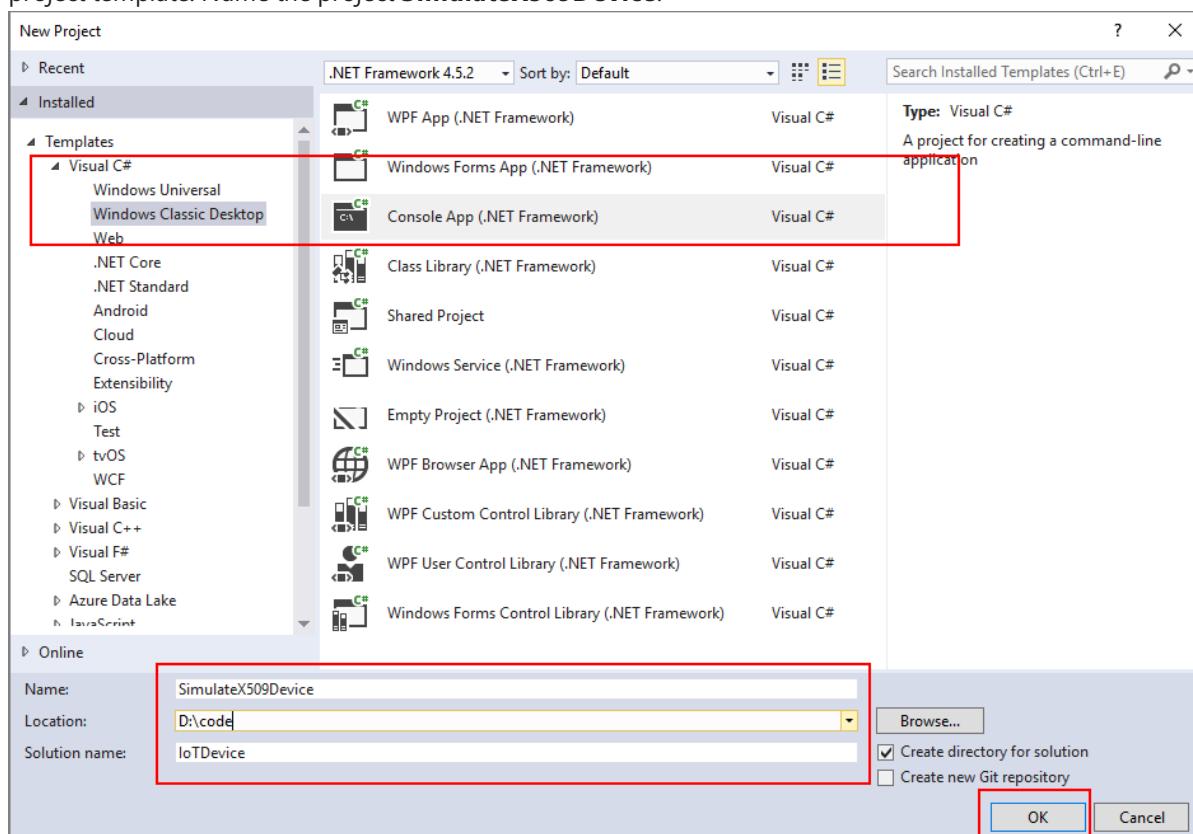
Parent device (Preview) ⓘ
No parent device

Authenticate your X.509 device with the X.509 certificates

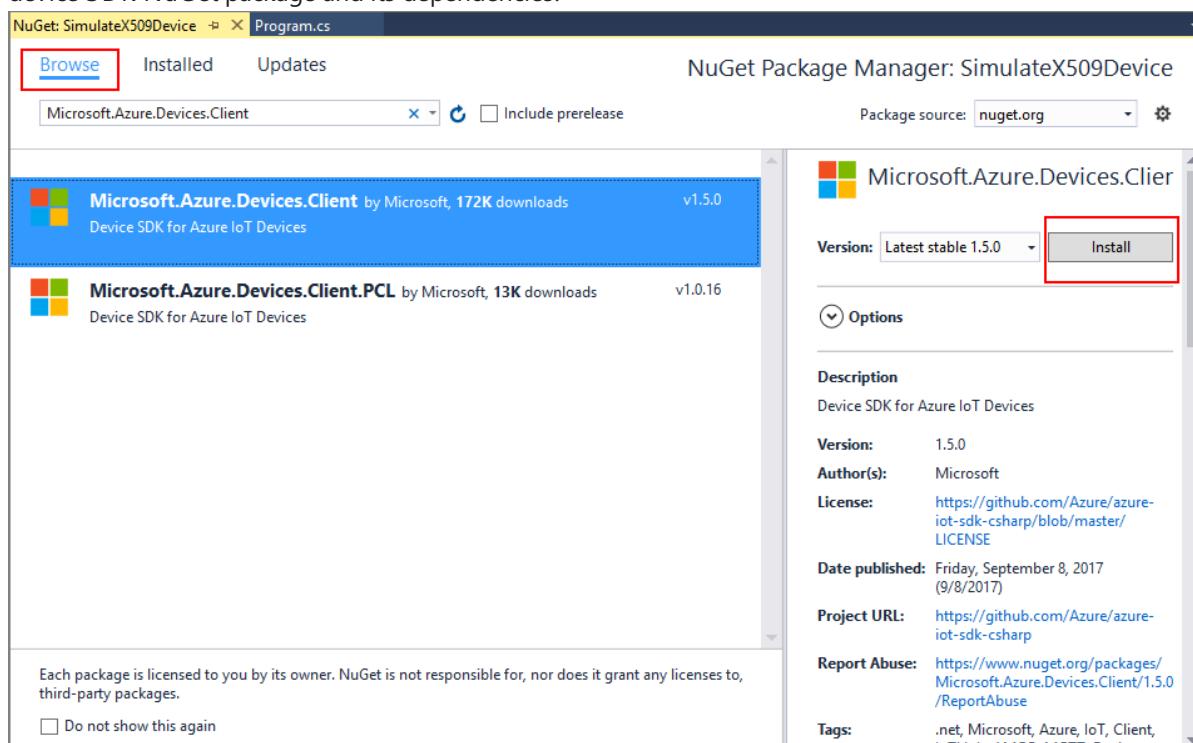
To authenticate your X.509 device, you need to first sign the device with the CA certificate. Signing of leaf devices is normally done at the manufacturing plant, where manufacturing tools have been enabled accordingly. As the device goes from one manufacturer to another, each manufacturer's signing action is captured as an intermediate certificate within the chain. The end result is a certificate chain from the CA certificate to the device's leaf certificate. Step 4 in [Managing test CA certificates for samples and tutorials](#) generates a device certificate.

Next, we will show you how to create a C# application to simulate the X.509 device registered for your IoT hub. We will send temperature and humidity values from the simulated device to your hub. Note that in this tutorial, we will create only the device application. It is left as an exercise to the readers to create the IoT Hub service application that will send response to the events sent by this simulated device. The C# application assumes that you have followed the steps in [Managing test CA certificates for samples and tutorials](#).

1. In Visual Studio, create a new Visual C# Windows Classic Desktop project by using the Console Application project template. Name the project **SimulateX509Device**.



2. In Solution Explorer, right-click the **SimulateX509Device** project, and then click **Manage NuGet Packages....** In the NuGet Package Manager window, select **Browse** and search for **microsoft.azure.devices.client**. Select **Install** to install the **Microsoft.Azure.Devices.Client** package, and accept the terms of use. This procedure downloads, installs, and adds a reference to the Azure IoT device SDK NuGet package and its dependencies.



3. Add the following lines of code at the top of the *Program.cs* file:

```
using Microsoft.Azure.Devices.Client;
using Microsoft.Azure.Devices.Shared;
using System.Security.Cryptography.X509Certificates;
```

4. Add the following lines of code inside the **Program** class:

```
private static int MESSAGE_COUNT = 5;
private const int TEMPERATURE_THRESHOLD = 30;
private static String deviceId = "<your-device-id>";
private static float temperature;
private static float humidity;
private static Random rnd = new Random();
```

Use the friendly device name you used in the preceding section in place of *<your_device_id>* placeholder.

5. Add the following function to create random numbers for temperature and humidity and send these values to the hub:

```
static async Task SendEvent(DeviceClient deviceClient)
{
    string dataBuffer;
    Console.WriteLine("Device sending {0} messages to IoTHub...\n", MESSAGE_COUNT);

    for (int count = 0; count < MESSAGE_COUNT; count++)
    {
        temperature = rnd.Next(20, 35);
        humidity = rnd.Next(60, 80);
        dataBuffer = string.Format("{{\"deviceId\":\"{0}\",\" messageId\":{1},\"temperature\":{2},\"humidity\":{3}}}", deviceId, count, temperature, humidity);
        Message eventMessage = new Message(Encoding.UTF8.GetBytes(dataBuffer));
        eventMessage.Properties.Add("temperatureAlert", (temperature > TEMPERATURE_THRESHOLD) ? "true" : "false");
        Console.WriteLine("\t{0}> Sending message: {1}, Data: [{2}]", DateTime.NowToLocalTime(), count, dataBuffer);

        await deviceClient.SendEventAsync(eventMessage);
    }
}
```

6. Finally, add the following lines of code to the **Main** function, replacing the placeholders *device-id*, *your-iot-hub-name* and *absolute-path-to-your-device-pfx-file* as required by your setup.

```

try
{
    var cert = new X509Certificate2(@"<absolute-path-to-your-device-pfx-file>", "1234");
    var auth = new DeviceAuthenticationWithX509Certificate("<device-id>", cert);
    var deviceClient = DeviceClient.Create("<your-iot-hub-name>.azure-devices.net", auth,
    TransportType.Amqp_Tcp_Only);

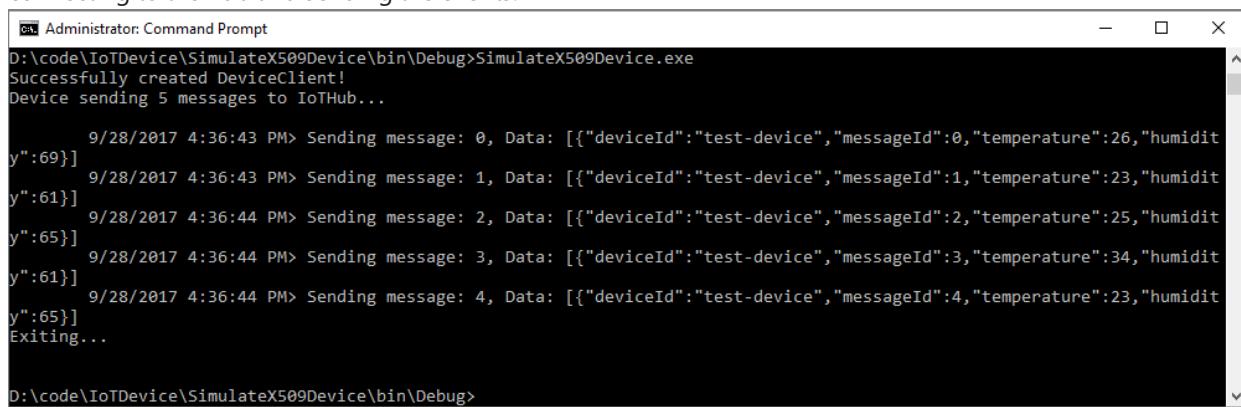
    if (deviceClient == null)
    {
        Console.WriteLine("Failed to create DeviceClient!");
    }
    else
    {
        Console.WriteLine("Successfully created DeviceClient!");
        SendEvent(deviceClient).Wait();
    }

    Console.WriteLine("Exiting...\n");
}
catch (Exception ex)
{
    Console.WriteLine("Error in sample: {0}", ex.Message);
}

```

This code connects to your IoT hub by creating the connection string for your X.509 device. Once successfully connected, it then sends temperature and humidity events to the hub, and waits for its response.

7. Since this application accesses a *.pfx* file, you may need to execute this in *Admin* mode. Build the Visual Studio solution. Open a new command window as an **Administrator**, and navigate to the folder containing this solution. Navigate to the *bin/Debug* path within the solution folder. Run the application **SimulateX509Device.exe** from the *Admin* command window. You should see your device successfully connecting to the hub and sending the events.



```

Administrator: Command Prompt
D:\code\IoTDevice\SimulateX509Device\bin\Debug>SimulateX509Device.exe
Successfully created DeviceClient!
Device sending 5 messages to IoTHub...
9/28/2017 4:36:43 PM> Sending message: 0, Data: [{"deviceId": "test-device", "messageId": 0, "temperature": 26, "humidity": 69}]
9/28/2017 4:36:43 PM> Sending message: 1, Data: [{"deviceId": "test-device", "messageId": 1, "temperature": 23, "humidity": 61}]
9/28/2017 4:36:44 PM> Sending message: 2, Data: [{"deviceId": "test-device", "messageId": 2, "temperature": 25, "humidity": 65}]
9/28/2017 4:36:44 PM> Sending message: 3, Data: [{"deviceId": "test-device", "messageId": 3, "temperature": 34, "humidity": 61}]
9/28/2017 4:36:44 PM> Sending message: 4, Data: [{"deviceId": "test-device", "messageId": 4, "temperature": 23, "humidity": 65}]
Exiting...
D:\code\IoTDevice\SimulateX509Device\bin\Debug>

```

See also

To learn more about securing your IoT solution, see:

- [IoT Security Best Practices](#)
- [IoT Security Architecture](#)
- [Secure your IoT deployment](#)

To further explore the capabilities of IoT Hub, see:

- [Deploying AI to edge devices with Azure IoT Edge](#)

How to upgrade your IoT hub

2/27/2019 • 2 minutes to read

As your IoT solution grows, Azure IoT Hub is ready to help you scale up. Azure IoT Hub offers two tiers, basic (B) and standard (S), to accommodate customers that want to use different features. Within each tier are three sizes (1, 2, and 3) that determine the number of messages that can be sent each day.

When you have more devices and need more capabilities, there are three ways to adjust your IoT hub to suit your needs:

- Add units within the IoT hub. For example, each additional unit in a B1 IoT hub allows for an additional 400,000 messages per day.
- Change the size of the IoT hub. For example, migrate from the B1 tier to the B2 tier to increase the number of messages that each unit can support per day.
- Upgrade to a higher tier. For example, upgrade from the B1 tier to the S1 tier for access to advanced features with the same messaging capacity.

These changes can all occur without interrupting existing operations.

If you want to downgrade your IoT hub, you can remove units and reduce the size of the IoT hub but you cannot downgrade to a lower tier. For example, you can move from the S2 tier to the S1 tier, but not from the S2 tier to the B1 tier. Only one type of [edition](#) within a tier can be chosen per IoT Hub. For example, you can create an IoT Hub with multiple units of S1, but not with a mix of units from different editions, such as S1 and B3, or S1 and S2.

These examples are meant to help you understand how to adjust your IoT hub as your solution changes. For specific information about each tier's capabilities, you should always refer to [Azure IoT Hub pricing](#).

Upgrade your existing IoT hub

1. Sign in to the [Azure portal](#) and navigate to your IoT hub.
2. Select **Pricing and scale**.

The screenshot shows the Azure IoT Hub Overview page for 'MyIoTHub-WestUS'. The left sidebar lists several settings: Overview, Activity log, Access control (IAM), Tags, SETTINGS, Shared access policies, Pricing and scale (which is highlighted with a red box), Operations monitoring, IP Filter, Certificates, Properties, Locks, and Automation script.

3. To change the tier for your hub, select **Pricing and scale tier**. Choose the new tier, then click **select**.

The screenshot shows the 'Pricing and scale' configuration page for 'MyIoTHub-WestUS'. The left sidebar shows the same settings as the previous screenshot. On the right, there is a message: 'IoT Hub is billed by reserved units. The maximum number of messages an IoT Hub can process per day is determined by the tier selected and number of units.' Below this, the 'Pricing and scale tier' is set to 'S1 - Standard' (highlighted with a red box). The 'IoT Hub units' field shows the value '1'.

4. To change the number of units in your hub, enter a new value under **IoT Hub units**.

5. Select **Save** to save your changes.

Your IoT hub is now adjusted, and your configurations are unchanged.

The maximum partition limit for basic tier IoT Hub and standard tier IoT Hub is 32. Most IoT Hubs only need 4 partitions. The partition limit is chosen when IoT Hub is created, and relates the device-to-cloud messages to the number of simultaneous readers of these messages. This value remains unchanged when you migrate from basic tier to standard tier.

Next steps

Get more details about [How to choose the right IoT Hub tier.](#)

Trace Azure IoT device-to-cloud messages with distributed tracing (preview)

2/26/2019 • 10 minutes to read

Microsoft Azure IoT Hub currently supports distributed tracing as a [preview feature](#).

IoT Hub is one of the first Azure services to support distributed tracing. As more Azure services support distributed tracing, you'll be able trace IoT messages throughout the Azure services involved in your solution. For a background on distributed tracing, see [Distributed Tracing](#).

Enabling distributed tracing for IoT Hub gives you the ability to:

- Precisely monitor the flow of each message through IoT Hub using [trace context](#). This trace context includes correlation IDs that allow you to correlate events from one component with events from another component. It can be applied for a subset or all IoT device messages using [device twin](#).
- Automatically log the trace context to [Azure Monitor diagnostic logs](#).
- Measure and understand message flow and latency from devices to IoT Hub and routing endpoints.
- Start considering how you want to implement distributed tracing for the non-Azure services in your IoT solution.

In this article, you use the [Azure IoT device SDK for C](#) with distributed tracing. Distributed tracing support is still in progress for the other SDKs.

Prerequisites

- The preview of distributed tracing is currently only supported for IoT Hubs created in the following regions:
 - **North Europe**
 - **Southeast Asia**
 - **West US 2**
- This article assumes that you're familiar with sending telemetry messages to your IoT hub. Make sure you've completed the [Send telemetry C Quickstart](#).
- Register a device with your IoT hub (steps available in each Quickstart) and note down the connection string.
- Install the latest version of [Git](#).

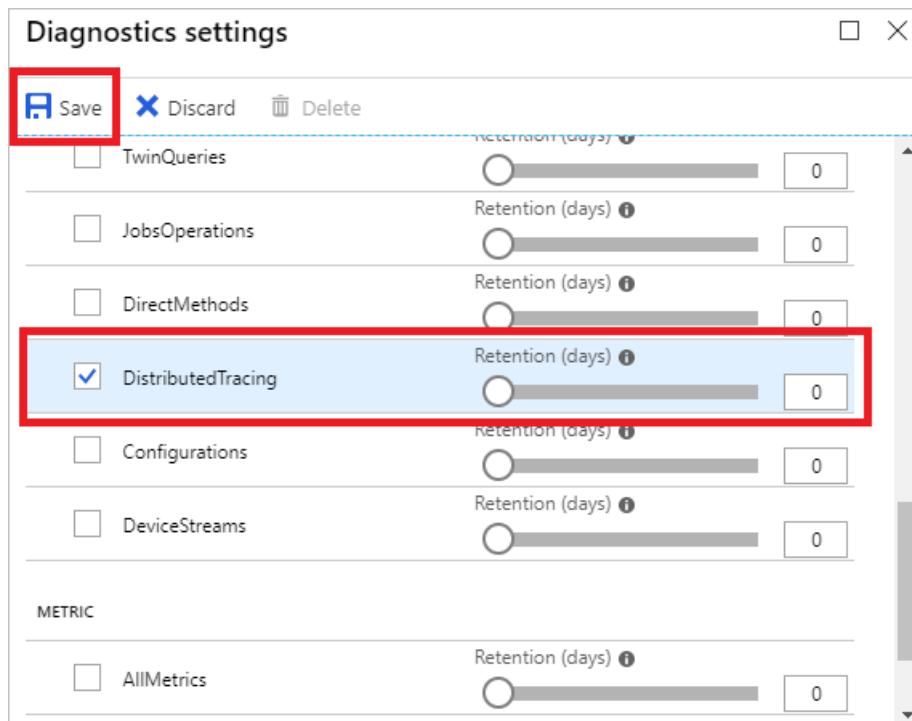
Configure IoT Hub

In this section, you configure an IoT Hub to log distributed tracing attributes (correlation IDs and timestamps).

1. Navigate to your IoT hub in the [Azure portal](#).
2. In the left pane for your IoT hub, scroll down to the **Monitoring** section and click **Diagnostics settings**.
3. If diagnostic settings aren't already turned on, click **Turn on diagnostics**. If you have already enabled diagnostic settings, click **Add diagnostic setting**.
4. In the **Name** field, enter a name for a new diagnostic setting. For example, **DistributedTracingSettings**.
5. Choose one or more of the following options that determine where the logging will be sent:

- **Archive to a storage account:** Configure a storage account to contain the logging information.
 - **Stream to an event hub:** Configure an event hub to contain the logging information.
 - **Send to Log Analytics:** Configure a log analytics workspace to contain the logging information.
6. In the **Log** section, select the operations that you want logging information for.

Make sure to include **DistributedTracing**, and configure a **Retention** for how many days you want the logging retained. Log retention does affect storage costs.



7. Click **Save** for the new setting.
8. (Optional) To see the messages flow to different places, set up [routing rules to at least two different endpoints](#).

Once the logging is turned on, IoT Hub records a log when a message containing valid trace properties is encountered in any of the following situations:

- The messages arrives at IoT Hub's gateway.
- The message is processed by the IoT Hub.
- The message is routed to custom endpoints. Routing must be enabled.

To learn more about these logs and their schemas, see [Distributed tracing in IoT Hub diagnostic logs](#).

Set up device

In this section, you prepare a development environment for use with the [Azure IoT C SDK](#). Then, you modify one of samples to enable distributed tracing on your device's telemetry messages.

These instructions are for building the sample on Windows. For other environments, see [Compile the C SDK](#) or [Prepackaged C SDK for Platform Specific Development](#).

Clone the source code and initialize

1. Install "[Desktop development with C++ workload](#)" for either Visual Studio 2015 or 2017.
2. Install [CMake](#). Make sure it is in your `PATH` by typing `cmake -version` from a command prompt.
3. Open a command prompt or Git Bash shell. Execute the following command to clone the [Azure IoT C SDK](#) GitHub repository:

```
git clone https://github.com/Azure/azure-iot-sdk-c.git --recursive -b public-preview
```

The size of this repository is currently around 220 MB. You should expect this operation to take several minutes to complete.

4. Create a `cmake` subdirectory in the root directory of the git repository, and navigate to that folder.

```
cd azure-iot-sdk-c  
mkdir cmake  
cd cmake  
cmake ..
```

If `cmake` can't find your C++ compiler, you might get build errors while running the above command. If that happens, try running this command in the [Visual Studio command prompt](#).

Once the build succeeds, the last few output lines will look similar to the following output:

```
$ cmake ..  
-- Building for: Visual Studio 15 2017  
-- Selecting Windows SDK version 10.0.16299.0 to target Windows 10.0.17134.  
-- The C compiler identification is MSVC 19.12.25835.0  
-- The CXX compiler identification is MSVC 19.12.25835.0  
  
...  
  
-- Configuring done  
-- Generating done  
-- Build files have been written to: E:/IoT Testing/azure-iot-sdk-c/cmake
```

Edit the send telemetry sample to enable distributed tracing

1. Use an editor to open the

```
azure-iot-sdk-c/iothub_client/samples/iothub_ll_telemetry_sample/iothub_ll_telemetry_sample.c
```

 source file.

2. Find the declaration of the `connectionString` constant:

```
/* Paste in the your iothub connection string */  
static const char* connectionString = "[device connection string]";  
#define MESSAGE_COUNT      5000  
static bool g_continueRunning = true;  
static size_t g_message_count_send_confirmations = 0;
```

Replace the value of the `connectionString` constant with the device connection string you made a note of in the [register a device](#) section of the [Send telemetry C Quickstart](#).

3. Change the `MESSAGE_COUNT` define to `5000`:

```
/* Paste in the your iothub connection string */  
static const char* connectionString = "[device connection string]";  
#define MESSAGE_COUNT      5000  
static bool g_continueRunning = true;  
static size_t g_message_count_send_confirmations = 0;
```

4. Find the line of code that calls `IoTHubDeviceClient_LL_SetConnectionStatusCallback` to register a connection status callback function before the send message loop. Add code under that line as shown below to call `IoTHubDeviceClient_LL_EnablePolicyConfiguration` enabling distributed tracing for the device:

```

// Setting connection status callback to get indication of connection to iothub
(void)IoTHubDeviceClient_LL_SetConnectionStatusCallback(device_ll_handle, connection_status_callback,
NULL);

// Enabled the distributed tracing policy for the device
(void)IoTHubDeviceClient_LL_EnablePolicyConfiguration(device_ll_handle,
POLICY_CONFIGURATION_DISTRIBUTED_TRACING, true);

do
{
    if (messages_sent < MESSAGE_COUNT)

```

The `IoTHubDeviceClient_LL_EnablePolicyConfiguration` function enables policies for specific IoT Hub features that are configured via [device twins](#). Once `POLICY_CONFIGURATION_DISTRIBUTED_TRACING` is enabled with the line of code above, the tracing behavior of the device will reflect distributed tracing changes made on the device twin.

- To keep the sample app running without using up all your quota, add a one-second delay at the end of the send message loop:

```

        else if (g_message_count_send_confirmations >= MESSAGE_COUNT)
    {
        // After all messages are all received stop running
        g_continueRunning = false;
    }

    IoTHubDeviceClient_LL_DoWork(device_ll_handle);
    ThreadAPI_Sleep(1000);

} while (g_continueRunning);

```

Compile and run

- Navigate to the `iothub_ll_telemetry_sample` project directory from the CMake directory (`azure-iot-sdk-c/cmake`) you created earlier, and compile the sample:

```

cd iothub_client/samples/iothub_ll_telemetry_sample
cmake --build . --target iothub_ll_telemetry_sample --config Debug

```

- Run the application. The device sends telemetry supporting distributed tracing.

```

Debug/iothub_ll_telemetry_sample.exe

```

- Keep the app running. Optionally observe the message being sent to IoT Hub by looking at the console window.

Using third-party clients

If you're not using the C SDK and still would like to preview distributed tracing for IoT Hub, construct the message to contain a `tracestate` application property with the creation time of the message in the unix timestamp format. For example, `tracestate=timestamp=1539243209`. To control the percentage of messages containing this property, implement logic to listen to cloud-initiated events such as twin updates.

Update sampling options

To change the percentage of messages to be traced from the cloud, you must update the device twin. You can accomplish this multiple ways including the JSON editor in portal and the IoT Hub service SDK. The following

subsections provide examples.

Update using the portal

1. Navigate to your IoT hub in [Azure portal](#), then click **IoT devices**.
2. Click your device.
3. Look for **Enable distributed tracing (preview)**, then select **Enable**.

The screenshot shows the 'Device details' page for a device named 'demoDevice'. The 'Save' button is highlighted with a red box. Below it, there are fields for Device Id ('demoDevice'), Primary key, Secondary key, Connection string (primary key), and Connection string (secondary key). Under 'Connect this device to an IoT hub', the 'Enable' radio button is selected. In the 'Parent device (Preview)' section, it says 'No parent device' and 'Set a parent device'. The 'Enable distributed tracing' section is also highlighted with a red box; its 'Enable' radio button is selected. A slider for 'Sampling rate' is set to 50, with a range from 0 to 100. To the right of the slider are two green circular icons with checkmarks.

4. Choose a **Sampling rate** between 0% and 100%.
5. Click **Save**.
6. Wait a few seconds, and hit **Refresh**, then if successfully acknowledged by device, a sync icon with a checkmark appears.
7. Go back to the console window for the telemetry message app. You will see messages being sent with **tracestate** in the application properties.

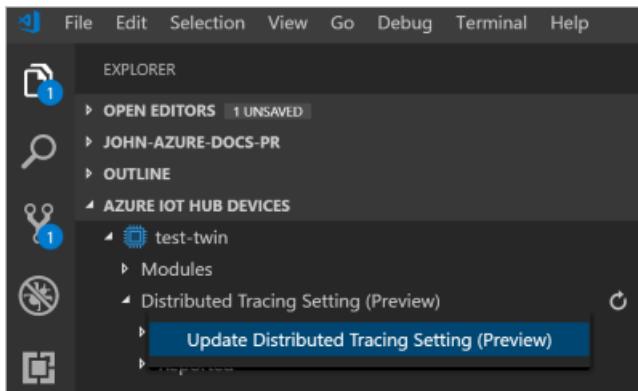
The terminal window shows several lines of IoT Hub telemetry messages. The messages are PUBLISH commands with the following properties:
- Property 'key': 'property_value'
- Property 'key': 'property_value&%24.tracestate=timestamp%3d1547666028'
- Property 'key': 'property_value&%24.tracestate=timestamp%3d1547666030'
- Property 'key': 'property_value&%24.tracestate=timestamp%3d1547666032'
Each message has a corresponding PUBACK message with the same properties. The messages are timestamped at approximately 11:13:47, 11:13:48, 11:13:49, 11:13:50, and 11:13:51.

8. (Optional) Change the sampling rate to a different value, and observe the change in frequency that

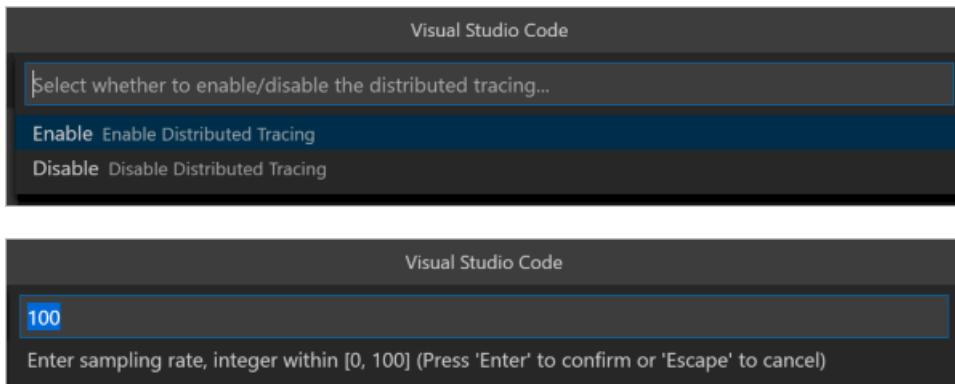
messages include `tracestate` in the application properties.

Update using Azure IoT Hub Toolkit for VS Code

1. Install VS Code, then install the latest version of Azure IoT Hub Toolkit for VS Code from [here](#).
2. Open VS Code and [set up IoT Hub connection string](#).
3. Expand the device and look for **Distributed Tracing Setting (Preview)**. Under it, click **Update Distributed Tracing Setting (Preview)** of sub node.



4. In the popup window, select **Enable**, then press Enter to confirm 100 as sampling rate.



Bulk update for multiple devices

To update the distributed tracing sampling configuration for multiple devices, use [automatic device configuration](#). Make sure you follow this twin schema:

```
{  
  "properties": {  
    "desired": {  
      "azureiot*com^dtracing^1": {  
        "sampling_mode": 1,  
        "sampling_rate": 100  
      }  
    }  
  }  
}
```

ELEMENT NAME	REQUIRED	TYPE	DESCRIPTION
<code>sampling_mode</code>	Yes	Integer	Two mode values are currently supported to turn sampling on and off. <code>1</code> is On and, <code>2</code> is Off.

Element Name	Required	Type	Description
sampling_rate	Yes	Integer	This value is a percentage. Only values from <code>0</code> to <code>100</code> (inclusive) are permitted.

Query and visualize

To see all the traces logged by an IoT Hub, query the log store that you selected in diagnostic settings. This section walks through a couple different options.

Query using Log Analytics

If you've set up [Log Analytics with diagnostic logs](#), query by looking for logs in the `DistributedTracing` category.

For example, this query shows all the traces logged:

```
// All distributed traces
AzureDiagnostics
| where Category == "DistributedTracing"
| project TimeGenerated, Category, OperationName, Level, CorrelationId, DurationMs, properties_s
| order by TimeGenerated asc
```

Example logs as shown by Log Analytics:

TimeGenerated	OperationName	Category	Level	CorrelationId	DurationMs	Properties
2018-02-22T03:28:28.633Z	DiagnosticIoTHubD2C	DistributedTracing	Informational	00-8cd869a412459a25f5b4f31311223344-0144d2590aacd909-01		{"deviceId":"AZ3166","messageSize":"96","callerLocalTimeUtc":"2018-02-22T03:27:28.633Z","calleeLocalTimeUtc":"2018-02-22T03:27:28.687Z"}
2018-02-22T03:28:38.633Z	DiagnosticIoTHubIngress	DistributedTracing	Informational	00-8cd869a412459a25f5b4f31311223344-349810a9bbd28730-01	20	{"isRoutingEnabled":"false","parentSpanId":"0144d2590aacd909"}
2018-02-22T03:28:48.633Z	DiagnosticIoTHubEgress	DistributedTracing	Informational	00-8cd869a412459a25f5b4f31311223344-349810a9bbd28730-01	23	{"endpointType":"EventHub","endpointName":"myEventHub","parentSpanId":"0144d2590aacd909"}

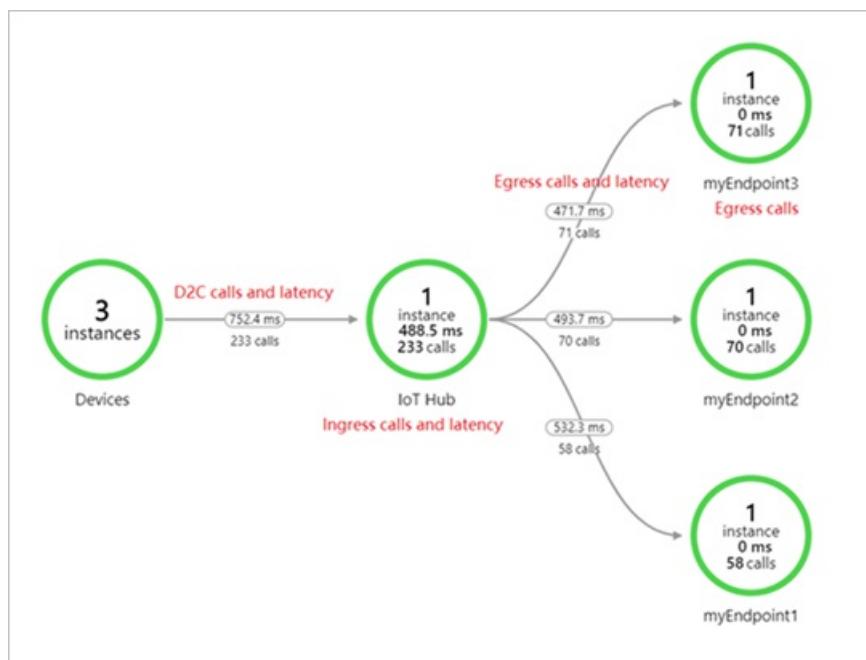
To understand the different types of logs, see [Azure IoT Hub diagnostic logs](#).

Application Map

To visualize the flow of IoT messages, set up the Application Map sample app. The sample app sends the distributed tracing logs to [Application Map](#) using an Azure Function and an Event Hub.

[GET THE SAMPLE ON
GITHUB](#)

This image below shows distributed tracing in App Map with three routing endpoints:



Understand Azure IoT distributed tracing

Context

Many IoT solutions, including our own [reference architecture](#) (English only), generally follow a variant of the [microservice architecture](#). As an IoT solution grows more complex, you end up using a dozen or more microservices. These microservices may or may not be from Azure. Pinpointing where IoT messages are dropping or slowing down can become challenging. For example, you have an IoT solution that uses 5 different Azure services and 1500 active devices. Each device sends 10 device-to-cloud messages/second (for a total of 15,000 messages/second), but you notice that your web app sees only 10,000 messages/second. Where is the issue? How do you find the culprit?

Distributed tracing pattern in microservice architecture

To reconstruct the flow of an IoT message across different services, each service should propagate a *correlation ID* that uniquely identifies the message. Once collected in a centralized system, correlation IDs enable you to see message flow. This method is called the [distributed tracing pattern](#).

To support wider adoption for distributed tracing, Microsoft is contributing to [W3C standard proposal for distributed tracing](#).

IoT Hub support

Once enabled, distributed tracing support for IoT Hub will follow this flow:

1. A message is generated on the IoT device.
2. The IoT device decides (with help from cloud) that this message should be assigned with a trace context.
3. The SDK adds a `tracestate` to the message application property, containing the message creation timestamp.
4. The IoT device sends the message to IoT Hub.
5. The message arrives at IoT hub gateway.

6. IoT Hub looks for the `tracestate` in the message application properties, and checks to see if it's in the correct format.
7. If so, IoT Hub generates and logs the `trace-id` and `span-id` to Azure Monitor diagnostic logs under the category `DiagnosticIoTHubD2C`.
8. Once the message processing is finished, IoT Hub generates another `span-id` and logs it along with the existing `trace-id` under the category `DiagnosticIoTHubIngress`.
9. If routing is enabled for the message, IoT Hub writes it to the custom endpoint, and logs another `span-id` with the same `trace-id` under the category `DiagnosticIoTHubEgress`.
10. The steps above are repeated for each message generated.

Public preview limits and considerations

- Proposal for W3C Trace Context standard is currently a working draft.
- Currently, the only development language supported by client SDK is C.
- Cloud-to-device twin capability isn't available for [IoT Hub basic tier](#). However, IoT Hub will still log to Azure Monitor if it sees a properly composed trace context header.
- To ensure efficient operation, IoT Hub will impose a throttle on the rate of logging that can occur as part of distributed tracing.

Next steps

- To learn more about the general distributed tracing pattern in microservices, see [Microservice architecture pattern: distributed tracing](#).
- To set up configuration to apply distributed tracing settings to a large number of devices, see [Configure and monitor IoT devices at scale](#).
- To learn more about Azure Monitor, see [What is Azure Monitor?](#).

Use IP filters

2/22/2019 • 4 minutes to read

Security is an important aspect of any IoT solution based on Azure IoT Hub. Sometimes you need to explicitly specify the IP addresses from which devices can connect as part of your security configuration. The *IP filter* feature enables you to configure rules for rejecting or accepting traffic from specific IPv4 addresses.

When to use

There are two specific use-cases when it is useful to block the IoT Hub endpoints for certain IP addresses:

- Your IoT hub should receive traffic only from a specified range of IP addresses and reject everything else. For example, you are using your IoT hub with [Azure Express Route](#) to create private connections between an IoT hub and your on-premises infrastructure.
- You need to reject traffic from IP addresses that have been identified as suspicious by the IoT hub administrator.

How filter rules are applied

The IP filter rules are applied at the IoT Hub service level. Therefore the IP filter rules apply to all connections from devices and back-end apps using any supported protocol.

Any connection attempt from an IP address that matches a rejecting IP rule in your IoT hub receives an unauthorized 401 status code and description. The response message does not mention the IP rule.

Default setting

By default, the **IP Filter** grid in the portal for an IoT hub is empty. This default setting means that your hub accepts connections from any IP address. This default setting is equivalent to a rule that accepts the 0.0.0.0/0 IP address range.

The screenshot shows the 'getStartedWithAnIoTHub - IP Filter' blade in the Azure portal. On the left, a navigation menu lists 'Overview', 'Activity log', 'Access control (IAM)', 'Device Explorer', 'SETTINGS' (with 'Shared access policies', 'Pricing and scale', 'Operations monitoring'), and 'IP Filter'. The 'IP Filter' item is highlighted with a red box. The main area contains a table header for 'IP FILTER RULE NAME', 'ACTION', and 'IPV4 ADDRESS RANGE'. Below the header, a message states: 'If the table is empty or no rule matches, the connection is accepted. Rules are applied in order: the first matching rule decides the action. To change the order of the IP filter rules, hover over the row to drag and drop to the desired location in the grid'. A note below says 'No results'.

Add or edit an IP filter rule

When you add an IP filter rule, you are prompted for the following values:

- An **IP filter rule name** that must be a unique, case-insensitive, alphanumeric string up to 128 characters long. Only the ASCII 7-bit alphanumeric characters plus `{'-', ':', '/', '\', '.', '+', '%', '_', '#', '*', '?', '!', '(', ')', ',', '=', '@', ';', ''}` are accepted.
- Select a **reject** or **accept** as the **action** for the IP filter rule.
- Provide a single IPv4 address or a block of IP addresses in CIDR notation. For example, in CIDR notation 192.168.100.0/22 represents the 1024 IPv4 addresses from 192.168.100.0 to 192.168.103.255.

The screenshot shows the 'IP Filter' configuration window. On the left, there's a toolbar with buttons for '+ Add', 'Undo', 'Delete', and 'Save'. The '+ Add' button is highlighted with a red box. The main area contains a table with columns: 'IP FILTER RULE NAME', 'ACTION', and 'IPV4 ADDRESS RANGE'. A message at the top of the table says 'No results'. On the right, there's a sidebar with fields for 'IP Filter Rule Name' (set to 'MaliciousIP'), 'Action' (set to 'Reject'), and 'IPv4 Address Range' (set to '6.6.6.6/6'). At the bottom right of the sidebar is a blue 'Create' button, which is also highlighted with a red box.

After you save the rule, you see an alert notifying you that the update is in progress.

This screenshot is similar to the previous one, showing the 'IP Filter' configuration window. The '+ Add' button is now disabled and highlighted with a red box. A notification bar at the top right says '*** Updating getStartedWithAnIoTHub 09:48' and has a close button. The main table and sidebar are identical to the previous screenshot, showing the rule 'MaliciousIP' with action 'Reject' and range '6.6.6.6/6'.

The **Add** option is disabled when you reach the maximum of 10 IP filter rules.

You can edit an existing rule by double-clicking the row that contains the rule.

NOTE

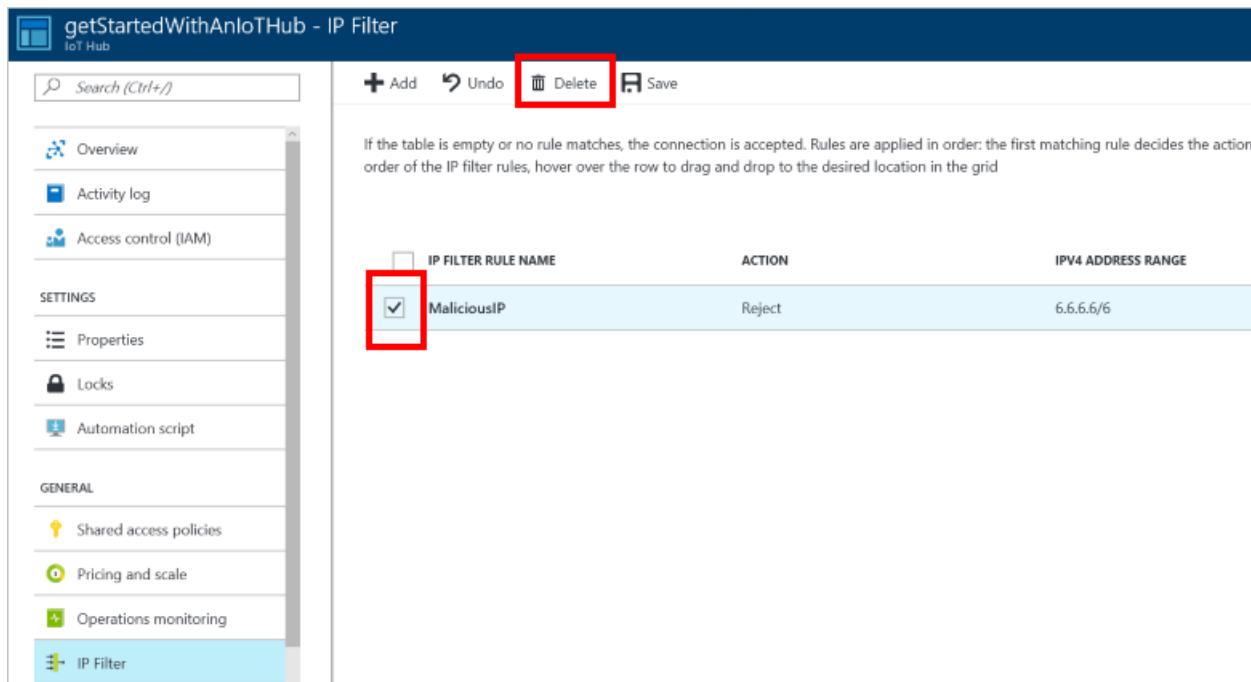
Rejecting IP addresses can prevent other Azure Services (such as Azure Stream Analytics, Azure Virtual Machines, or the Device Explorer in the portal) from interacting with the IoT hub.

WARNING

If you use Azure Stream Analytics (ASA) to read messages from an IoT hub with IP filtering enabled, use the Event Hub-compatible name and endpoint of your IoT Hub in the ASA connection string.

Delete an IP filter rule

To delete an IP filter rule, select one or more rules in the grid and click **Delete**.



IP FILTER RULE NAME	ACTION	IPV4 ADDRESS RANGE
MaliciousIP	Reject	6.6.6.6/6

Retrieve and update IP filters using Azure CLI

Your IoT Hub's IP filters can be retrieved and updated through [Azure CLI](#).

To retrieve current IP filters of your IoT Hub, run:

```
az resource show -n <iotHubName> -g <resourceGroupName> --resource-type Microsoft.Devices/IotHubs
```

This will return a JSON object where your existing IP filters are listed under the `properties.ipFilterRules` key:

```
{  
...  
  "properties": {  
    "ipFilterRules": [  
      {  
        "action": "Reject",  
        "filterName": "MaliciousIP",  
        "ipMask": "6.6.6.6/6"  
      },  
      {  
        "action": "Allow",  
        "filterName": "GoodIP",  
        "ipMask": "131.107.160.200"  
      },  
      ...  
    ],  
  },  
...  
}
```

To add a new IP filter for your IoT Hub, run:

```
az resource update -n <iothubName> -g <resourceGroupName> --resource-type Microsoft.Devices/IotHubs --add  
properties.ipFilterRules "{\"action\":\"Reject\", \"filterName\":\"MaliciousIP\", \"ipMask\":\"6.6.6.6/6\"}"
```

To remove an existing IP filter in your IoT Hub, run:

```
az resource update -n <iothubName> -g <resourceGroupName> --resource-type Microsoft.Devices/IotHubs --add  
properties.ipFilterRules <ipFilterIndexToRemove>
```

Note that `<ipFilterIndexToRemove>` must correspond to the ordering of IP filters in your IoT Hub's `properties.ipFilterRules`.

Retrieve and update IP filters using Azure PowerShell

NOTE

This article has been updated to use the new Azure PowerShell Az module. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For installation instructions, see [Install Azure PowerShell](#).

Your IoT Hub's IP filters can be retrieved and set through [Azure PowerShell](#).

```

# Get your IoT Hub resource using its name and its resource group name
$iothubResource = Get-AzResource -ResourceGroupName <resourceGroupName> -ResourceName <iotHubName> -
ExpandProperties

# Access existing IP filter rules
$iothubResource.Properties.ipFilterRules |% { Write-host $_ }

# Construct a new IP filter
$filter = @{'filterName'='MaliciousIP'; 'action'='Reject'; 'ipMask'='6.6.6.6/6'}

# Add your new IP filter rule
$iothubResource.Properties.ipFilterRules += $filter

# Remove an existing IP filter rule using its name, e.g., 'GoodIP'
$iothubResource.Properties.ipFilterRules = @($iothubResource.Properties.ipFilterRules | Where 'filterName' -ne
'GoodIP')

# Update your IoT Hub resource with your updated IP filters
$iothubResource | Set-AzResource -Force

```

Update IP filter rules using REST

You may also retrieve and modify your IoT Hub's IP filter using Azure resource Provider's REST endpoint. See `properties.ipFilterRules` in [createOrUpdate method](#).

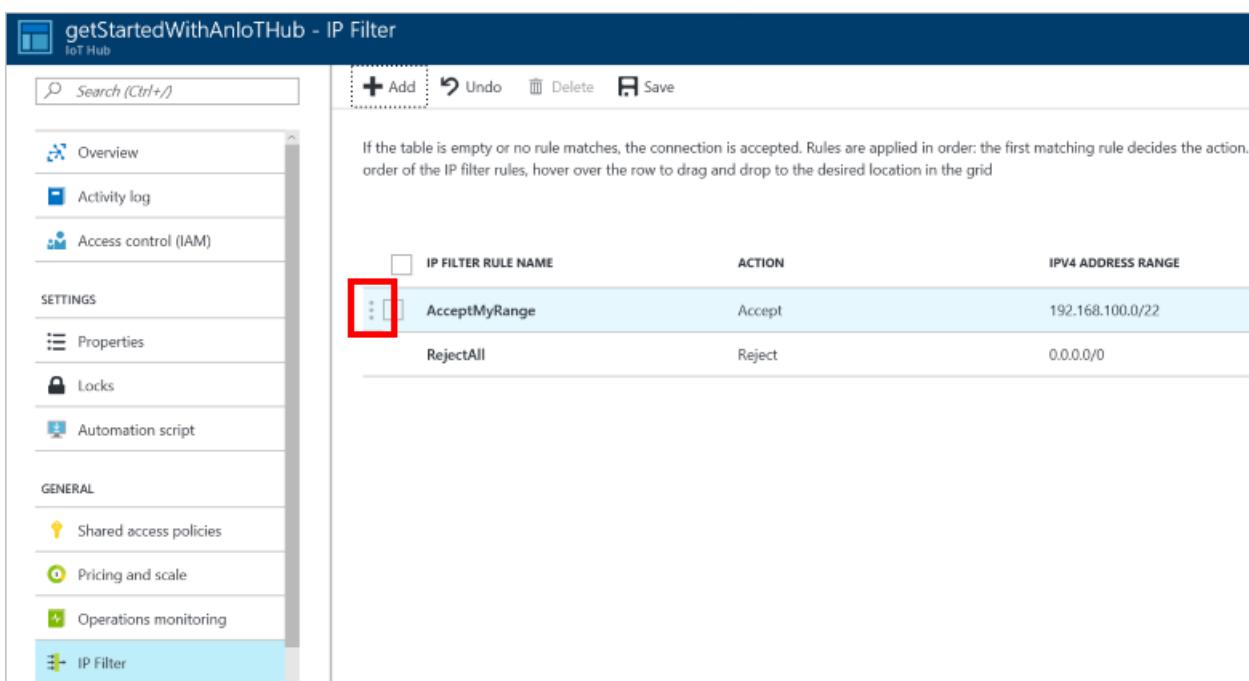
IP filter rule evaluation

IP filter rules are applied in order and the first rule that matches the IP address determines the accept or reject action.

For example, if you want to accept addresses in the range 192.168.100.0/22 and reject everything else, the first rule in the grid should accept the address range 192.168.100.0/22. The next rule should reject all addresses by using the range 0.0.0.0/0.

You can change the order of your IP filter rules in the grid by clicking the three vertical dots at the start of a row and using drag and drop.

To save your new IP filter rule order, click **Save**.



The screenshot shows the Azure portal interface for managing an IoT Hub named 'getStartedWithAnIoTHub'. The left sidebar contains navigation links for Overview, Activity log, Access control (IAM), Properties, Locks, Automation script, Shared access policies, Pricing and scale, Operations monitoring, and IP Filter. The 'IP Filter' link is currently selected and highlighted in blue. The main content area is titled 'IP Filter' and includes a search bar, an 'Add' button, and 'Undo', 'Delete', and 'Save' buttons. A note states: 'If the table is empty or no rule matches, the connection is accepted. Rules are applied in order: the first matching rule decides the action. Order of the IP filter rules, hover over the row to drag and drop to the desired location in the grid.' A table lists two IP filter rules:

IP FILTER RULE NAME	ACTION	IPV4 ADDRESS RANGE
AcceptMyRange	Accept	192.168.100.0/22
RejectAll	Reject	0.0.0.0/0

Next steps

To further explore the capabilities of IoT Hub, see:

- [Operations monitoring](#)
- [IoT Hub metrics](#)

Configure and monitor IoT devices at scale using the Azure portal

8/13/2018 • 7 minutes to read

Automatic device management in Azure IoT Hub automates many of the repetitive and complex tasks of managing large device fleets over the entirety of their lifecycles. With automatic device management, you can target a set of devices based on their properties, define a desired configuration, and let IoT Hub update devices whenever they come into scope. This is performed using an automatic device configuration, which will also allow you to summarize completion and compliance, handle merging and conflicts, and roll out configurations in a phased approach.

NOTE

The features described in this article are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

Automatic device configurations work by updating a set of device twins with desired properties and reporting a summary based on device twin reported properties. It introduces a new class and JSON document called a *Configuration* which has three parts:

- The **target condition** defines the scope of device twins to be updated. The target condition is specified as a query on device twin tags and/or reported properties.
- The **target content** defines the desired properties to be added or updated in the targeted device twins. The content includes a path to the section of desired properties to be changed.
- The **metrics** define the summary counts of various configuration states such as **Success**, **In Progress**, and **Error**. Custom metrics are specified as queries on device twin reported properties. System metrics are default metrics that measure twin update status, such as the number of device twins that are targeted and the number of twins that have been successfully updated.

Implement device twins to configure devices

Automatic device configurations require the use of device twins to synchronize state between the cloud and devices. Refer to [Understand and use device twins in IoT Hub](#) for guidance on using device twins.

Identify devices using tags

Before you can create a configuration, you must specify which devices you want to affect. Azure IoT Hub identifies devices using tags in the device twin. Each device can have multiple tags, and you can define them any way that makes sense for your solution. For example, if you manage devices in different locations, you may add the following tags to a device twin:

```
"tags": {  
    "location": {  
        "state": "Washington",  
        "city": "Tacoma"  
    }  
},
```

Create a configuration

1. In the [Azure portal](#), go to your IoT hub.
2. Select **IoT device configuration**.
3. Select **Add Configuration**.

There are five steps to create a configuration. The following sections walk through each one.

Name and Label

1. Give your configuration a unique name that is up to 128 lowercase letters. Avoid spaces and the following invalid characters: & ^ [] { } \ | " < > / .
2. Add labels to help track your configurations. Labels are **Name, Value** pairs that describe your configuration. For example, `HostPlatform, Linux` or `Version, 3.0.1`.
3. Select **Next** to move to the next step.

Specify Settings

This section specifies the target content to be set in targeted device twins. There are two inputs for each set of settings. The first is the device twin path, which is the path to the JSON section within the twin desired properties that will be set. The second is the JSON content to be inserted in that section. For example, set the Device Twin Path and Content to the following:

The screenshot shows the Azure portal interface with the title 'Create Configuration - 1'. The left sidebar shows 'Microsoft Azure' with various service icons. The main content area is titled 'Create Configuration' under 'Configuration Wizard'. A callout box points to the 'Device Twin Path' input field, which contains the value 'properties.desired.chiller-water'. Below it, the 'Content' field displays the following JSON code:

```
1 {
2   "temperature": "66",
3   "pressure": 28
4 }
```

At the bottom of the content area is a blue button labeled 'Add Device Twin Setting'.

You can also set individual settings by specifying the entire path in the Device Twin Path and the value in the Content with no brackets. For example, set the Device Twin Path to `properties.desired.chiller-water.temperature` and set the Content to `66`.

If two or more configurations target the same Device Twin Path, the Content from the highest priority configuration will apply (priority is defined in Step 4).

If you wish to remove a property, specify the property value to `null`.

You can add additional settings by selecting **Add Device Twin Setting**.

Specify Metrics (optional)

Metrics provide summary counts of the various states that a device may report back as a result of applying configuration content. For example, you may create a metric for pending settings changes, a metric for errors, and a metric for successful settings changes.

1. Enter a name for **Metric Name**.
2. Enter a query for **Metric Criteria**. The query is based on device twin reported properties. The metric represents the number of rows returned by the query.

For example:

```
SELECT deviceId FROM devices  
WHERE properties.reported.chillerWaterSettings.status='pending'
```

You can include a clause that the configuration was applied, for example:

```
/* Include the double brackets. */  
SELECT deviceId FROM devices  
WHERE configurations.[[yourconfigname]].status='Applied'
```

Target Devices

Use the tags property from your device twins to target the specific devices that should receive this configuration. You can also target devices by device twin reported properties.

Since multiple configurations may target the same device, you should give each configuration a priority number. If there's ever a conflict, the configuration with the highest priority wins.

1. Enter a positive integer for the configuration **Priority**. The highest numerical value is considered the highest priority. If two configurations have the same priority number, the one that was created most recently wins.
2. Enter a **Target condition** to determine which devices will be targeted with this configuration. The condition is based on device twin tags or device twin reported properties and should match the expression format. For example, `tags.environment='test'` or `properties.reported.chillerProperties.model='4000x'`. You can specify `*` to target all devices.
3. Select **Next** to move on to the final step.

Review Configuration

Review your configuration information, then select **Submit**.

Monitor a configuration

To view the details of a configuration and monitor the devices running it, use the following steps:

1. In the [Azure portal](#), go to your IoT hub.
2. Select **IoT device configuration**.
3. Inspect the configuration list. For each configuration, you can view the following details:
 - **ID** - the name of the configuration.
 - **Target condition** - the query used to define targeted devices.
 - **Priority** - the priority number assigned to the configuration.
 - **Creation time** - the timestamp from when the configuration was created. This timestamp is used to break ties when two configurations have the same priority.
 - **System metrics** - metrics that are calculated by IoT Hub and cannot be customized by developers. Targeted specifies the number of device twins that match the target condition. Applies

specified the number of device twins that have been modified by the configuration, which can include partial modifications in the event that a separate, higher priority configuration also made changes.

- **Custom metrics** - metrics that have been specified by the developer as queries against device twin reported properties. Up to five custom metrics can be defined per configuration.
4. Select the configuration that you want to monitor.
 5. Inspect the configuration details. You can use tabs to view specific details about the devices that received the configuration.
 - **Target Condition** - the devices that match the target condition.
 - **Metrics** - a list of system metrics and custom metrics. You can view a list of devices that are counted for each metric by selecting the metric in the drop-down and then selecting **View Devices**.
 - **Device Twin Settings** - the device twin settings that are set by the configuration.
 - **Configuration Labels** - key-value pairs used to describe a configuration. Labels have no impact on functionality.

Modify a configuration

When you modify a configuration, the changes immediately replicate to all targeted devices.

If you update the target condition, the following updates occur:

- If a device twin didn't meet the old target condition, but meets the new target condition and this configuration is the highest priority for that device twin, then this configuration is applied to the device twin.
- If a device twin no longer meets the target condition, the settings from the configuration will be removed and the device twin will be modified by the next highest priority configuration.
- If a device twin currently running this configuration no longer meets the target condition and doesn't meet the target condition of any other configurations, then the settings from the configuration will be removed and no other changes will be made on the twin.

To modify a configuration, use the following steps:

1. In the [Azure portal](#), go to your IoT hub.
2. Select **IoT device configuration**.
3. Select the configuration that you want to modify.
4. Make updates to the following fields:
 - Target condition
 - Labels
 - Priority
 - Metrics
5. Select **Save**.
6. Follow the steps in [Monitor a configuration](#) to watch the changes roll out.

Delete a configuration

When you delete a configuration, any device twins take on their next highest priority configuration. If device twins don't meet the target condition of any other configuration, then no other settings are applied.

1. In the [Azure portal](#) go to your IoT hub.
2. Select **IoT device configuration**.
3. Use the checkbox to select the configuration that you want to delete.
4. Select **Delete**.
5. A prompt will ask you to confirm.

Next steps

In this article, you learned how configure and monitor IoT devices at scale. Follow these links to learn more about managing Azure IoT Hub:

- [Manage your IoT Hub device identities in bulk](#)
- [IoT Hub metrics](#)
- [Operations monitoring](#)

To further explore the capabilities of IoT Hub, see:

- [IoT Hub developer guide](#)
- [Deploying AI to edge devices with Azure IoT Edge](#)

To explore using the IoT Hub Device Provisioning Service to enable zero-touch, just-in-time provisioning, see:

- [Azure IoT Hub Device Provisioning Service](#)

Configure and monitor IoT devices at scale using the Azure CLI

9/24/2018 • 7 minutes to read

Automatic device management in Azure IoT Hub automates many of the repetitive and complex tasks of managing large device fleets over the entirety of their lifecycles. With automatic device management, you can target a set of devices based on their properties, define a desired configuration, and let IoT Hub update devices whenever they come into scope. This is performed using an automatic device configuration, which will also allow you to summarize completion and compliance, handle merging and conflicts, and roll out configurations in a phased approach.

NOTE

The features described in this article are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

Automatic device configurations work by updating a set of device twins with desired properties and reporting a summary based on device twin reported properties. It introduces a new class and JSON document called a *Configuration* which has three parts:

- The **target condition** defines the scope of device twins to be updated. The target condition is specified as a query on device twin tags and/or reported properties.
- The **target content** defines the desired properties to be added or updated in the targeted device twins. The content includes a path to the section of desired properties to be changed.
- The **metrics** define the summary counts of various configuration states such as **Success**, **In Progress**, and **Error**. Custom metrics are specified as queries on device twin reported properties. System metrics are default metrics that measure twin update status, such as the number of device twins that are targeted and the number of twins that have been successfully updated.

CLI prerequisites

- An [IoT hub](#) in your Azure subscription.
- [Azure CLI](#) in your environment. At a minimum, your Azure CLI version must be 2.0.24 or above. Use `az --version` to validate. This version supports az extension commands and introduces the Knack command framework.
- The [IoT extension for Azure CLI](#).

Implement device twins to configure devices

Automatic device configurations require the use of device twins to synchronize state between the cloud and devices. Refer to [Understand and use device twins in IoT Hub](#) for guidance on using device twins.

Identify devices using tags

Before you can create a configuration, you must specify which devices you want to affect. Azure IoT Hub identifies devices using tags in the device twin. Each device can have multiple tags, and you can define them any way that makes sense for your solution. For example, if you manage devices in different locations, you may add the

following tags to a device twin:

```
"tags": {  
    "location": {  
        "state": "Washington",  
        "city": "Tacoma"  
    }  
},
```

Define the target content and metrics

The target content and metric queries are specified as JSON documents that describe the device twin desired properties to be set and reported properties to be measured. To create an automatic device configuration using Azure CLI, save the target content and metrics locally as .txt files. You will use the file paths in a later next section when you run the command to apply the configuration to your device.

Here's a basic target content sample:

```
{  
    "content": {  
        "deviceContent": {  
            "properties.desired.chillerWaterSettings": {  
                "temperature": 38,  
                "pressure": 78  
            }  
        }  
    }  
}
```

Here are examples of metric queries:

```
{  
    "queries": {  
        "Compliant": "select deviceId from devices where configurations.[[chillersettingswashington]].status = 'Applied' AND properties.reported.chillerWaterSettings.status='current'",  
        "Error": "select deviceId from devices where configurations.[[chillersettingswashington]].status = 'Applied' AND properties.reported.chillerWaterSettings.status='error'",  
        "Pending": "select deviceId from devices where configurations.[[chillersettingswashington]].status = 'Applied' AND properties.reported.chillerWaterSettings.status='pending'"  
    }  
}
```

Create a configuration

You configure target devices by creating a configuration that consists of the target content and metrics.

Use the following command to create a configuration:

```
az iot hub configuration create --config-id [configuration id] \  
    --labels [labels] --content [file path] --hub-name [hub name] \  
    --target-condition [target query] --priority [int] \  
    --metrics [metric queries]
```

- **--config-id** - The name of the configuration that will be created in the IoT hub. Give your configuration a unique name that is up to 128 lowercase letters. Avoid spaces and the following invalid characters:
`& ^ [] { } \ | " < > /`.
- **--labels** - Add labels to help track your configuration. Labels are Name, Value pairs that describe your

deployment. For example, `HostPlatform`, `Linux` or `Version, 3.0.1`

- **--content** - Inline JSON or file path to the target content to be set as twin desired properties.
- **--hub-name** - Name of the IoT hub in which the configuration will be created. The hub must be in the current subscription. Switch to the desired subscription with the command
`az account set -s [subscription name]`
- **--target-condition** - Enter a target condition to determine which devices will be targeted with this configuration. The condition is based on device twin tags or device twin desired properties and should match the expression format. For example, `tags.environment='test'` or `properties.desired.devicemode1='4000x'`.
- **--priority** - A positive integer. In the event that two or more configurations are targeted at the same device, the configuration with the highest numerical value for Priority will apply.
- **--metrics** - Filepath to the metric queries. Metrics provide summary counts of the various states that a device may report back as a result of applying configuration content. For example, you may create a metric for pending settings changes, a metric for errors, and a metric for successful settings changes.

Monitor a configuration

Use the following command to display the contents of a configuration:

```
az iot hub configuration show --config-id [configuration id] \
--hub-name [hub name]
```

- **--config-id** - The name of the configuration that exists in the IoT hub.
- **--hub-name** - Name of the IoT hub in which the configuration exists. The hub must be in the current subscription. Switch to the desired subscription with the command `az account set -s [subscription name]`

Inspect the configuration in the command window. The **metrics** property lists a count for each metric that is evaluated by each hub:

- **targetedCount** - A system metric that specifies the number of device twins in IoT Hub that match the targeting condition.
- **appliedCount** - A system metric specifies the number of devices that have had the target content applied.
- **Your custom metric** - Any metrics you have defined will be considered user metrics.

You can show a list of device IDs or objects for each of the metrics by using the following command:

```
az iot hub configuration show-metric --config-id [configuration id] \
--metric-id [metric id] --hub-name [hub name] --metric-type [type]
```

- **--config-id** - The name of the deployment that exists in the IoT hub.
- **--metric-id** - The name of the metric for which you want to see the list of device IDs, for example `appliedCount`.
- **--hub-name** - Name of the IoT hub in which the deployment exists. The hub must be in the current subscription. Switch to the desired subscription with the command `az account set -s [subscription name]`.
- **--metric-type** - Metric type can be `system` or `user`. System metrics are `targetedCount` and `appliedCount`. All other metrics are user metrics.

Modify a configuration

When you modify a configuration, the changes immediately replicate to all targeted devices.

If you update the target condition, the following updates occur:

- If a device twin didn't meet the old target condition, but meets the new target condition and this configuration is the highest priority for that device twin, then this configuration is applied to the device twin.
- If a device twin no longer meets the target condition, the settings from the configuration will be removed and the device twin will be modified by the next highest priority configuration.
- If a device twin currently running this configuration no longer meets the target condition and doesn't meet the target condition of any other configurations, then the settings from the configuration will be removed and no other changes will be made on the twin.

Use the following command to update a configuration:

```
az iot hub configuration update --config-id [configuration id] \  
    --hub-name [hub name] --set [property1.property2='value']
```

- **--config-id** - The name of the configuration that exists in the IoT hub.
- **--hub-name** - Name of the IoT hub in which the configuration exists. The hub must be in the current subscription. Switch to the desired subscription with the command `az account set -s [subscription name]`.
- **--set** - Update a property in the configuration. You can update the following properties:
 - targetCondition - for example `targetCondition=tags.location.state='Oregon'`
 - labels
 - priority

Delete a configuration

When you delete a configuration, any device twins take on their next highest priority configuration. If device twins don't meet the target condition of any other configuration, then no other settings are applied.

Use the following command to delete a configuration:

```
az iot hub configuration delete --config-id [configuration id] \  
    --hub-name [hub name]
```

- **--config-id** - The name of the configuration that exists in the IoT hub.
- **--hub-name** - Name of the IoT hub in which the configuration exists. The hub must be in the current subscription. Switch to the desired subscription with the command `az account set -s [subscription name]`.

Next steps

In this article, you learned how configure and monitor IoT devices at scale. Follow these links to learn more about managing Azure IoT Hub:

- [Manage your IoT Hub device identities in bulk](#)
- [IoT Hub metrics](#)
- [Operations monitoring](#)

To further explore the capabilities of IoT Hub, see:

- [IoT Hub developer guide](#)
- [Deploying AI to edge devices with Azure IoT Edge](#)

To explore using the IoT Hub Device Provisioning Service to enable zero-touch, just-in-time provisioning, see:

- [Azure IoT Hub Device Provisioning Service](#)

Manage your IoT Hub device identities in bulk

2/28/2019 • 11 minutes to read

Each IoT hub has an identity registry you can use to create per-device resources in the service. The identity registry also enables you to control access to the device-facing endpoints. This article describes how to import and export device identities in bulk to and from an identity registry.

NOTE

The features described in this article are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

Import and export operations take place in the context of *Jobs* that enable you to execute bulk service operations against an IoT hub.

The **RegistryManager** class includes the **ExportDevicesAsync** and **ImportDevicesAsync** methods that use the **Job** framework. These methods enable you to export, import, and synchronize the entirety of an IoT hub identity registry.

This topic discusses using the **RegistryManager** class and **Job** system to perform bulk imports and exports of devices to and from an IoT hub's identity registry. You can also use the Azure IoT Hub Device Provisioning Service to enable zero-touch, just-in-time provisioning to one or more IoT hubs without requiring human intervention. To learn more, see the [provisioning service documentation](#).

What are jobs?

Identity registry operations use the **Job** system when the operation:

- Has a potentially long execution time compared to standard run-time operations.
- Returns a large amount of data to the user.

Instead of a single API call waiting or blocking on the result of the operation, the operation asynchronously creates a **Job** for that IoT hub. The operation then immediately returns a **JobProperties** object.

The following C# code snippet shows how to create an export job:

```
// Call an export job on the IoT Hub to retrieve all devices
JobProperties exportJob = await
    registryManager.ExportDevicesAsync(containerSasUri, false);
```

NOTE

To use the **RegistryManager** class in your C# code, add the **Microsoft.Azure.Devices** NuGet package to your project. The **RegistryManager** class is in the **Microsoft.Azure.Devices** namespace.

You can use the **RegistryManager** class to query the state of the **Job** using the returned **JobProperties** metadata. To create an instance of the **RegistryManager** class, use the **CreateFromConnectionString** method.

```
RegistryManager registryManager =
    RegistryManager.CreateFromConnectionString("{your IoT Hub connection string}");
```

To find the connection string for your IoT hub, in the Azure portal:

- Navigate to your IoT hub.
- Select **Shared access policies**.
- Select a policy, taking into account the permissions you need.
- Copy the connectionstring from the panel on the right-hand side of the screen.

The following C# code snippet shows how to poll every five seconds to see if the job has finished executing:

```
// Wait until job is finished
while(true)
{
    exportJob = await registryManager.GetJobAsync(exportJob.JobId);
    if (exportJob.Status == JobStatus.Completed ||
        exportJob.Status == JobStatus.Failed ||
        exportJob.Status == JobStatus.Cancelled)
    {
        // Job has finished executing
        break;
    }

    await Task.Delay(TimeSpan.FromSeconds(5));
}
```

Export devices

Use the **ExportDevicesAsync** method to export the entirety of an IoT hub identity registry to an [Azure Storage](#) blob container using a [Shared Access Signature](#).

This method enables you to create reliable backups of your device information in a blob container that you control.

The **ExportDevicesAsync** method requires two parameters:

- A *string* that contains a URI of a blob container. This URI must contain a SAS token that grants write access to the container. The job creates a block blob in this container to store the serialized export device data. The SAS token must include these permissions:

```
SharedAccessBlobPermissions.Write | SharedAccessBlobPermissions.Read
| SharedAccessBlobPermissions.Delete
```

- A *boolean* that indicates if you want to exclude authentication keys from your export data. If **false**, authentication keys are included in export output. Otherwise, keys are exported as **null**.

The following C# code snippet shows how to initiate an export job that includes device authentication keys in the export data and then poll for completion:

```

// Call an export job on the IoT Hub to retrieve all devices
JobProperties exportJob =
    await registryManager.ExportDevicesAsync(containerSasUri, false);

// Wait until job is finished
while(true)
{
    exportJob = await registryManager.GetJobAsync(exportJob.JobId);
    if (exportJob.Status == JobStatus.Completed ||
        exportJob.Status == JobStatus.Failed ||
        exportJob.Status == JobStatus.Cancelled)
    {
        // Job has finished executing
        break;
    }

    await Task.Delay(TimeSpan.FromSeconds(5));
}

```

The job stores its output in the provided blob container as a block blob with the name **devices.txt**. The output data consists of JSON serialized device data, with one device per line.

The following example shows the output data:

```

{"id":"Device1","eTag":"MA==","status":"enabled","authentication":{"symmetricKey":
{"primaryKey":"abc=","secondaryKey":"def="}}}
{"id":"Device2","eTag":"MA==","status":"enabled","authentication":{"symmetricKey":
{"primaryKey":"abc=","secondaryKey":"def="}}}
 {"id":"Device3","eTag":"MA==","status":"disabled","authentication":{"symmetricKey":
 {"primaryKey":"abc=","secondaryKey":"def="}}}
 {"id":"Device4","eTag":"MA==","status":"disabled","authentication":{"symmetricKey":
 {"primaryKey":"abc=","secondaryKey":"def="}}}
 {"id":"Device5","eTag":"MA==","status":"enabled","authentication":{"symmetricKey":
 {"primaryKey":"abc=","secondaryKey":"def="}}}

```

If a device has twin data, then the twin data is also exported together with the device data. The following example shows this format. All data from the "twinETag" line until the end is twin data.

```
{
    "id": "export-6d84f075-0",
    "eTag": "MQ==",
    "status": "enabled",
    "statusReason": "firstUpdate",
    "authentication": null,
    "twinETag": "AAAAAAAAAAI=",
    "tags": {
        "Location": "LivingRoom"
    },
    "properties": {
        "desired": {
            "Thermostat": {
                "Temperature": 75.1,
                "Unit": "F"
            },
            "$metadata": {
                "$lastUpdated": "2017-03-09T18:30:52.3167248Z",
                "$lastUpdatedVersion": 2,
                "Thermostat": {
                    "$lastUpdated": "2017-03-09T18:30:52.3167248Z",
                    "$lastUpdatedVersion": 2,
                    "Temperature": {
                        "$lastUpdated": "2017-03-09T18:30:52.3167248Z",
                        "$lastUpdatedVersion": 2
                    },
                    "Unit": {
                        "$lastUpdated": "2017-03-09T18:30:52.3167248Z",
                        "$lastUpdatedVersion": 2
                    }
                }
            },
            "$version": 2
        },
        "reported": {
            "$metadata": {
                "$lastUpdated": "2017-03-09T18:30:51.1309437Z"
            },
            "$version": 1
        }
    }
}
```

If you need access to this data in code, you can easily deserialize this data using the **ExportImportDevice** class. The following C# code snippet shows how to read device information that was previously exported to a block blob:

```
var exportedDevices = new List<ExportImportDevice>();

using (var streamReader = new StreamReader(await
blob.OpenReadAsync(AccessCondition.GenerateIfExistsCondition(), null, null), Encoding.UTF8))
{
    while (streamReader.Peek() != -1)
    {
        string line = await streamReader.ReadLineAsync();
        var device = JsonConvert.DeserializeObject<ExportImportDevice>(line);
        exportedDevices.Add(device);
    }
}
```

Import devices

The **ImportDevicesAsync** method in the **RegistryManager** class enables you to perform bulk import and

synchronization operations in an IoT hub identity registry. Like the **ExportDevicesAsync** method, the **ImportDevicesAsync** method uses the **Job** framework.

Take care using the **ImportDevicesAsync** method because in addition to provisioning new devices in your identity registry, it can also update and delete existing devices.

WARNING

An import operation cannot be undone. Always back up your existing data using the **ExportDevicesAsync** method to another blob container before you make bulk changes to your identity registry.

The **ImportDevicesAsync** method takes two parameters:

- A *string* that contains a URI of an [Azure Storage](#) blob container to use as *input* to the job. This URI must contain a SAS token that grants read access to the container. This container must contain a blob with the name **devices.txt** that contains the serialized device data to import into your identity registry. The import data must contain device information in the same JSON format that the **ExportImportDevice** job uses when it creates a **devices.txt** blob. The SAS token must include these permissions:

```
SharedAccessBlobPermissions.Read
```

- A *string* that contains a URI of an [Azure Storage](#) blob container to use as *output* from the job. The job creates a block blob in this container to store any error information from the completed import **Job**. The SAS token must include these permissions:

```
SharedAccessBlobPermissions.Write | SharedAccessBlobPermissions.Read  
| SharedAccessBlobPermissions.Delete
```

NOTE

The two parameters can point to the same blob container. The separate parameters simply enable more control over your data as the output container requires additional permissions.

The following C# code snippet shows how to initiate an import job:

```
JobProperties importJob =  
    await registryManager.ImportDevicesAsync(containerSasUri, containerSasUri);
```

This method can also be used to import the data for the device twin. The format for the data input is the same as the format shown in the **ExportDevicesAsync** section. In this way, you can reimport the exported data. The **\$metadata** is optional.

Import behavior

You can use the **ImportDevicesAsync** method to perform the following bulk operations in your identity registry:

- Bulk registration of new devices
- Bulk deletions of existing devices
- Bulk status changes (enable or disable devices)
- Bulk assignment of new device authentication keys
- Bulk auto-regeneration of device authentication keys

- Bulk update of twin data

You can perform any combination of the preceding operations within a single **ImportDevicesAsync** call. For example, you can register new devices and delete or update existing devices at the same time. When used along with the **ExportDevicesAsync** method, you can completely migrate all your devices from one IoT hub to another.

If the import file includes twin metadata, then this metadata overwrites the existing twin metadata. If the import file does not include twin metadata, then only the `lastUpdateTime` metadata is updated using the current time.

Use the optional **importMode** property in the import serialization data for each device to control the import process per-device. The **importMode** property has the following options:

IMPORTMODE	DESCRIPTION
createOrUpdate	<p>If a device does not exist with the specified id, it is newly registered.</p> <p>If the device already exists, existing information is overwritten with the provided input data without regard to the ETag value.</p> <p>The user can optionally specify twin data along with the device data. The twin's etag, if specified, is processed independently from the device's etag. If there is a mismatch with the existing twin's etag, an error is written to the log file.</p>
create	<p>If a device does not exist with the specified id, it is newly registered.</p> <p>If the device already exists, an error is written to the log file.</p> <p>The user can optionally specify twin data along with the device data. The twin's etag, if specified, is processed independently from the device's etag. If there is a mismatch with the existing twin's etag, an error is written to the log file.</p>
update	<p>If a device already exists with the specified id, existing information is overwritten with the provided input data without regard to the ETag value.</p> <p>If the device does not exist, an error is written to the log file.</p>
updateIfMatchETag	<p>If a device already exists with the specified id, existing information is overwritten with the provided input data only if there is an ETag match.</p> <p>If the device does not exist, an error is written to the log file.</p> <p>If there is an ETag mismatch, an error is written to the log file.</p>
createOrUpdateIfMatchETag	<p>If a device does not exist with the specified id, it is newly registered.</p> <p>If the device already exists, existing information is overwritten with the provided input data only if there is an ETag match.</p> <p>If there is an ETag mismatch, an error is written to the log file.</p> <p>The user can optionally specify twin data along with the device data. The twin's etag, if specified, is processed independently from the device's etag. If there is a mismatch with the existing twin's etag, an error is written to the log file.</p>
delete	<p>If a device already exists with the specified id, it is deleted without regard to the ETag value.</p> <p>If the device does not exist, an error is written to the log file.</p>

IMPORTMODE	DESCRIPTION
deleteIfMatchETag	If a device already exists with the specified id , it is deleted only if there is an ETag match. If the device does not exist, an error is written to the log file. If there is an ETag mismatch, an error is written to the log file.

NOTE

If the serialization data does not explicitly define an **importMode** flag for a device, it defaults to **createOrUpdate** during the import operation.

Import devices example – bulk device provisioning

The following C# code sample illustrates how to generate multiple device identities that:

- Include authentication keys.
- Write that device information to a block blob.
- Import the devices into the identity registry.

```

// Provision 1,000 more devices
var serializedDevices = new List<string>();

for (var i = 0; i < 1000; i++)
{
    // Create a new ExportImportDevice
    // CryptoKeyGenerator is in the Microsoft.Azure.Devices.Common namespace
    var deviceToAdd = new ExportImportDevice()
    {
        Id = Guid.NewGuid().ToString(),
        Status = DeviceStatus.Enabled,
        Authentication = new AuthenticationMechanism()
        {
            SymmetricKey = new SymmetricKey()
            {
                PrimaryKey = CryptoKeyGenerator.GenerateKey(32),
                SecondaryKey = CryptoKeyGenerator.GenerateKey(32)
            }
        },
        ImportMode = ImportMode.Create
    };

    // Add device to the list
    serializedDevices.Add(JsonConvert.SerializeObject(deviceToAdd));
}

// Write the list to the blob
var sb = new StringBuilder();
serializedDevices.ForEach(serializedDevice => sb.AppendLine(serializedDevice));
await blob.DeleteIfExistsAsync();

using (CloudBlobStream stream = await blob.OpenWriteAsync())
{
    byte[] bytes = Encoding.UTF8.GetBytes(sb.ToString());
    for (var i = 0; i < bytes.Length; i += 500)
    {
        int length = Math.Min(bytes.Length - i, 500);
        await stream.WriteAsync(bytes, i, length);
    }
}

// Call import using the blob to add new devices
// Log information related to the job is written to the same container
// This normally takes 1 minute per 100 devices
JobProperties importJob =
    await registryManager.ImportDevicesAsync(containerSasUri, containerSasUri);

// Wait until job is finished
while(true)
{
    importJob = await registryManager.GetJobAsync(importJob.JobId);
    if (importJob.Status == JobStatus.Completed ||
        importJob.Status == JobStatus.Failed ||
        importJob.Status == JobStatus.Cancelled)
    {
        // Job has finished executing
        break;
    }

    await Task.Delay(TimeSpan.FromSeconds(5));
}

```

Import devices example – bulk deletion

The following code sample shows you how to delete the devices you added using the previous code sample:

```

// Step 1: Update each device's ImportMode to be Delete
sb = new StringBuilder();
serializedDevices.ForEach(serializedDevice =>
{
    // Deserialize back to an ExportImportDevice
    var device = JsonConvert.DeserializeObject<ExportImportDevice>(serializedDevice);

    // Update property
    device.ImportMode = ImportMode.Delete;

    // Re-serialize
    sb.AppendLine(JsonConvert.SerializeObject(device));
});

// Step 2: Write the new import data back to the block blob
await blob.DeleteIfExistsAsync();
using (CloudBlobStream stream = await blob.OpenWriteAsync())
{
    byte[] bytes = Encoding.UTF8.GetBytes(sb.ToString());
    for (var i = 0; i < bytes.Length; i += 500)
    {
        int length = Math.Min(bytes.Length - i, 500);
        await stream.WriteAsync(bytes, i, length);
    }
}

// Step 3: Call import using the same blob to delete all devices
importJob = await registryManager.ImportDevicesAsync(containerSasUri, containerSasUri);

// Wait until job is finished
while(true)
{
    importJob = await registryManager.GetJobAsync(importJob.JobId);
    if (importJob.Status == JobStatus.Completed ||
        importJob.Status == JobStatus.Failed ||
        importJob.Status == JobStatus.Cancelled)
    {
        // Job has finished executing
        break;
    }

    await Task.Delay(TimeSpan.FromSeconds(5));
}

```

Get the container SAS URI

The following code sample shows you how to generate a [SAS URI](#) with read, write, and delete permissions for a blob container:

```

static string GetContainerSasUri(CloudBlobContainer container)
{
    // Set the expiry time and permissions for the container.
    // In this case no start time is specified, so the
    // shared access signature becomes valid immediately.
    var sasConstraints = new SharedAccessBlobPolicy();
    sasConstraints.SharedAccessExpiryTime = DateTime.UtcNow.AddHours(24);
    sasConstraints.Permissions =
        SharedAccessBlobPermissions.Write |
        SharedAccessBlobPermissions.Read |
        SharedAccessBlobPermissions.Delete;

    // Generate the shared access signature on the container,
    // setting the constraints directly on the signature.
    string sasContainerToken = container.GetSharedAccessSignature(sasConstraints);

    // Return the URI string for the container,
    // including the SAS token.
    return container.Uri + sasContainerToken;
}

```

Next steps

In this article, you learned how to perform bulk operations against the identity registry in an IoT hub. Follow these links to learn more about managing Azure IoT Hub:

- [IoT Hub metrics](#)
- [Operations monitoring](#)

To further explore the capabilities of IoT Hub, see:

- [IoT Hub developer guide](#)
- [Deploying AI to edge devices with Azure IoT Edge](#)

To explore using the IoT Hub Device Provisioning Service to enable zero-touch, just-in-time provisioning, see:

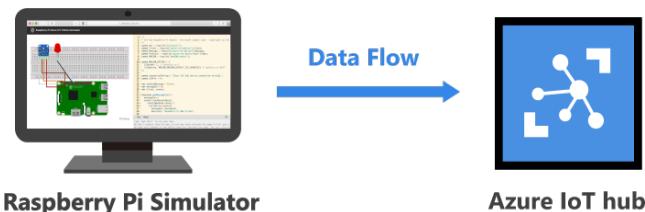
- [Azure IoT Hub Device Provisioning Service](#)

Connect Raspberry Pi online simulator to Azure IoT Hub (Node.js)

12/14/2018 • 6 minutes to read

In this tutorial, you begin by learning the basics of working with Raspberry Pi online simulator. You then learn how to seamlessly connect the Pi simulator to the cloud by using [Azure IoT Hub](#).

If you have physical devices, visit [Connect Raspberry Pi to Azure IoT Hub](#) to get started.



START RASPBERRY PI SIMULATOR

What you do

- Learn the basics of Raspberry Pi online simulator.
- Create an IoT hub.
- Register a device for Pi in your IoT hub.
- Run a sample application on Pi to send simulated sensor data to your IoT hub.

Connect simulated Raspberry Pi to an IoT hub that you create. Then you run a sample application with the simulator to generate sensor data. Finally, you send the sensor data to your IoT hub.

What you learn

- How to create an Azure IoT hub and get your new device connection string. If you don't have an Azure account, [create a free Azure trial account](#) in just a few minutes.
- How to work with Raspberry Pi online simulator.
- How to send sensor data to your IoT hub.

Overview of Raspberry Pi web simulator

Click the button to launch Raspberry Pi online simulator.



There are three areas in the web simulator.

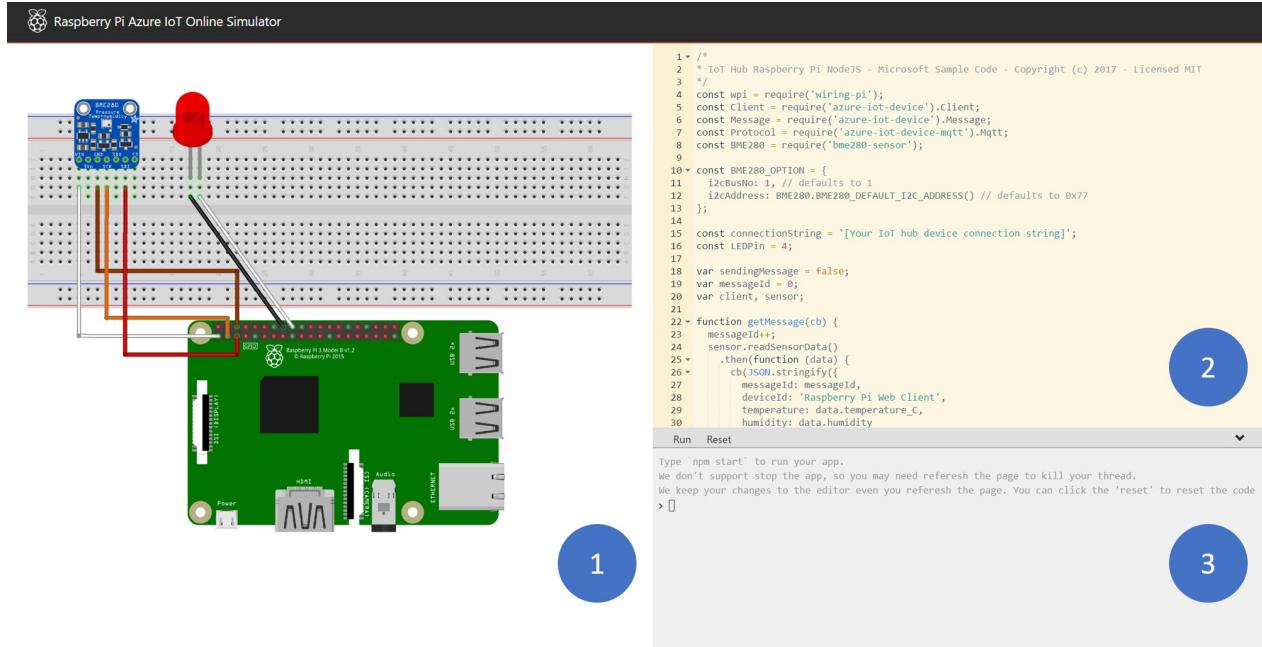
1. Assembly area - The default circuit is that a Pi connects with a BME280 sensor and an LED. The area is locked in preview version so currently you cannot do customization.
2. Coding area - An online code editor for you to code with Raspberry Pi. The default sample application helps to collect sensor data from BME280 sensor and sends to your Azure IoT Hub. The application is fully compatible with real Pi devices.

3. Integrated console window - It shows the output of your code. At the top of this window, there are three buttons.

- **Run** - Run the application in the coding area.
- **Reset** - Reset the coding area to the default sample application.
- **Fold/Expand** - On the right side there is a button for you to fold/expand the console window.

NOTE

The Raspberry Pi web simulator is now available in preview version. We'd like to hear your voice in the [Gitter Chatroom](#). The source code is public on [GitHub](#).



Create an IoT hub

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose +**Create a resource**, then choose **Internet of Things**.
3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Review + create

Create an IoT Hub to help you connect, monitor, and manage billions of your IoT assets. [Learn More](#)

PROJECT DETAILS

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

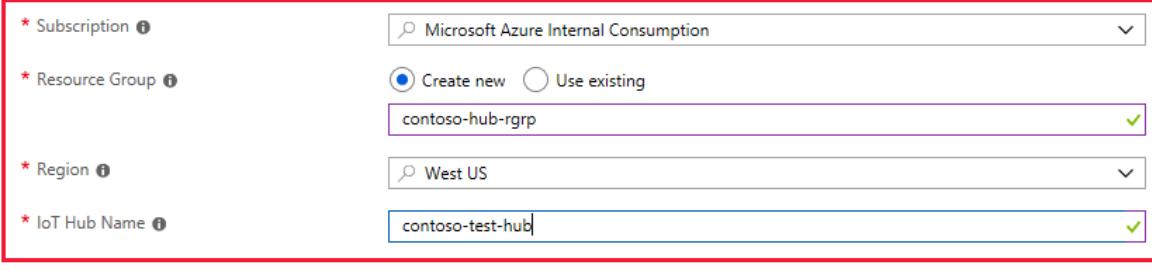
* Subscription Create new Use existing
contoso-hub-rgrp ✓

* Resource Group Create new Use existing
contoso-hub-rgrp ✓

* Region Create new Use existing
contoso-test-hub ✓

* IoT Hub Name Create new Use existing
contoso-test-hub ✓

[Review + create](#) [Next: Size and scale »](#) [Automation options](#)



Fill in the fields.

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics **Size and scale** Review + create

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS

* Pricing and scale tier [?](#) S1: Standard tier [?](#) [Learn how to choose the right IoT Hub tier for your solution](#)

Number of S1 IoT Hub units [?](#) 1 [1](#) This determines your IoT Hub scale capability and can be changed as your need increases.

Device-to-cloud-messages [?](#) Enabled

Message routing [?](#) Enabled

Cloud-to-device commands [?](#) Enabled

IoT Edge [?](#) Enabled

Device management [?](#) Enabled

Advanced Settings

Device-to-cloud partitions [?](#) 4 [4](#)

[Review + create](#) [« Previous: Basics](#) [Automation options](#)

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: The number of messages allowed per unit per day depends on your hub's pricing tier. For example, if you want the IoT hub to support ingress of 700,000 messages, you choose two S1 tier units.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

Advanced / Device-to-cloud partitions: This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most IoT hubs only need four partitions.

5. Click **Review + create** to review your choices. You see something similar to this screen.

Basics **Size and scale** **Review + create**

BASICS

Subscription i	Microsoft Azure Internal Consumption
Resource Group i	contoso-hub-rgrp
Region i	West US
IoT Hub Name i	contoso-test-hub

SIZE AND SCALE

Pricing and scale tier i	S1
Number of S1 IoT Hub units i	1
Messages per day i	400,000
Cost per month	25.00 USD

Create « Previous: Size and scale Automation options

- Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Retrieve connection string for IoT hub

After your hub has been created, retrieve the connection string for the hub. This is used to connect devices and applications to your hub.

- Click on your hub to see the IoT Hub pane with Settings, and so on. Click **Shared access policies**.
- In **Shared access policies**, select the **iothubowner** policy.
- Under **Shared access keys**, copy the **Connection string -- primary key** to be used later.

ContosoHub - Shared access policies

iothubowner

Access policy name: iothubowner

Permissions:

- Registry read [i](#)
- Registry write [i](#)
- Service connect [i](#)
- Device connect [i](#)

Shared access keys:

Primary key:	<code>HostName=ContosoHub.azure-devices.net;SharedAccessKey...</code>	
Secondary key:	<code>HostName=ContosoHub.azure-devices.net;SharedAccessKey...</code>	

For more information, see [Access control](#) in the "IoT Hub developer guide."

Register a new device in the IoT hub

In this section, you create a device identity in the identity registry in your IoT hub. A device cannot connect to IoT hub unless it has an entry in the identity registry. For more information, see the "Identity registry" section of the [IoT Hub developer guide](#)

1. In your IoT hub navigation menu, open **IoT Devices**, then click **Add** to register a new device in your IoT hub.

The screenshot shows the 'ContosoHub - IoT devices' blade in the Azure portal. On the left, there's a navigation menu with items like 'Events', 'Shared access policies', 'Pricing and scale', etc., and a red box highlights the 'IoT devices' item under 'EXPLORERS'. At the top, there's a search bar, a 'Refresh' button, and a 'Delete' button. A large red box highlights the '+ Add' button. Below the buttons, there's a message: 'You can use this tool to view, create, update, and delete devices on your IoT Hub.' There's also a 'Query' section with a query editor containing: 'Query ⓘ SELECT * FROM devices WHERE optional (e.g. tags.location='US')' and an 'Execute' button. At the bottom, there's a table header with columns: DEVICE ID, STATUS, LAST ACTIVITY, LAST STATUS..., AUTHENTICATI..., CLOUD TO DEV... and a row 'No results'.

2. Provide a name for your new device, such as **myDeviceId**, and click **Save**. This action creates a new device identity for your IoT hub.



Create a device

□ X



Learn more about creating devices



* Device ID i

myDeviceId



Authentication type i

Symmetric key X.509 Self-Signed X.509 CA Signed

* Primary key i

Enter your primary key

* Secondary key i

Enter your secondary key

Auto-generate keys i



Connect this device to an IoT hub i

Enable Disable

Parent device (Preview) i

No parent device

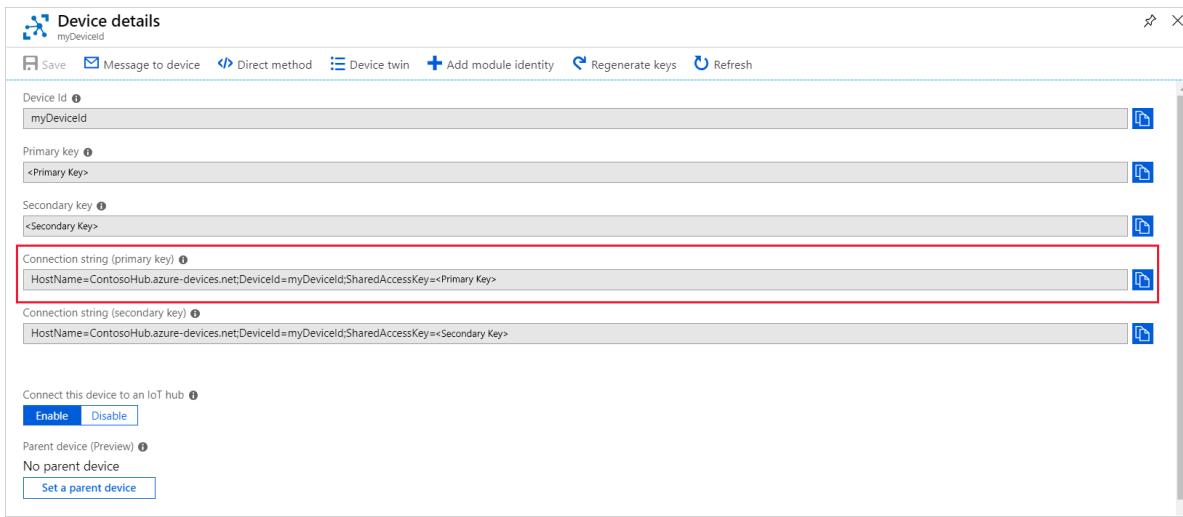
[Set a parent device](#)

[Save](#)

IMPORTANT

The device ID may be visible in the logs collected for customer support and troubleshooting, so make sure to avoid any sensitive information while naming it.

3. After the device is created, open the device from the list in the **IoT devices** pane. Copy the **Connection string---primary key** to use later.



NOTE

The IoT Hub identity registry only stores device identities to enable secure access to the IoT hub. It stores device IDs and keys to use as security credentials, and an enabled/disabled flag that you can use to disable access for an individual device. If your application needs to store other device-specific metadata, it should use an application-specific store. For more information, see [IoT Hub developer guide](#).

Run a sample application on Pi web simulator

1. In coding area, make sure you are working on the default sample application. Replace the placeholder in Line 15 with the Azure IoT hub device connection string.

```
14
15 const connectionString = '[Your IoT hub device connection string]';
16 const LEDPin = 4;
17
```

2. Click **Run** or type `npm start` to run the application.

You should see the following output that shows the sensor data and the messages that are sent to your IoT hub

```
Running Reset

Type `npm start` to run your app.
We don't support stop the app, so you may need refresh the page to kill your thread.
We keep your changes to the editor even you refresh the page. You can click the 'reset' to reset the code
>
Sending message: {"messageId":1,"deviceId":"Raspberry Pi Web Client","temperature":25.584710773750324,"humidity"
>
Message sent to Azure IoT Hub
```

Next steps

You've run a sample application to collect sensor data and send it to your IoT hub.

To continue to get started with Azure IoT Hub and to explore all extended IoT scenarios, see the following:

- [Manage cloud device messaging with Azure IoT Hub Toolkit extension for Visual Studio Code](#)
- [Manage devices with Azure IoT Hub Toolkit extension for Visual Studio Code](#)
- [Set up message routing](#)
- [Use Power BI to visualize real-time sensor data from your IoT hub](#)
- [Use the Web Apps feature of Azure App Service to visualize real-time sensor data from your IoT hub](#)

- Forecast weather by using the sensor data from your IoT hub in Azure Machine Learning
- Use Logic Apps for remote monitoring and notifications

Azure IoT Hub get started with physical devices tutorials

10/15/2018 • 2 minutes to read

These tutorials introduce you to Azure IoT Hub and the device SDKs. The tutorials cover common IoT scenarios to demonstrate the capabilities of IoT Hub. The tutorials also illustrate how to combine IoT Hub with other Azure services and tools to build more powerful IoT solutions. The tutorials listed in the following table show you how to create physical IoT devices.

IOT DEVICE	PROGRAMMING LANGUAGE
Raspberry Pi	Node.js, C
IoT DevKit	Arduino in VSCode
Adafruit Feather HUZZAH ESP8266	Arduino

Extended IoT scenarios

Use other Azure services and tools. When you have connected your device to IoT Hub, you can explore additional scenarios that use other Azure tools and services:

SCENARIO	AZURE SERVICE OR TOOL
Manage IoT Hub messages	VS Code Azure IoT Hub Toolkit extension
Manage your IoT device	Azure CLI and the IoT extension
Manage your IoT device	VS Code Azure IoT Hub Toolkit extension
Save IoT Hub messages to Azure storage	Azure table storage
Visualize sensor data	Microsoft Power BI
Visualize sensor data	Azure Web Apps
Forecast weather with sensor data	Azure Machine Learning
Automatic anomaly detection and reaction	Azure Logic Apps

Next steps

When you have completed these tutorials, you can further explore the capabilities of IoT Hub in the [Developer guide](#).

Connect Raspberry Pi to Azure IoT Hub (Node.js)

3/6/2019 • 10 minutes to read

In this tutorial, you begin by learning the basics of working with Raspberry Pi that's running Raspbian. You then learn how to seamlessly connect your devices to the cloud by using [Azure IoT Hub](#). For Windows 10 IoT Core samples, go to the [Windows Dev Center](#).

Don't have a kit yet? Try [Raspberry Pi online simulator](#). Or buy a new kit [here](#).

What you do

- Create an IoT hub.
- Register a device for Pi in your IoT hub.
- Set up Raspberry Pi.
- Run a sample application on Pi to send sensor data to your IoT hub.

What you learn

- How to create an Azure IoT hub and get your new device connection string.
- How to connect Pi with a BME280 sensor.
- How to collect sensor data by running a sample application on Pi.
- How to send sensor data to your IoT hub.

What you need



- A Raspberry Pi 2 or Raspberry Pi 3 board.
- An Azure subscription. If you don't have an Azure subscription, create a [free account](#) before you begin.
- A monitor, a USB keyboard, and mouse that connects to Pi.
- A Mac or PC that is running Windows or Linux.
- An internet connection.
- A 16 GB or above microSD card.
- A USB-SD adapter or microSD card to burn the operating system image onto the microSD card.
- A 5-volt 2-amp power supply with the 6-foot micro USB cable.

The following items are optional:

- An assembled Adafruit BME280 temperature, pressure, and humidity sensor.
- A breadboard.
- 6 F/M jumper wires.
- A diffused 10-mm LED.

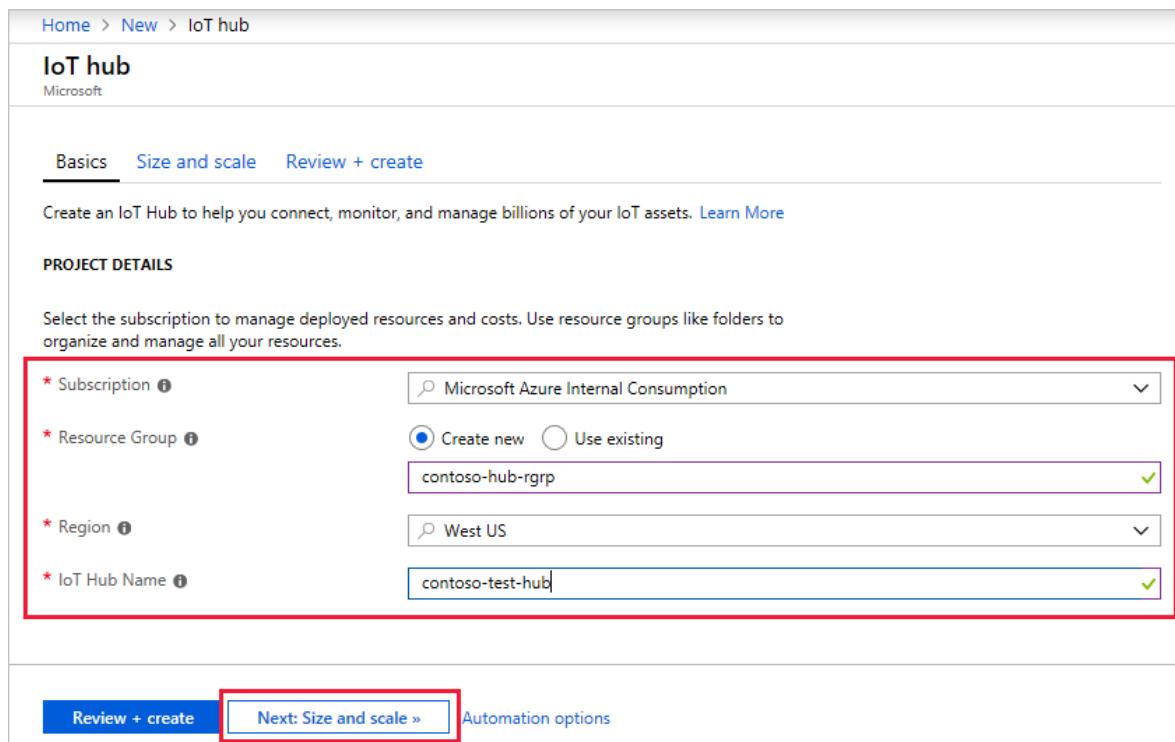
NOTE

If you don't have the optional items, you can use simulated sensor data.

Create an IoT hub

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose +**Create a resource**, then choose **Internet of Things**.
3. Click **Iot Hub** from the list on the right. You see the first screen for creating an IoT hub.



Fill in the fields.

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

The screenshot shows the 'Size and scale' configuration page for an IoT hub. At the top, there are tabs for 'Basics', 'Size and scale' (which is selected), and 'Review + create'. A note below the tabs states: 'Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send.' A link 'Learn more' is provided. The 'SCALE TIER AND UNITS' section includes a dropdown for 'Pricing and scale tier' set to 'S1: Standard tier' with a note 'Learn how to choose the right IoT Hub tier for your solution'. Below it, a slider for 'Number of S1 IoT Hub units' is set to 1, with a note: 'This determines your IoT Hub scale capability and can be changed as your need increases.' A large box lists enabled features: 'Device-to-cloud-messages' (Enabled), 'Message routing' (Enabled), 'Cloud-to-device commands' (Enabled), 'IoT Edge' (Enabled), and 'Device management' (Enabled). At the bottom, there's an 'Advanced Settings' section with a slider for 'Device-to-cloud partitions' set to 4. Navigation buttons at the very bottom include 'Review + create' (highlighted in blue), '< Previous: Basics', and 'Automation options'.

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: The number of messages allowed per unit per day depends on your hub's pricing tier. For example, if you want the IoT hub to support ingress of 700,000 messages, you choose two S1 tier units.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

Advanced / Device-to-cloud partitions: This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most IoT hubs only need four partitions.

5. Click **Review + create** to review your choices. You see something similar to this screen.

The screenshot shows the 'Review + create' step of the IoT hub creation wizard. It displays basic information like subscription, resource group, region, and IoT Hub name, as well as size and scale details. At the bottom, the 'Create' button is highlighted with a red box.

6. Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Retrieve connection string for IoT hub

After your hub has been created, retrieve the connection string for the hub. This is used to connect devices and applications to your hub.

1. Click on your hub to see the IoT Hub pane with Settings, and so on. Click **Shared access policies**.
2. In **Shared access policies**, select the **iothubowner** policy.
3. Under **Shared access keys**, copy the **Connection string -- primary key** to be used later.

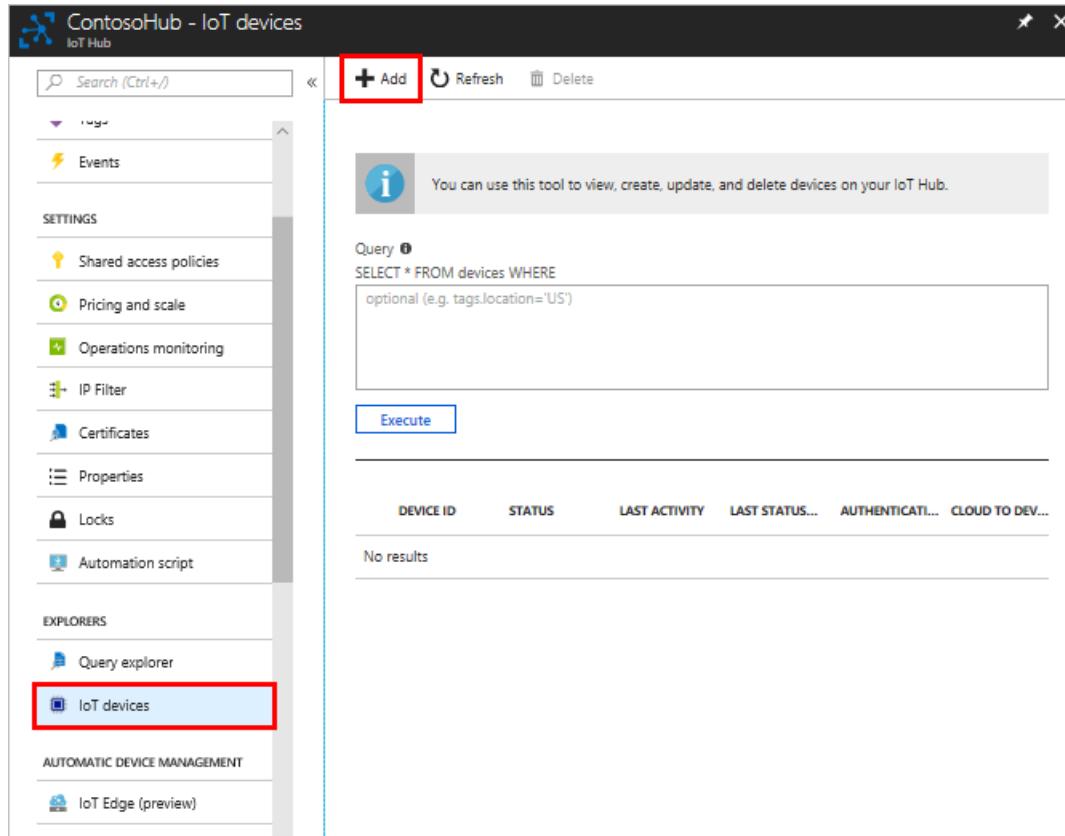
The screenshot shows the 'Shared access policies' blade for the ContosoHub IoT Hub. It highlights the 'iothubowner' policy settings, specifically the 'Primary key' connection string which is highlighted with a red box.

For more information, see [Access control](#) in the "IoT Hub developer guide."

Register a new device in the IoT hub

In this section, you create a device identity in the identity registry in your IoT hub. A device cannot connect to IoT hub unless it has an entry in the identity registry. For more information, see the "Identity registry" section of the [IoT Hub developer guide](#)

1. In your IoT hub navigation menu, open **IoT Devices**, then click **Add** to register a new device in your IoT hub.



The screenshot shows the 'ContosoHub - IoT devices' blade in the Azure portal. On the left, there's a navigation menu with items like 'Events', 'SETTINGS' (with 'Shared access policies', 'Pricing and scale', etc.), 'EXPLORERS' (with 'Query explorer'), and 'AUTOMATIC DEVICE MANAGEMENT' (with 'IoT Edge (preview)'). The 'IoT devices' item is highlighted with a red box. At the top right, there are buttons for '+ Add', 'Refresh', and 'Delete'. Below them is a query editor with a placeholder 'optional (e.g. tags.location='US')' and an 'Execute' button. A message says: 'You can use this tool to view, create, update, and delete devices on your IoT Hub.' A table at the bottom shows columns for 'DEVICE ID', 'STATUS', 'LAST ACTIVITY', 'LAST STATUS...', 'AUTHENTICATI...', and 'CLOUD TO DEV...'. It displays 'No results'.

2. Provide a name for your new device, such as **myDeviceId**, and click **Save**. This action creates a new device identity for your IoT hub.

Create a device

Learn more about creating devices

* Device ID ⓘ
myDeviceId

Authentication type ⓘ
Symmetric key X.509 Self-Signed X.509 CA Signed

* Primary key ⓘ
Enter your primary key

* Secondary key ⓘ
Enter your secondary key

Auto-generate keys ⓘ

Connect this device to an IoT hub ⓘ
Enable Disable

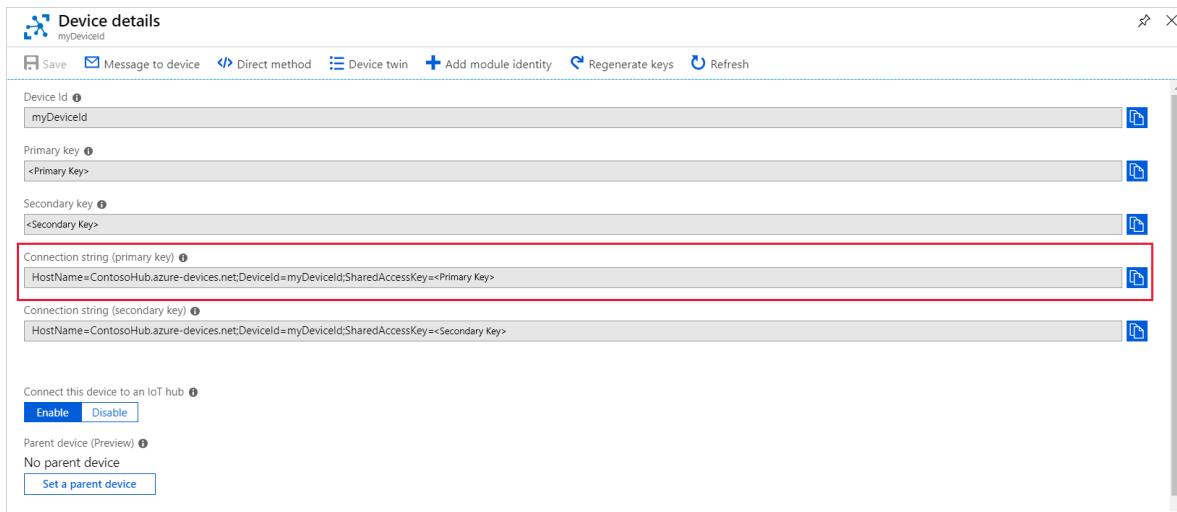
Parent device (Preview) ⓘ
No parent device
Set a parent device

Save

IMPORTANT

The device ID may be visible in the logs collected for customer support and troubleshooting, so make sure to avoid any sensitive information while naming it.

3. After the device is created, open the device from the list in the **IoT devices** pane. Copy the **Connection string---primary key** to use later.



NOTE

The IoT Hub identity registry only stores device identities to enable secure access to the IoT hub. It stores device IDs and keys to use as security credentials, and an enabled/disabled flag that you can use to disable access for an individual device. If your application needs to store other device-specific metadata, it should use an application-specific store. For more information, see [IoT Hub developer guide](#).

Set up Raspberry Pi

Install the Raspbian operating system for Pi

Prepare the microSD card for installation of the Raspbian image.

1. Download Raspbian.

a. [Download Raspbian Stretch](#) (the .zip file).

WARNING

Please use above link to download [raspbian-2017-07-5](#) zip image. The latest version of Raspbian images has some known issues with Wiring-Pi Node, which might cause failure in your next steps.

b. Extract the Raspbian image to a folder on your computer.

2. Install Raspbian to the microSD card.

a. [Download and install the Etcher SD card burner utility](#).

b. Run Etcher and select the Raspbian image that you extracted in step 1.

c. Select the microSD card drive. Etcher may have already selected the correct drive.

d. Click Flash to install Raspbian to the microSD card.

e. Remove the microSD card from your computer when installation is complete. It's safe to remove the microSD card directly because Etcher automatically ejects or unmounts the microSD card upon completion.

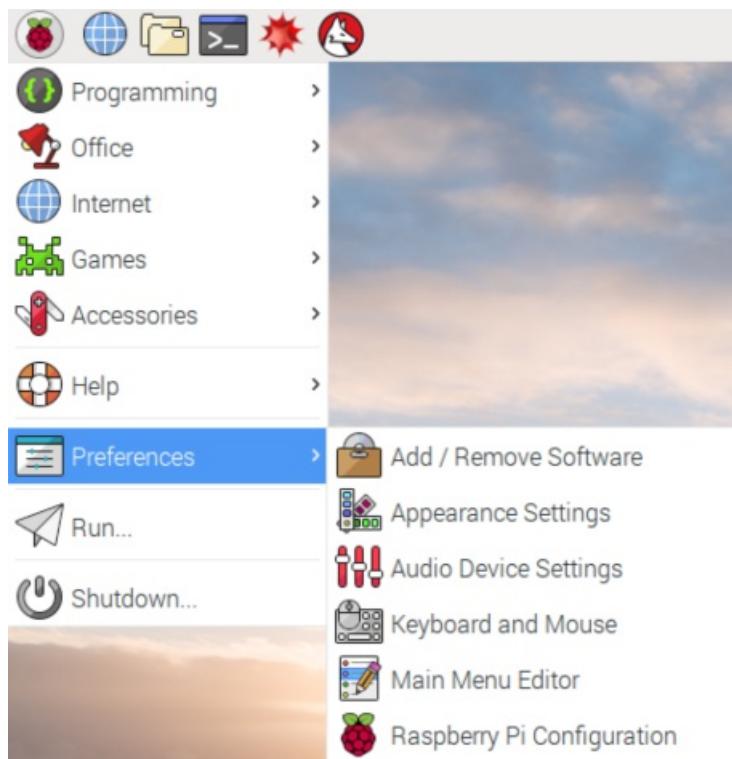
f. Insert the microSD card into Pi.

Enable SSH and I2C

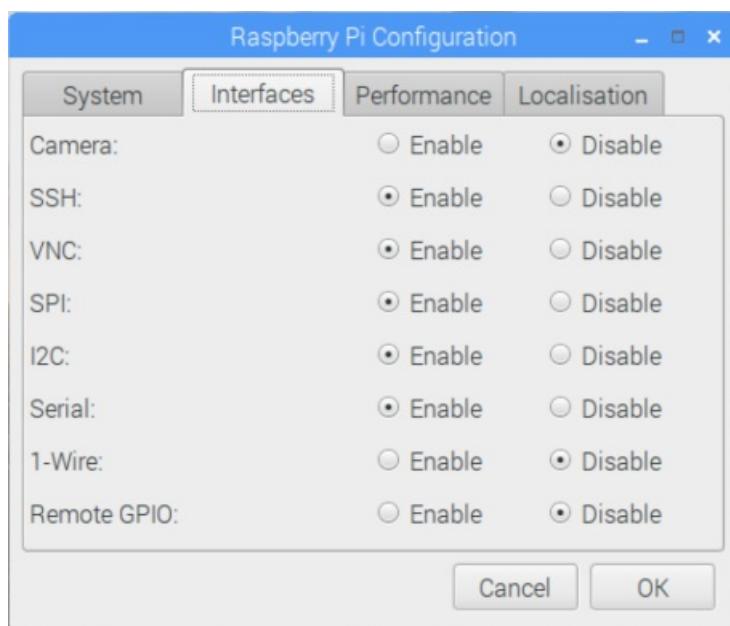
1. Connect Pi to the monitor, keyboard, and mouse.

2. Start Pi and then log in Raspbian by using `pi` as the user name and `raspberry` as the password.

3. Click the Raspberry icon > **Preferences** > **Raspberry Pi Configuration**.



4. On the **Interfaces** tab, set **I2C** and **SSH** to **Enable**, and then click **OK**. If you don't have physical sensors and want to use simulated sensor data, this step is optional.

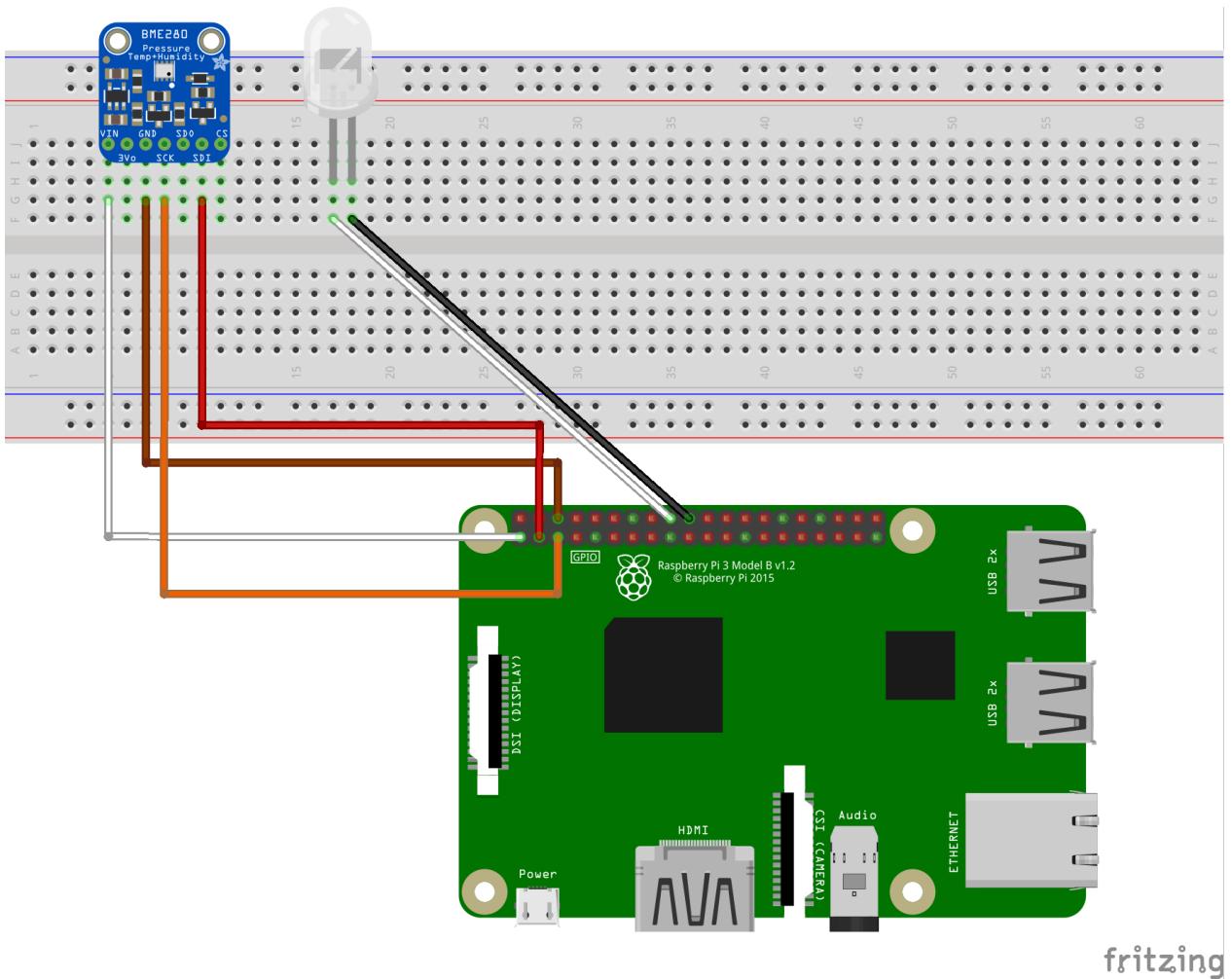


NOTE

To enable SSH and I2C, you can find more reference documents on raspberrypi.org and Adafruit.com.

Connect the sensor to Pi

Use the breadboard and jumper wires to connect an LED and a BME280 to Pi as follows. If you don't have the sensor, [skip this section](#).



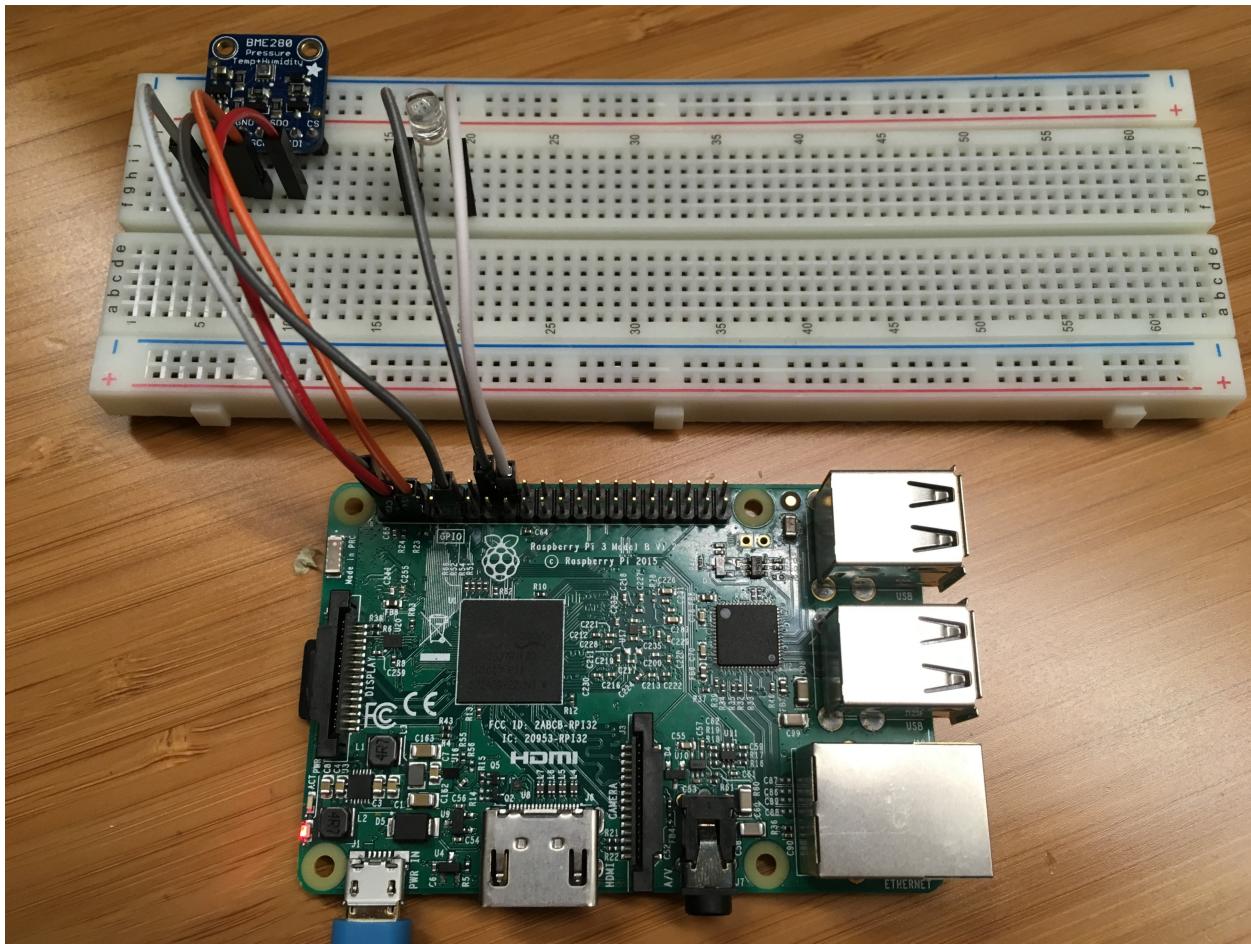
The BME280 sensor can collect temperature and humidity data. The LED blinks when the device sends a message to the cloud.

For sensor pins, use the following wiring:

START (SENSOR & LED)	END (BOARD)	CABLE COLOR
VDD (Pin 5G)	3.3V PWR (Pin 1)	White cable
GND (Pin 7G)	GND (Pin 6)	Brown cable
SDI (Pin 10G)	I2C1 SDA (Pin 3)	Red cable
SCK (Pin 8G)	I2C1 SCL (Pin 5)	Orange cable
LED VDD (Pin 18F)	GPIO 24 (Pin 18)	White cable
LED GND (Pin 17F)	GND (Pin 20)	Black cable

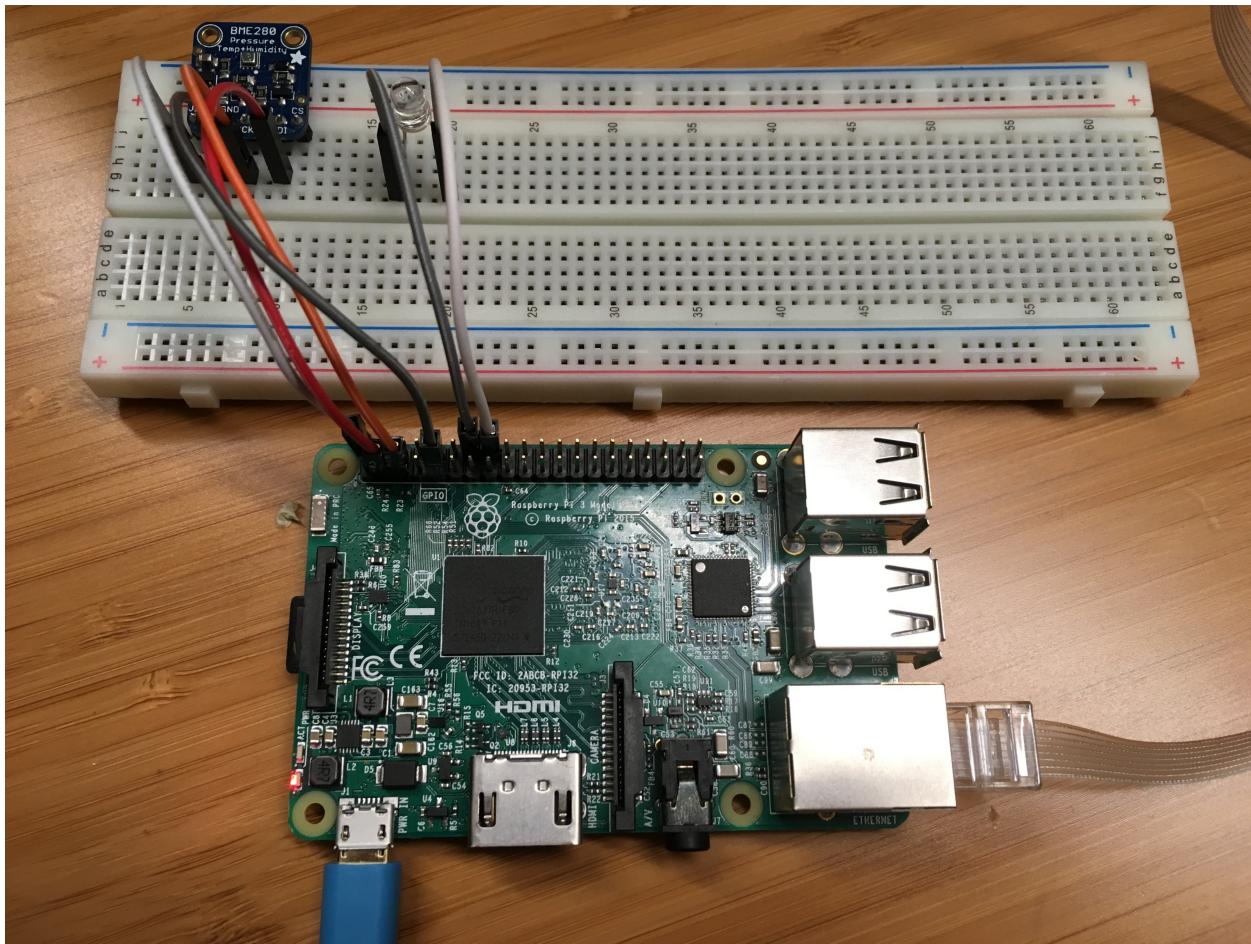
Click to view [Raspberry Pi 2 & 3 pin mappings](#) for your reference.

After you've successfully connected BME280 to your Raspberry Pi, it should be like below image.



Connect Pi to the network

Turn on Pi by using the micro USB cable and the power supply. Use the Ethernet cable to connect Pi to your wired network or follow the [instructions from the Raspberry Pi Foundation](#) to connect Pi to your wireless network. After your Pi has been successfully connected to the network, you need to take a note of the [IP address of your Pi](#).



NOTE

Make sure that Pi is connected to the same network as your computer. For example, if your computer is connected to a wireless network while Pi is connected to a wired network, you might not see the IP address in the devdisco output.

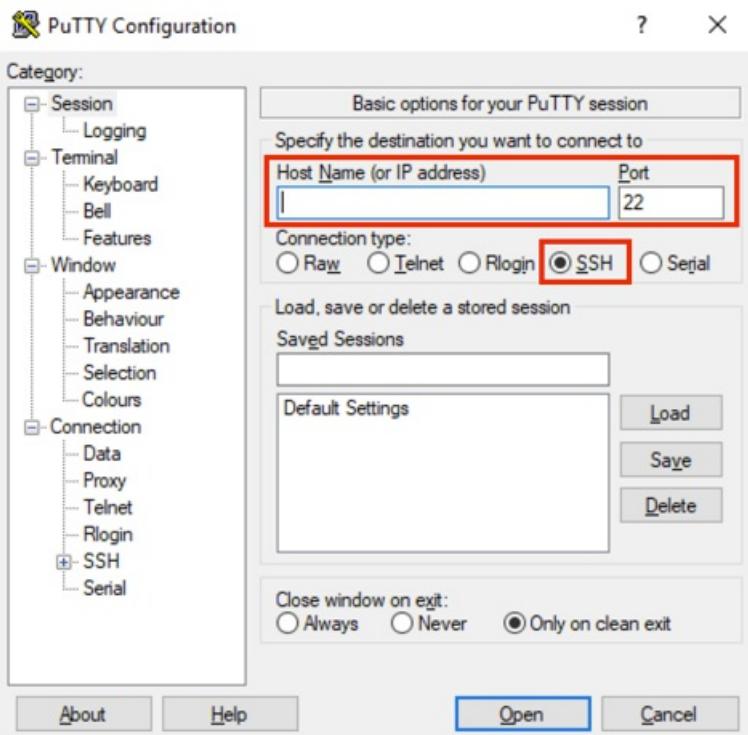
Run a sample application on Pi

Clone sample application and install the prerequisite packages

1. Connect to your Raspberry Pi with one of the following SSH clients from your host computer:

Windows Users

- a. Download and install [PuTTY](#) for Windows.
- b. Copy the IP address of your Pi into the Host name (or IP address) section and select SSH as the connection type.



Mac and Ubuntu Users

Use the built-in SSH client on Ubuntu or macOS. You might need to run `ssh pi@<ip address of pi>` to connect Pi via SSH.

NOTE

The default username is `pi` and the password is `raspberry`.

2. Install Node.js and NPM to your Pi.

First check your Node.js version.

```
node -v
```

If the version is lower than 4.x, or if there is no Node.js on your Pi, install the latest version.

```
curl -sL https://deb.nodesource.com/setup_4.x | sudo -E bash  
sudo apt-get -y install nodejs
```

3. Clone the sample application.

```
git clone https://github.com/Azure-Samples/iot-hub-node-raspberrypi-client-app
```

4. Install all packages for the sample. The installation includes Azure IoT device SDK, BME280 Sensor library, and Wiring Pi library.

```
cd iot-hub-node-raspberrypi-client-app  
sudo npm install
```

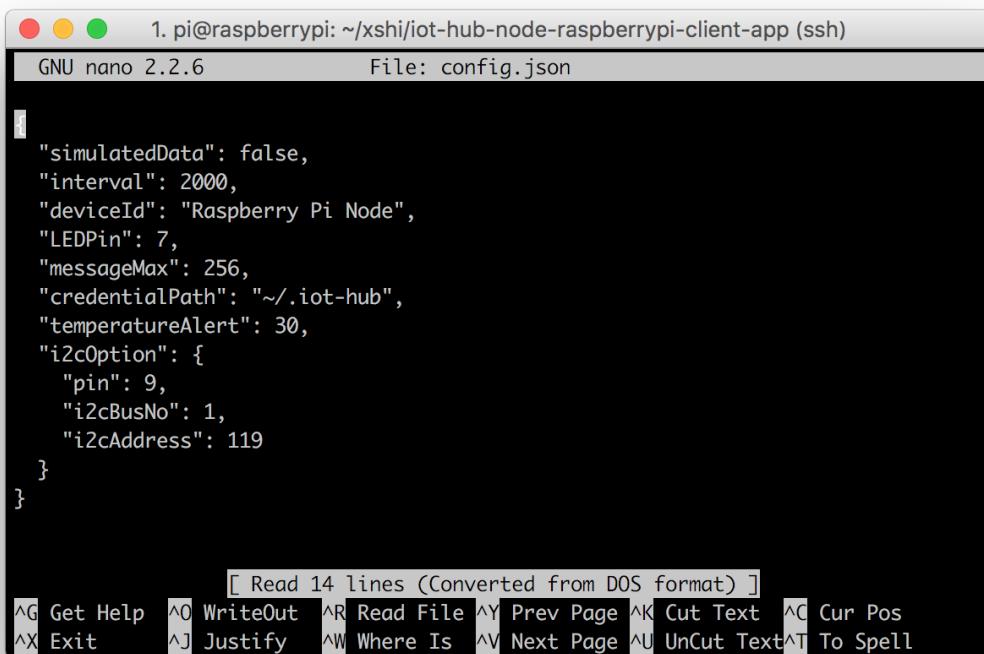
NOTE

It might take several minutes to finish this installation process depending on your network connection.

Configure the sample application

1. Open the config file by running the following commands:

```
nano config.json
```



```
pi@raspberrypi: ~/xshi/iot-hub-node-raspberrypi-client-app (ssh)
GNU nano 2.2.6          File: config.json

{
    "simulatedData": false,
    "interval": 2000,
    "deviceId": "Raspberry Pi Node",
    "LEDPin": 7,
    "messageMax": 256,
    "credentialPath": "~/.iot-hub",
    "temperatureAlert": 30,
    "i2cOption": {
        "pin": 9,
        "i2cBusNo": 1,
        "i2cAddress": 119
    }
}

[ Read 14 lines (Converted from DOS format) ]
^G Get Help ^O WriteOut ^R Read File ^Y Prev Page ^K Cut Text ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is ^V Next Page ^U Uncut Text ^T To Spell
```

There are two items in this file you can configure. The first one is `interval`, which defines the time interval (in milliseconds) between messages sent to the cloud. The second one is `simulatedData`, which is a Boolean value for whether to use simulated sensor data or not.

If you **don't have the sensor**, set the `simulatedData` value to `true` to make the sample application create and use simulated sensor data.

2. Save and exit by typing Control-O > Enter > Control-X.

Run the sample application

Run the sample application by running the following command:

```
sudo node index.js '<YOUR AZURE IOT HUB DEVICE CONNECTION STRING>'
```

NOTE

Make sure you copy-paste the device connection string into the single quotes.

You should see the following output that shows the sensor data and the messages that are sent to your IoT hub.

```
pi@raspberrypi:~/xshi/iot-hub-node-raspberrypi-client-app (ssh)
pi@raspberrypi:~/xshi/iot-hub-node-raspberrypi-client-app $ sudo node index.js 
HostName=IoTGetStarted.azure-devices.net;DeviceId=new-device;SharedAccessKey=d0q
ltgHj6U8Wb+3PX5I9ism5eIGtJLRTb89M7C3eUQ0='
Sending message: {"messageId":1,"deviceId":"Raspberry Pi Node","temperature":22.
09817088597284,"humidity":79.44195810046365}
Message sent to Azure IoT Hub
Sending message: {"messageId":2,"deviceId":"Raspberry Pi Node","temperature":26.
183512024547063,"humidity":61.42521225412357}
Message sent to Azure IoT Hub
Sending message: {"messageId":3,"deviceId":"Raspberry Pi Node","temperature":29.
520917564174873,"humidity":62.00662798413029}
Message sent to Azure IoT Hub
Sending message: {"messageId":4,"deviceId":"Raspberry Pi Node","temperature":22.
591091037492344,"humidity":70.1062754469173}
Message sent to Azure IoT Hub
Sending message: {"messageId":5,"deviceId":"Raspberry Pi Node","temperature":26.
451696863853265,"humidity":72.71690012385488}
Message sent to Azure IoT Hub
Sending message: {"messageId":6,"deviceId":"Raspberry Pi Node","temperature":25.
```

Next steps

You've run a sample application to collect sensor data and send it to your IoT hub. To see the messages that your Raspberry Pi has sent to your IoT hub or send messages to your Raspberry Pi, see the [Use Azure IoT Tools for Visual Studio Code to send and receive messages between your device and IoT Hub](#).

To continue to get started with Azure IoT Hub and to explore all extended IoT scenarios, see the following:

- [Manage cloud device messaging with Azure IoT Hub Toolkit extension for Visual Studio Code](#)
- [Manage devices with Azure IoT Hub Toolkit extension for Visual Studio Code](#)
- [Set up message routing](#)
- [Use Power BI to visualize real-time sensor data from your IoT hub](#)
- [Use the Web Apps feature of Azure App Service to visualize real-time sensor data from your IoT hub](#)
- [Forecast weather by using the sensor data from your IoT hub in Azure Machine Learning](#)
- [Use Logic Apps for remote monitoring and notifications](#)

Connect Raspberry Pi to Azure IoT Hub (C)

3/6/2019 • 9 minutes to read

In this tutorial, you begin by learning the basics of working with Raspberry Pi that's running Raspbian. You then learn how to seamlessly connect your devices to the cloud by using [Azure IoT Hub](#). For Windows 10 IoT Core samples, go to the [Windows Dev Center](#).

Don't have a kit yet? Try [Raspberry Pi online simulator](#). Or buy a new kit [here](#).

What you do

- Create an IoT hub.
- Register a device for Pi in your IoT hub.
- Setup Raspberry Pi.
- Run a sample application on Pi to send sensor data to your IoT hub.

Connect Raspberry Pi to an IoT hub that you create. Then you run a sample application on Pi to collect temperature and humidity data from a BME280 sensor. Finally, you send the sensor data to your IoT hub.

What you learn

- How to create an Azure IoT hub and get your new device connection string.
- How to connect Pi with a BME280 sensor.
- How to collect sensor data by running a sample application on Pi.
- How to send sensor data to your IoT hub.

What you need



- The Raspberry Pi 2 or Raspberry Pi 3 board.
- An active Azure subscription. If you don't have an Azure account, [create a free Azure trial account](#) in just a few minutes.
- A monitor, a USB keyboard, and mouse that connect to Pi.
- A Mac or a PC that is running Windows or Linux.

- An Internet connection.
- A 16 GB or above microSD card.
- A USB-SD adapter or microSD card to burn the operating system image onto the microSD card.
- A 5-volt 2-amp power supply with the 6-foot micro USB cable.

The following items are optional:

- An assembled Adafruit BME280 temperature, pressure, and humidity sensor.
- A breadboard.
- 6 F/M jumper wires.
- A diffused 10-mm LED.

NOTE

These items are optional because the code sample support simulated sensor data.

Create an IoT hub

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose +**Create a resource**, then choose **Internet of Things**.
3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.

Home > New > IoT hub

IoT hub
Microsoft

Basics [Size and scale](#) [Review + create](#)

Create an IoT Hub to help you connect, monitor, and manage billions of your IoT assets. [Learn More](#)

PROJECT DETAILS

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

* Subscription <small>i</small>	<input type="text" value="Microsoft Azure Internal Consumption"/>
* Resource Group <small>i</small>	<input checked="" type="radio"/> Create new <input type="radio"/> Use existing <input type="text" value="contoso-hub-rgrp"/>
* Region <small>i</small>	<input type="text" value="West US"/>
* IoT Hub Name <small>i</small>	<input type="text" value="contoso-test-hub"/>

[Review + create](#) [Next: Size and scale >](#) [Automation options](#)

Fill in the fields.

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

The screenshot shows the 'Size and scale' tab of the IoT hub creation wizard. It includes sections for selecting the pricing tier (S1: Standard tier), specifying the number of units (1 unit selected), and configuring advanced settings like device-to-cloud partitions (4 selected). The page also lists enabled features: Device-to-cloud-messages, Message routing, Cloud-to-device commands, IoT Edge, and Device management.

Home > New > IoT hub

IoT hub
Microsoft

Basics **Size and scale** **Review + create**

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS

* Pricing and scale tier [?](#) S1: Standard tier [?](#) Learn how to choose the right IoT Hub tier for your solution

Number of S1 IoT Hub units [?](#) 1 This determines your IoT Hub scale capability and can be changed as your need increases.

Device-to-cloud-messages [?](#) Enabled

Message routing [?](#) Enabled

Cloud-to-device commands [?](#) Enabled

IoT Edge [?](#) Enabled

Device management [?](#) Enabled

Advanced Settings

Device-to-cloud partitions [?](#) 4

Review + create [« Previous: Basics](#) [Automation options](#)

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: The number of messages allowed per unit per day depends on your hub's pricing tier. For example, if you want the IoT hub to support ingress of 700,000 messages, you choose two S1 tier units.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

Advanced / Device-to-cloud partitions: This property relates the device-to-cloud messages to the

number of simultaneous readers of the messages. Most IoT hubs only need four partitions.

5. Click **Review + create** to review your choices. You see something similar to this screen.

The screenshot shows the 'Review + create' step of creating an IoT hub. It displays basic information like Subscription, Resource Group, Region, and IoT Hub Name, as well as size and scale details like Pricing tier, Number of units, Messages per day, and Cost per month. At the bottom, there are buttons for 'Create' (highlighted), '« Previous: Size and scale', and 'Automation options'.

Basics	Size and scale	Review + create
BASICS		
Subscription ⓘ	Microsoft Azure Internal Consumption	
Resource Group ⓘ	contoso-hub-rgrp	
Region ⓘ	West US	
IoT Hub Name ⓘ	contoso-test-hub	
SIZE AND SCALE		
Pricing and scale tier ⓘ	S1	
Number of S1 IoT Hub units ⓘ	1	
Messages per day ⓘ	400,000	
Cost per month	25.00 USD	

6. Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Retrieve connection string for IoT hub

After your hub has been created, retrieve the connection string for the hub. This is used to connect devices and applications to your hub.

1. Click on your hub to see the IoT Hub pane with Settings, and so on. Click **Shared access policies**.
2. In **Shared access policies**, select the **iothubowner** policy.
3. Under **Shared access keys**, copy the **Connection string -- primary key** to be used later.

The screenshot shows the 'Shared access policies' section of the IoT Hub. A new policy named 'iothubowner' is being created. The 'Permissions' section includes 'Registry read', 'Registry write', 'Service connect', and 'Device connect'. The 'Shared access keys' section shows the 'Primary key' connection string, which is highlighted with a red box.

For more information, see [Access control](#) in the "IoT Hub developer guide."

Register a new device in the IoT hub

In this section, you create a device identity in the identity registry in your IoT hub. A device cannot connect to IoT hub unless it has an entry in the identity registry. For more information, see the "Identity registry" section of the [IoT Hub developer guide](#)

1. In your IoT hub navigation menu, open **IoT Devices**, then click **Add** to register a new device in your IoT hub.

The screenshot shows the 'IoT devices' section of the IoT Hub. The 'Add' button is highlighted with a red box. The 'IoT devices' option is selected in the navigation menu, also highlighted with a red box.

2. Provide a name for your new device, such as **myDeviceId**, and click **Save**. This action creates a new device

identity for your IoT hub.

Create a device

Learn more about creating devices

* Device ID ⓘ
myDeviceId

Authentication type ⓘ
Symmetric key X.509 Self-Signed X.509 CA Signed

* Primary key ⓘ
Enter your primary key

* Secondary key ⓘ
Enter your secondary key

Auto-generate keys ⓘ

Connect this device to an IoT hub ⓘ
Enable Disable

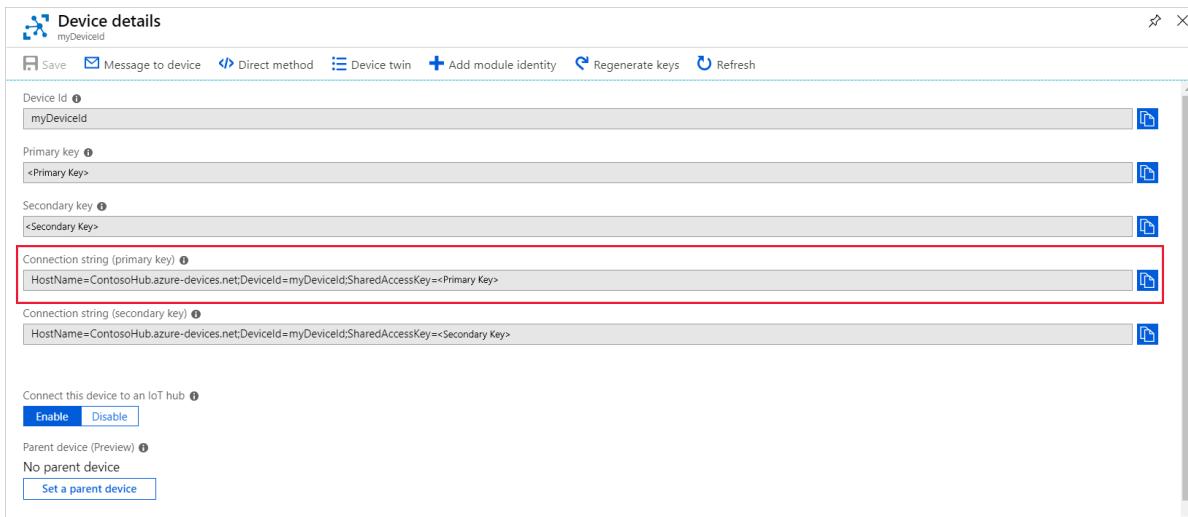
Parent device (Preview) ⓘ
No parent device
Set a parent device

Save

IMPORTANT

The device ID may be visible in the logs collected for customer support and troubleshooting, so make sure to avoid any sensitive information while naming it.

- After the device is created, open the device from the list in the **IoT devices** pane. Copy the **Connection string--primary key** to use later.



NOTE

The IoT Hub identity registry only stores device identities to enable secure access to the IoT hub. It stores device IDs and keys to use as security credentials, and an enabled/disabled flag that you can use to disable access for an individual device. If your application needs to store other device-specific metadata, it should use an application-specific store. For more information, see [IoT Hub developer guide](#).

Setup Raspberry Pi

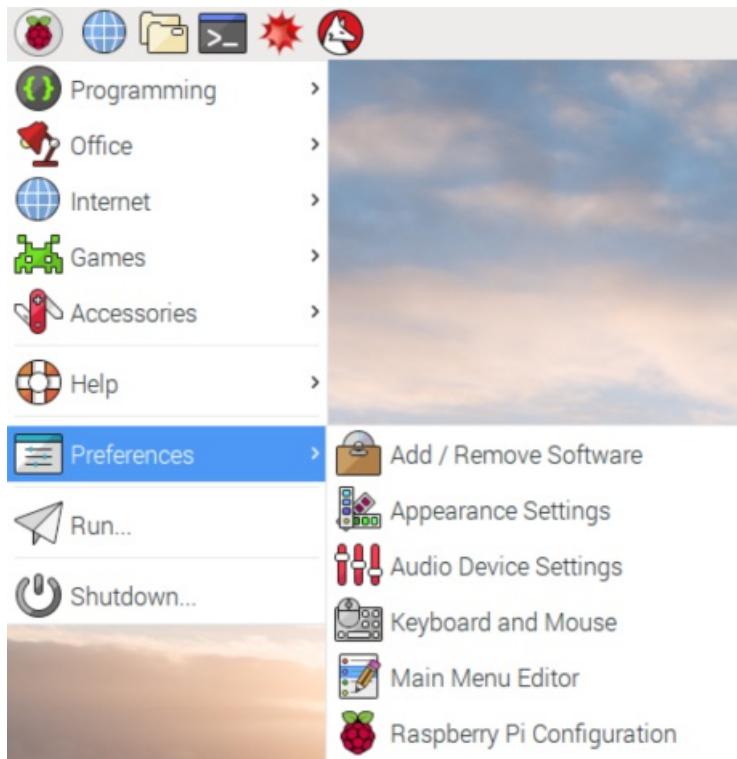
Install the Raspbian operating system for Pi

Prepare the microSD card for installation of the Raspbian image.

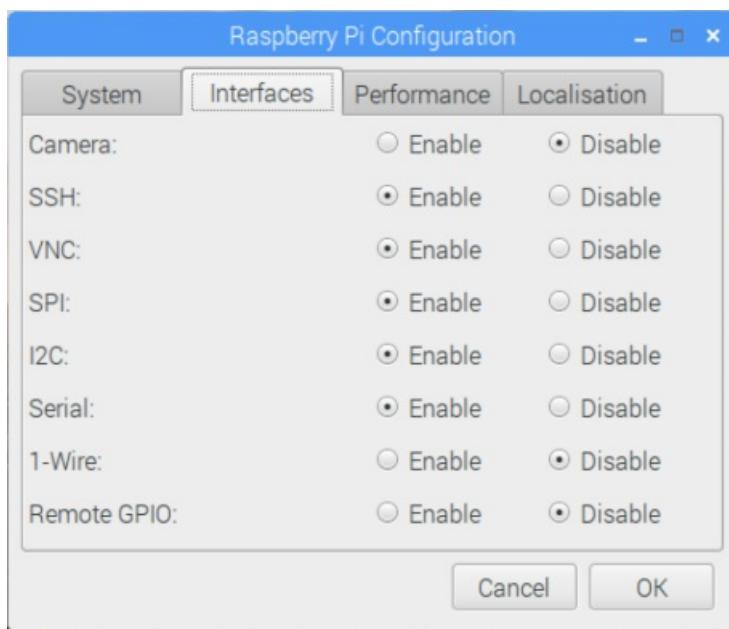
1. Download Raspbian.
 - a. [Download Raspbian Stretch with Desktop](#) (the .zip file).
 - b. Extract the Raspbian image to a folder on your computer.
2. Install Raspbian to the microSD card.
 - a. [Download and install the Etcher SD card burner utility](#).
 - b. Run Etcher and select the Raspbian image that you extracted in step 1.
 - c. Select the microSD card drive. Note that Etcher may have already selected the correct drive.
 - d. Click Flash to install Raspbian to the microSD card.
 - e. Remove the microSD card from your computer when installation is complete. It's safe to remove the microSD card directly because Etcher automatically ejects or unmounts the microSD card upon completion.
 - f. Insert the microSD card into Pi.

Enable SSH and SPI

1. Connect Pi to the monitor, keyboard and mouse, start Pi and then log in Raspbian by using `pi` as the user name and `raspberry` as the password.
2. Click the Raspberry icon > **Preferences > Raspberry Pi Configuration**.



3. On the **Interfaces** tab, set **SPI** and **SSH** to **Enable**, and then click **OK**. If you don't have physical sensors and want to use simulated sensor data, this step is optional.

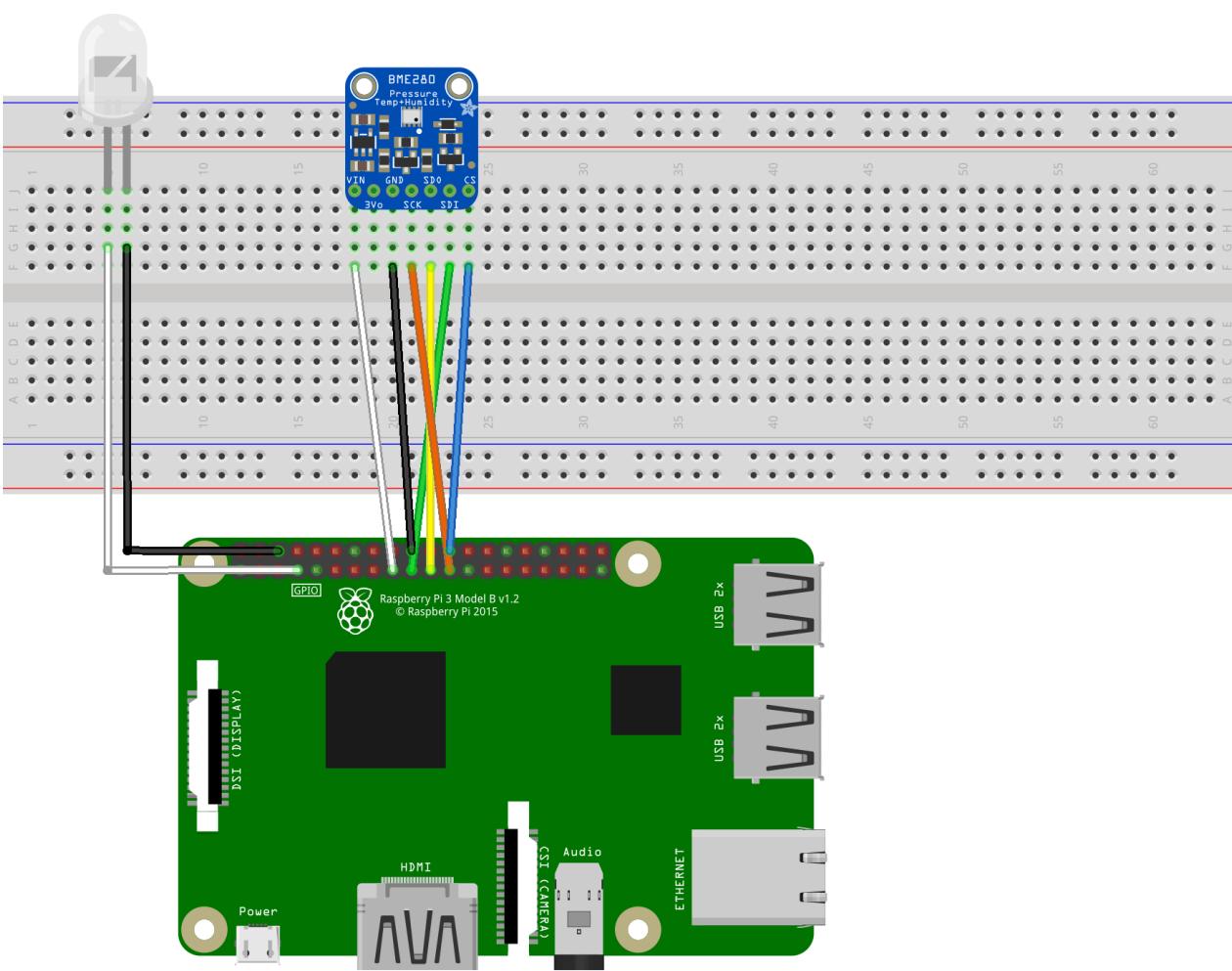


NOTE

To enable SSH and SPI, you can find more reference documents on [raspberrypi.org](https://www.raspberrypi.org) and [RASPI-CONFIG](#).

Connect the sensor to Pi

Use the breadboard and jumper wires to connect an LED and a BME280 to Pi as follows. If you don't have the sensor, [skip this section](#).



fritzing

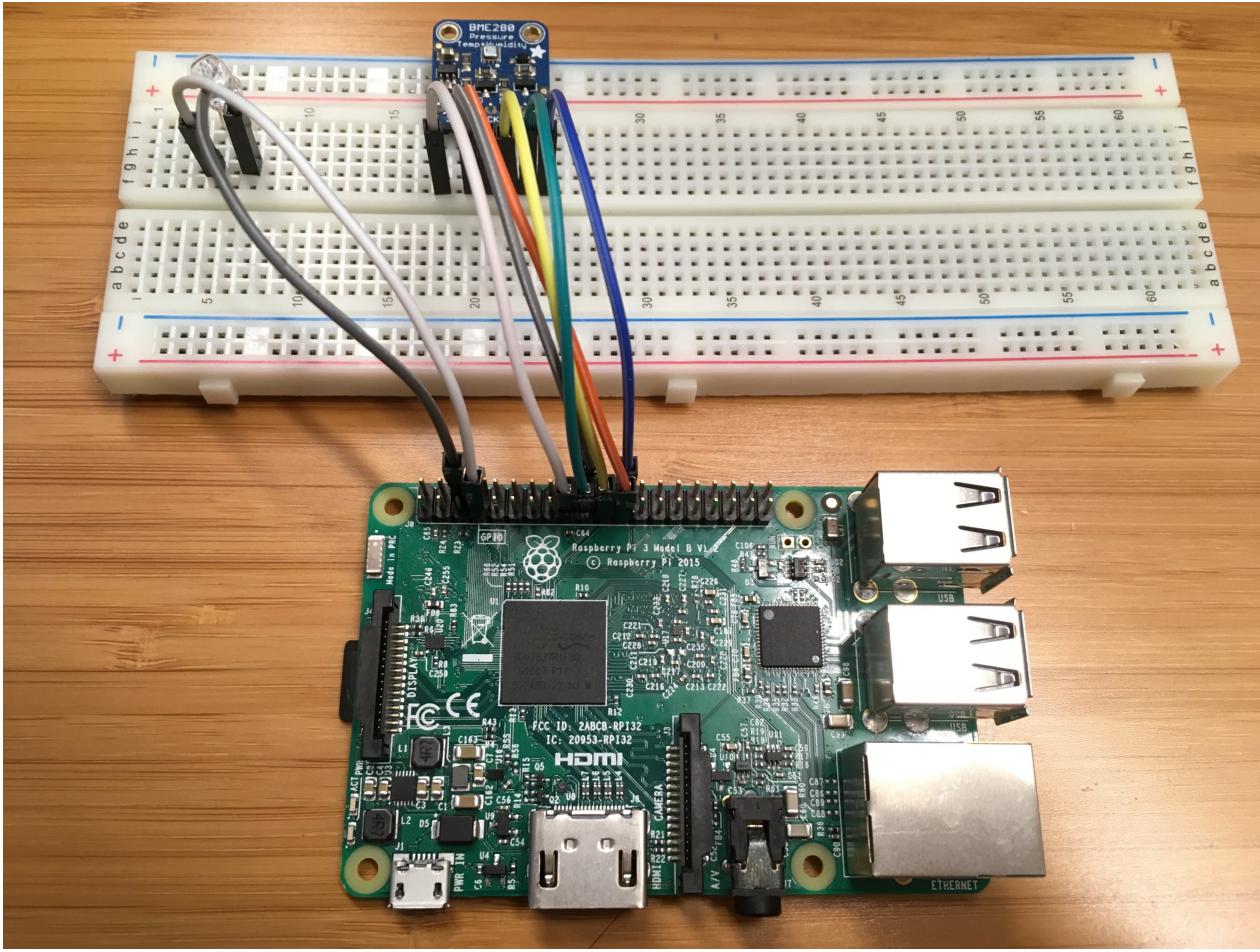
The BME280 sensor can collect temperature and humidity data. And the LED will blink if there is a communication between device and the cloud.

For sensor pins, use the following wiring:

START (SENSOR & LED)	END (BOARD)	CABLE COLOR
LED VDD (Pin 5G)	GPIO 4 (Pin 7)	White cable
LED GND (Pin 6G)	GND (Pin 6)	Black cable
VDD (Pin 18F)	3.3V PWR (Pin 17)	White cable
GND (Pin 20F)	GND (Pin 20)	Black cable
SCK (Pin 21F)	SPI0 SCLK (Pin 23)	Orange cable
SDO (Pin 22F)	SPI0 MISO (Pin 21)	Yellow cable
SDI (Pin 23F)	SPI0 MOSI (Pin 19)	Green cable
CS (Pin 24F)	SPI0 CS (Pin 24)	Blue cable

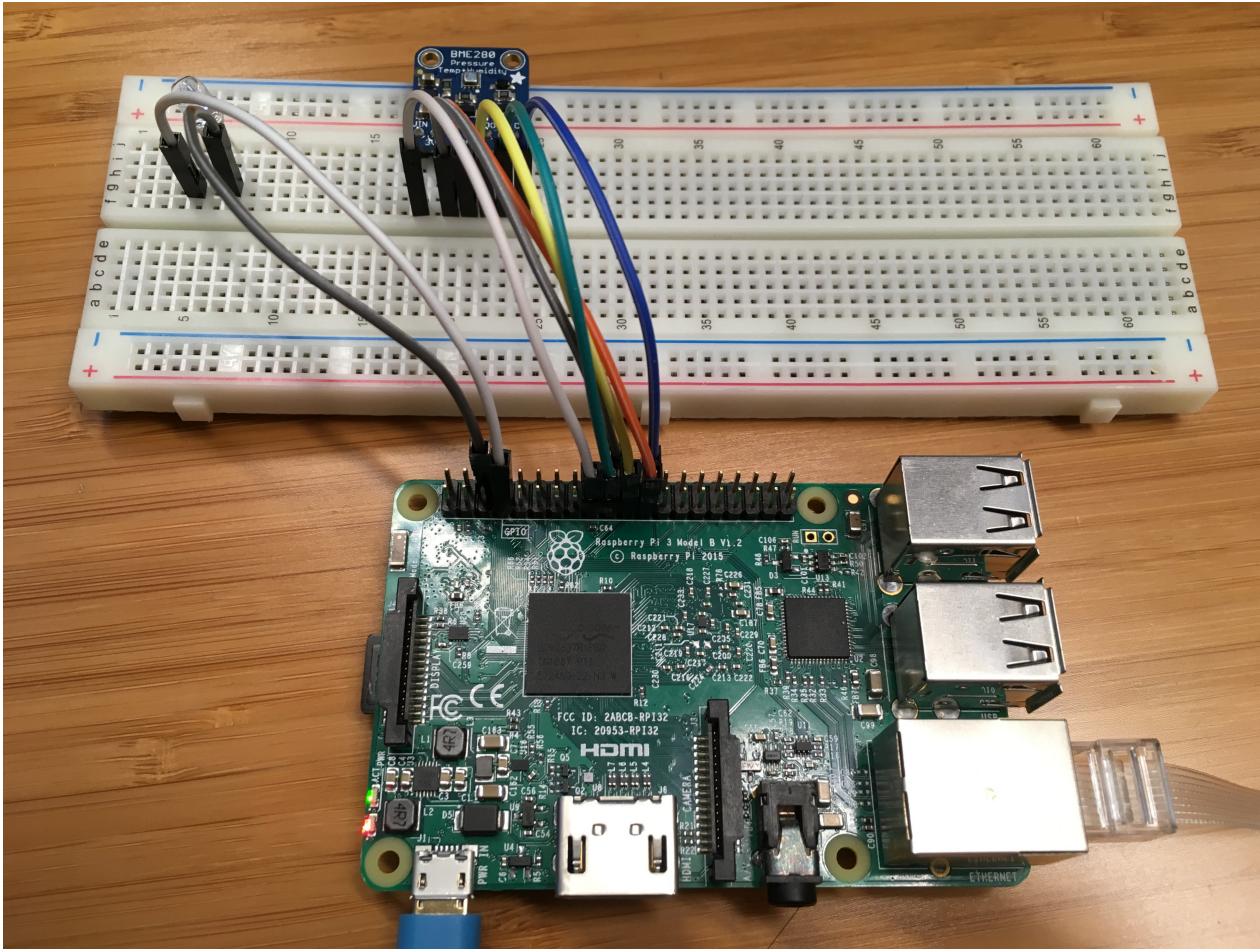
Click to view [Raspberry Pi 2 & 3 Pin mappings](#) for your reference.

After you've successfully connected BME280 to your Raspberry Pi, it should be like below image.



Connect Pi to the network

Turn on Pi by using the micro USB cable and the power supply. Use the Ethernet cable to connect Pi to your wired network or follow the [instructions from the Raspberry Pi Foundation](#) to connect Pi to your wireless network. After your Pi has been successfully connected to the network, you need to take a note of the [IP address of your Pi](#).



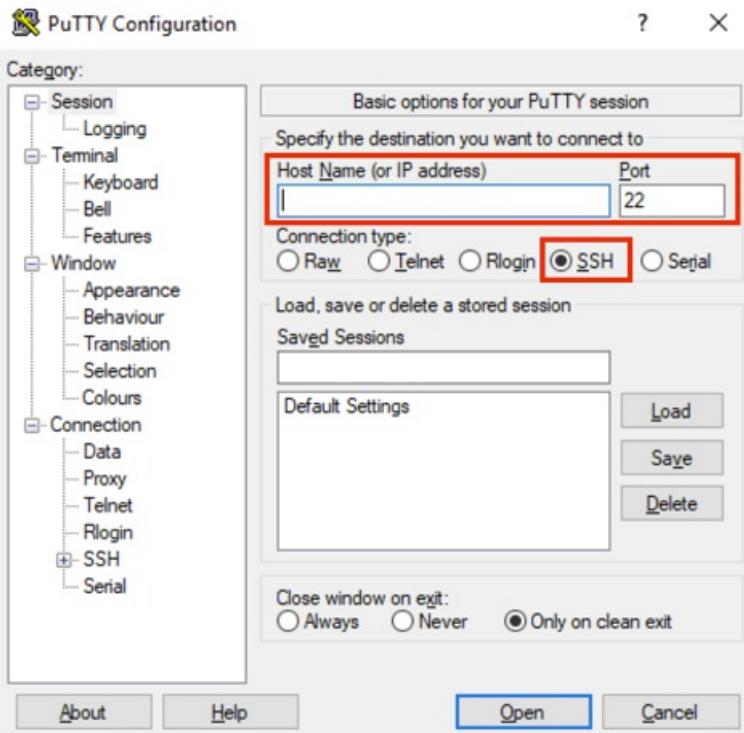
Run a sample application on Pi

Login to your Raspberry Pi

1. Use one of the following SSH clients from your host computer to connect to your Raspberry Pi.

Windows Users

- a. Download and install [PuTTY](#) for Windows.
- b. Copy the IP address of your Pi into the Host name (or IP address) section and select SSH as the connection type.



Mac and Ubuntu Users

Use the built-in SSH client on Ubuntu or macOS. You might need to run `ssh pi@<ip address of pi>` to connect Pi via SSH.

NOTE

The default username is `pi`, and the password is `raspberry`.

Configure the sample application

- Clone the sample application by running the following command:

```
sudo apt-get install git-core
git clone https://github.com/Azure-Samples/iot-hub-c-raspberrypi-client-app.git
```

- Run setup script:

```
cd ./iot-hub-c-raspberrypi-client-app
sudo chmod u+x setup.sh
sudo ./setup.sh
```

NOTE

If you **don't have a physical BME280**, you can use '--simulated-data' as command line parameter to simulate temperature&humidity data. `sudo ./setup.sh --simulated-data`

Build and run the sample application

- Build the sample application by running the following command:

```
cmake . && make
```

```
1. pi@raspberrypi: ~/xshi/iot-hub-c-raspberrypi-client-app (ssh)
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/pi/xshi/iot-hub-c-raspberrypi-client-ap
p
Scanning dependencies of target app
[ 25%] Building C object CMakeFiles/app.dir/main.c.o
[ 50%] Building C object CMakeFiles/app.dir/bme280.c.o
[ 75%] Building C object CMakeFiles/app.dir/wiring.c.o
[100%] Linking C executable app
[100%] Built target app
pi@raspberrypi:~/xshi/iot-hub-c-raspberrypi-client-app $
```

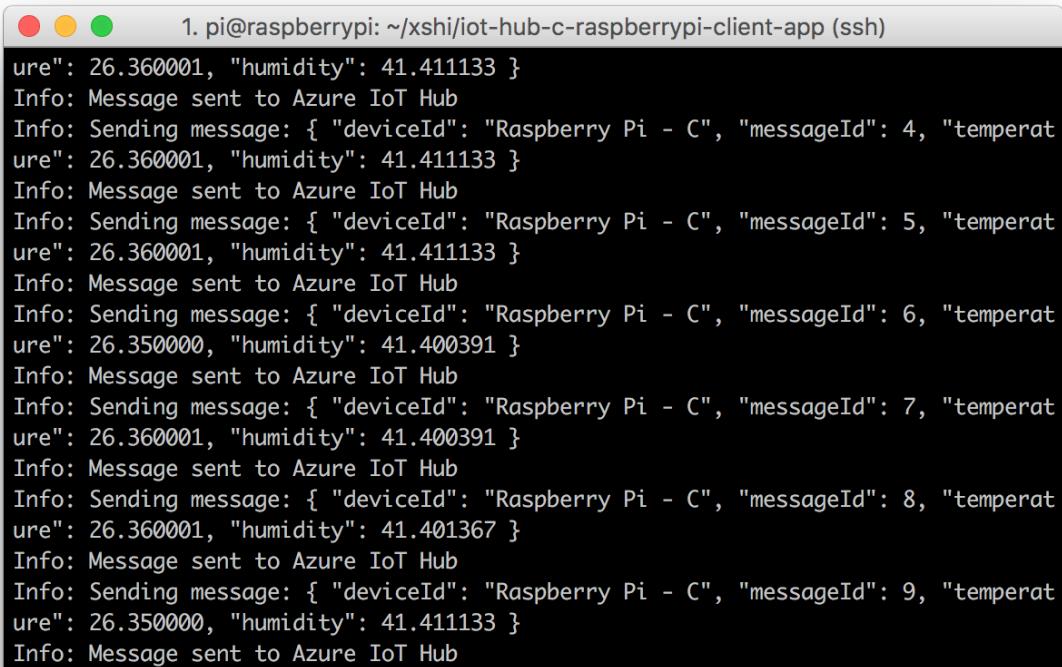
- Run the sample application by running the following command:

```
sudo ./app '<DEVICE CONNECTION STRING>'
```

NOTE

Make sure you copy-paste the device connection string into the single quotes.

You should see the following output that shows the sensor data and the messages that are sent to your IoT hub.



```
1. pi@raspberrypi: ~/xshi/iot-hub-c-raspberrypi-client-app (ssh)
ure": 26.360001, "humidity": 41.411133 }
Info: Message sent to Azure IoT Hub
Info: Sending message: { "deviceId": "Raspberry Pi - C", "messageId": 4, "temperat
ure": 26.360001, "humidity": 41.411133 }
Info: Message sent to Azure IoT Hub
Info: Sending message: { "deviceId": "Raspberry Pi - C", "messageId": 5, "temperat
ure": 26.360001, "humidity": 41.411133 }
Info: Message sent to Azure IoT Hub
Info: Sending message: { "deviceId": "Raspberry Pi - C", "messageId": 6, "temperat
ure": 26.350000, "humidity": 41.400391 }
Info: Message sent to Azure IoT Hub
Info: Sending message: { "deviceId": "Raspberry Pi - C", "messageId": 7, "temperat
ure": 26.360001, "humidity": 41.400391 }
Info: Message sent to Azure IoT Hub
Info: Sending message: { "deviceId": "Raspberry Pi - C", "messageId": 8, "temperat
ure": 26.360001, "humidity": 41.401367 }
Info: Message sent to Azure IoT Hub
Info: Sending message: { "deviceId": "Raspberry Pi - C", "messageId": 9, "temperat
ure": 26.350000, "humidity": 41.411133 }
Info: Message sent to Azure IoT Hub
```

Next steps

You've run a sample application to collect sensor data and send it to your IoT hub. To see the messages that your Raspberry Pi has sent to your IoT hub or send messages to your Raspberry Pi, see the [Use Azure IoT Tools for Visual Studio Code to send and receive messages between your device and IoT Hub](#).

To continue to get started with Azure IoT Hub and to explore all extended IoT scenarios, see the following:

- [Manage cloud device messaging with Azure IoT Hub Toolkit extension for Visual Studio Code](#)
- [Manage devices with Azure IoT Hub Toolkit extension for Visual Studio Code](#)
- [Set up message routing](#)
- [Use Power BI to visualize real-time sensor data from your IoT hub](#)
- [Use the Web Apps feature of Azure App Service to visualize real-time sensor data from your IoT hub](#)
- [Forecast weather by using the sensor data from your IoT hub in Azure Machine Learning](#)
- [Use Logic Apps for remote monitoring and notifications](#)

Connect IoT DevKit AZ3166 to Azure IoT Hub

3/6/2019 • 8 minutes to read

You can use the [MXChip IoT DevKit](#) to develop and prototype Internet of Things (IoT) solutions that take advantage of Microsoft Azure services. It includes an Arduino-compatible board with rich peripherals and sensors, an open-source board package, and a growing [projects catalog](#).

What you do

Connect the DevKit to an Azure IoT hub that you create. Then collect the temperature and humidity data from sensors, and send the data to the IoT hub.

Don't have a DevKit yet? Try the [DevKit simulator](#) or [purchase a DevKit](#).

What you learn

- How to connect the IoT DevKit to a wireless access point and prepare your development environment.
- How to create an IoT hub and register a device for the MXChip IoT DevKit.
- How to collect sensor data by running a sample application on the MXChip IoT DevKit.
- How to send the sensor data to your IoT hub.

What you need

- An MXChip IoT DevKit board with a Micro-USB cable. [Get it now](#).
- A computer running Windows 10 or macOS 10.10+.
- An active Azure subscription. [Activate a free 30-day trial Microsoft Azure account](#).

Prepare your hardware

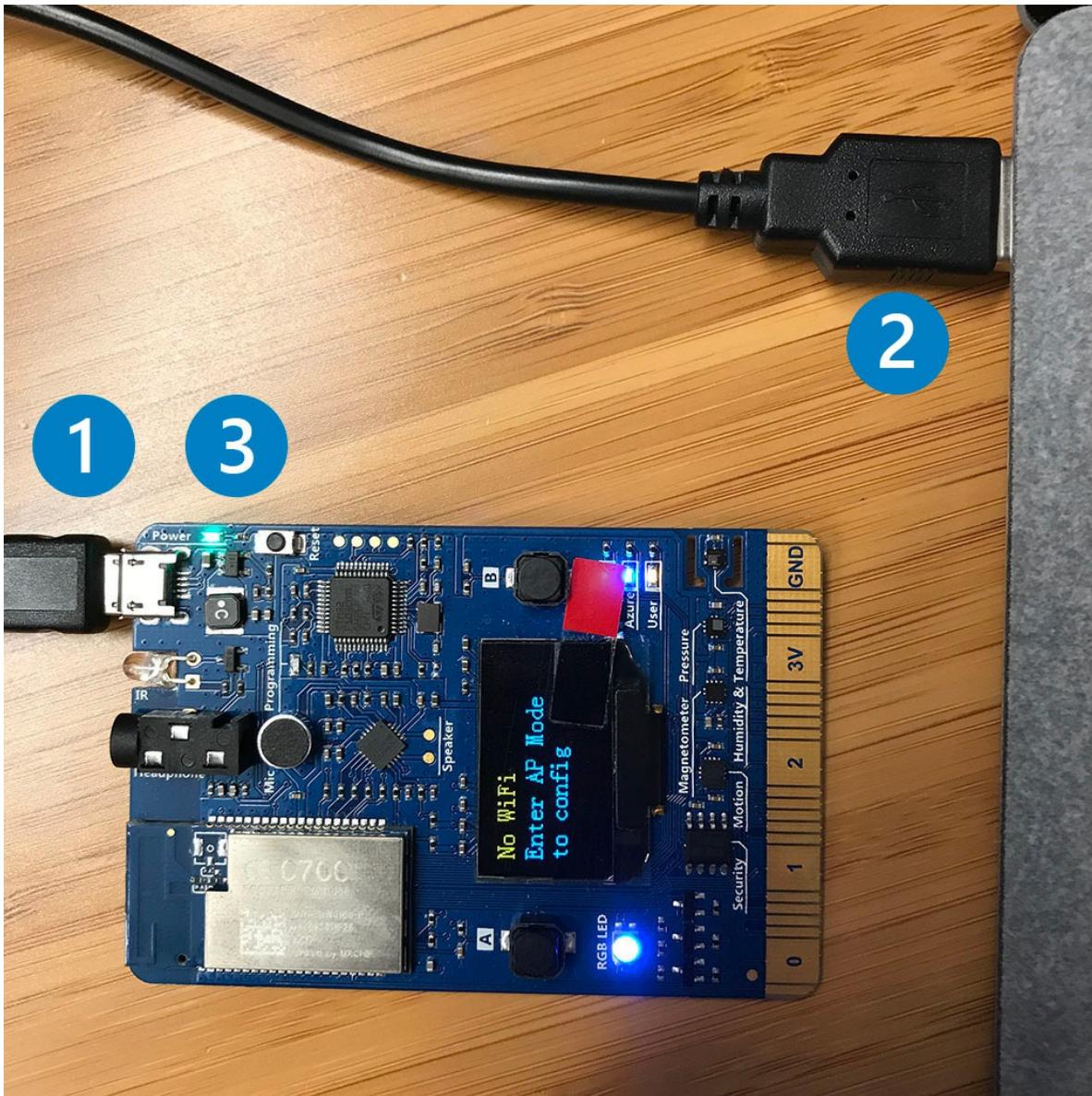
Hook up the following hardware to your computer:

- DevKit board
- Micro-USB cable



To connect the DevKit to your computer, follow these steps:

1. Connect the USB end to your computer.
2. Connect the Micro-USB end to the DevKit.
3. The green LED for power confirms the connection.

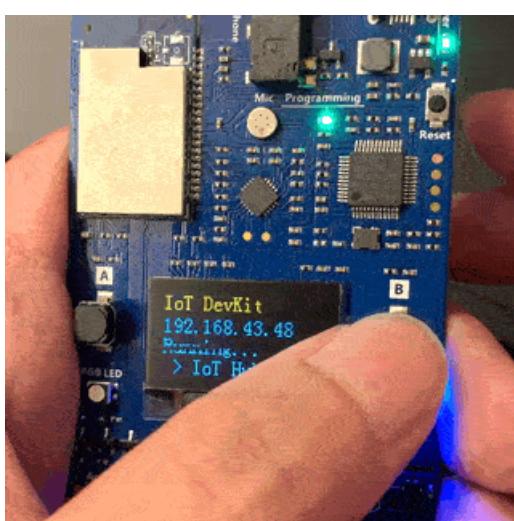
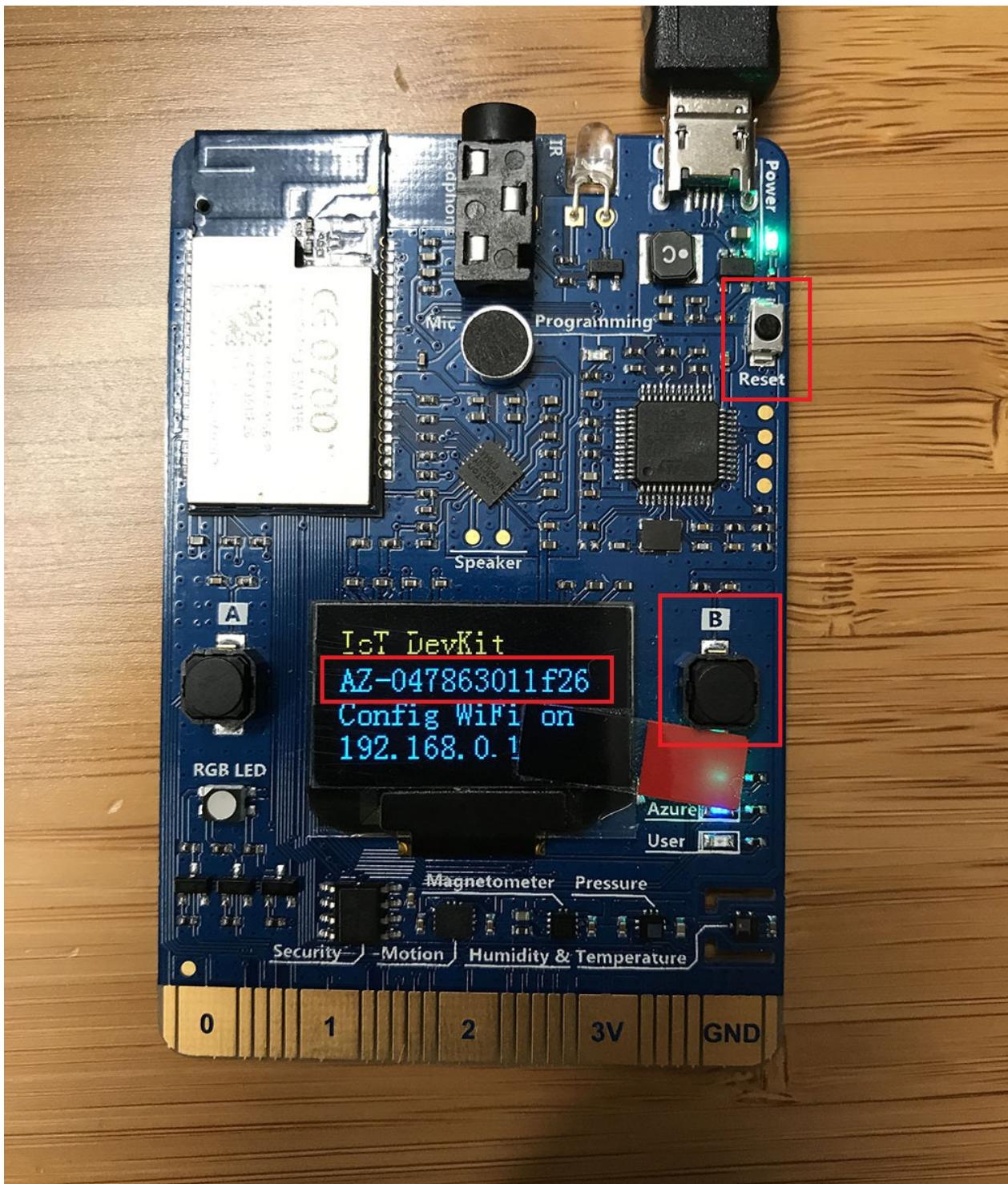


Configure Wi-Fi

IoT projects rely on internet connectivity. Use the following instructions to configure the DevKit to connect to Wi-Fi.

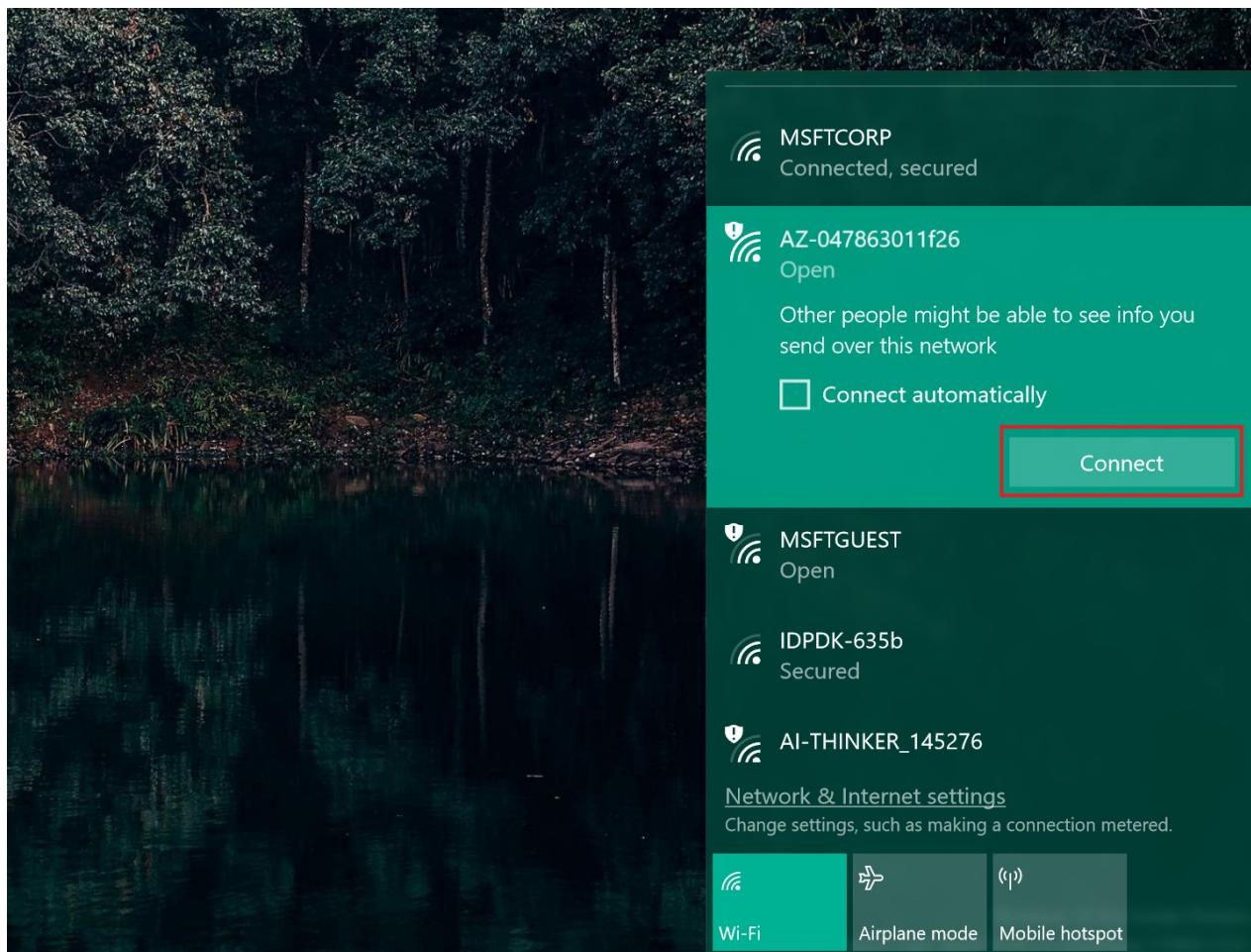
Enter AP mode

Hold down button B, push and release the reset button, and then release button B. Your DevKit enters AP mode for configuring Wi-Fi. The screen displays the service set identifier (SSID) of the DevKit and the configuration portal IP address.



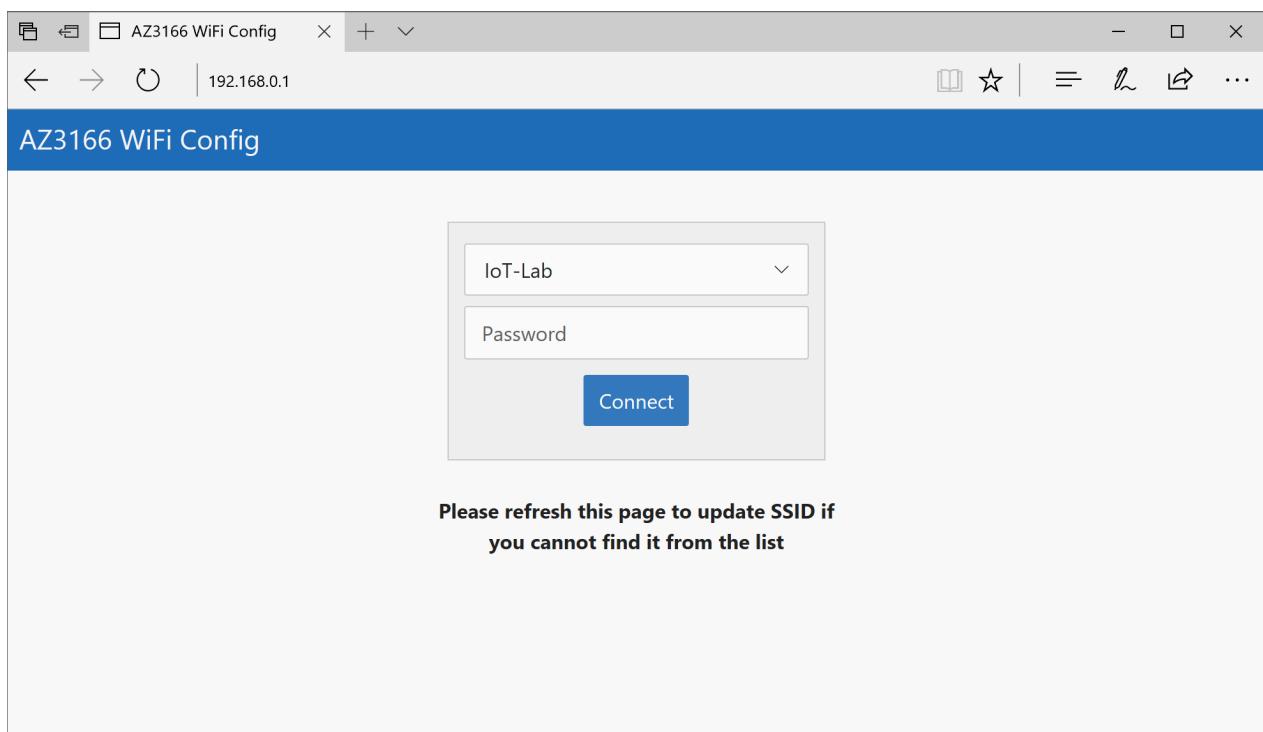
Connect to DevKit AP

Now, use another Wi-Fi enabled device (computer or mobile phone) to connect to the DevKit SSID (highlighted in the previous image). Leave the password empty.

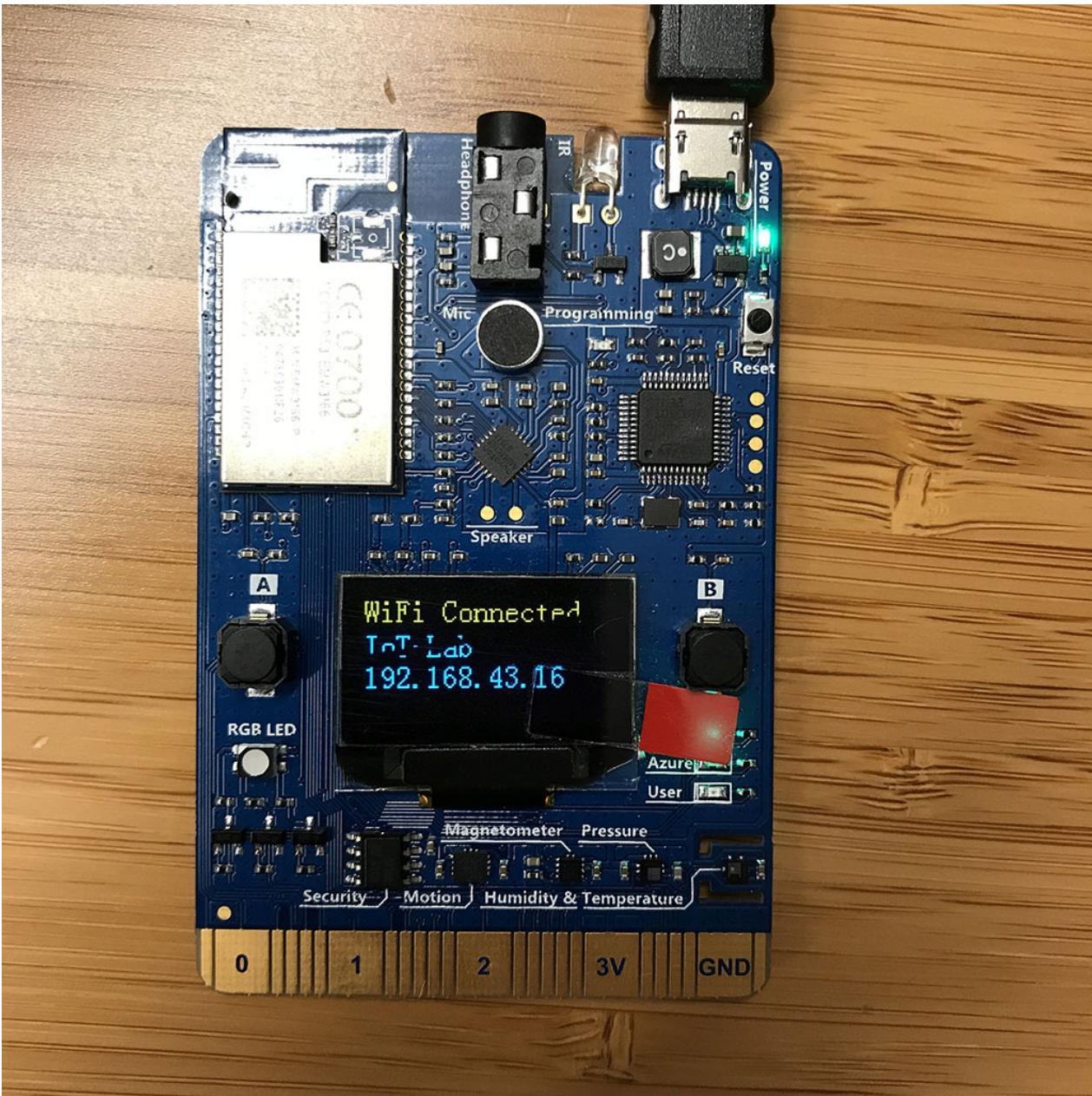


Configure Wi-Fi for the DevKit

Open the IP address shown on the DevKit screen on your computer or mobile phone browser, select the Wi-Fi network that you want the DevKit to connect to, and then type the password. Select **Connect**.



When the connection succeeds, the DevKit reboots in a few seconds. You then see the Wi-Fi name and IP address on the screen:



NOTE

You will need a 2.4GHz network for IoT DevKit working. The WiFi module on the IoT DevKit is not compatible with 5GHz network. Check [FAQ](#) for more details.

After Wi-Fi is configured, your credentials will persist on the device for that connection, even if the device is unplugged. For example, if you configure the DevKit for Wi-Fi in your home and then take the DevKit to the office, you will need to reconfigure AP mode (starting at the step in the "Enter AP Mode" section) to connect the DevKit to your office Wi-Fi.

Start using the DevKit

The default app running on the DevKit checks the latest version of the firmware and displays some sensor diagnosis data for you.

Upgrade to the latest firmware

NOTE

Since v1.1, DevKit enables ST-SAFE in bootloader. You need to upgrade the firmware if you are running a version prior to v1.1.

If you need a firmware upgrade, the screen will show the current and latest firmware versions. To upgrade, follow the [Upgrade firmware](#) guide.

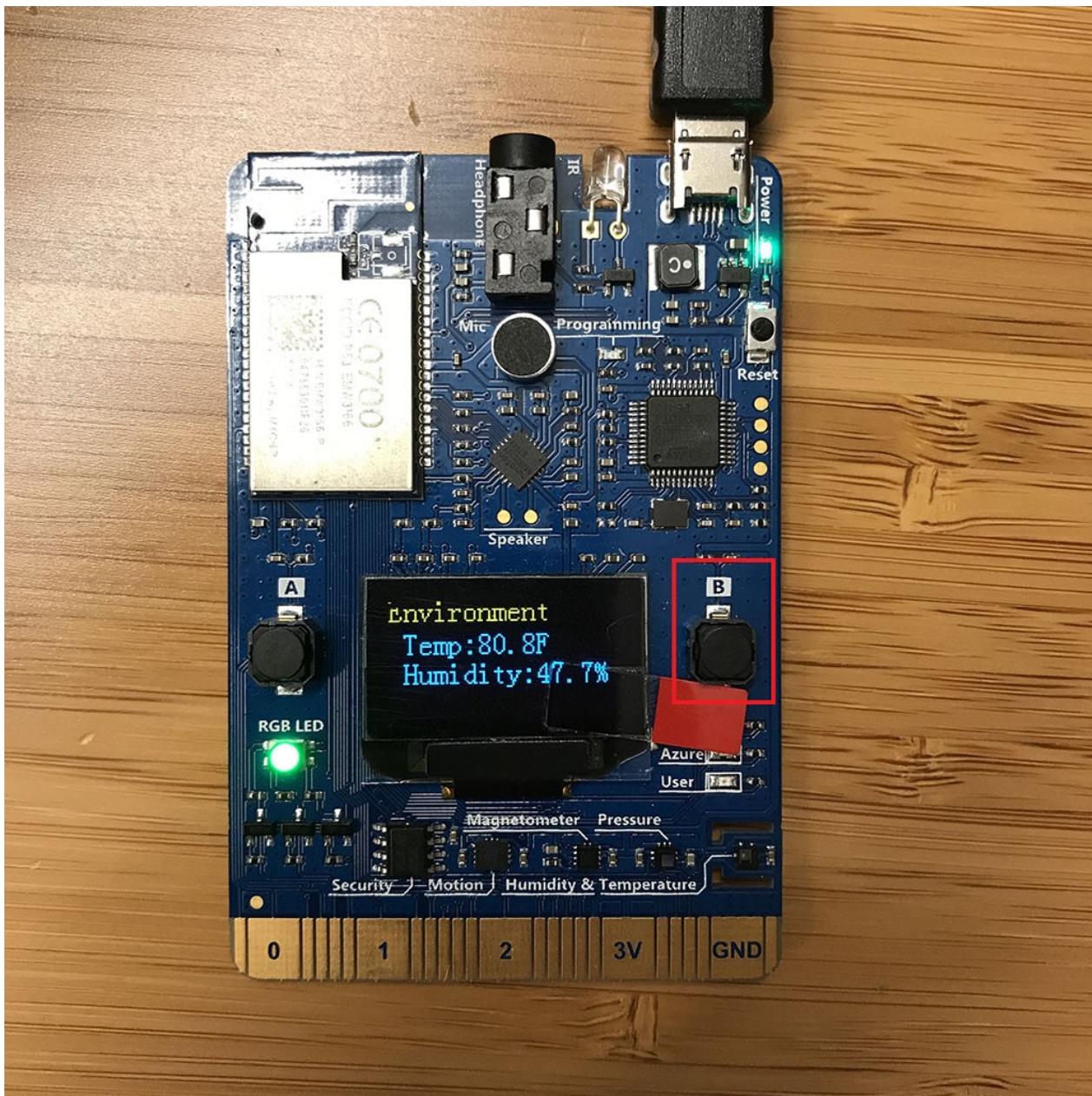


NOTE

This is a one-time effort. After you start developing on the DevKit and upload your app, the latest firmware will come with your app.

Test various sensors

Press button B to test the sensors. Continue pressing and releasing the button B to cycle through each sensor.



Prepare the development environment

Install Azure IoT Tools

We recommend [Azure IoT Tools](#) extension pack for Visual Studio Code to develop on the DevKit. The Azure IoT Tools contains [Azure IoT Device Workbench](#) to develop and debug on various IoT devkit devices and [Azure IoT Hub Toolkit](#) to manage and interact with Azure IoT Hub.

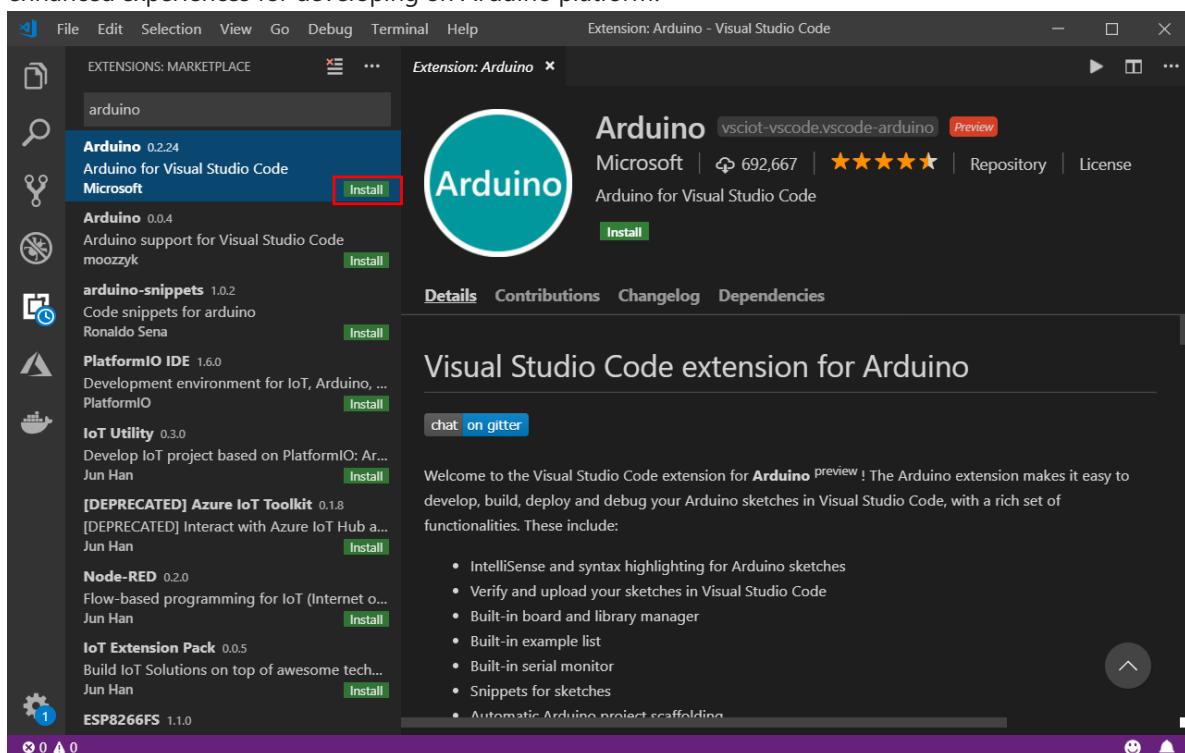
You can watch these [Channel 9](#) videos to have overview about what they do:

- [Introduction to the new IoT Workbench extension for VS Code](#)
- [What's new in the IoT Toolkit extension for VS Code](#)

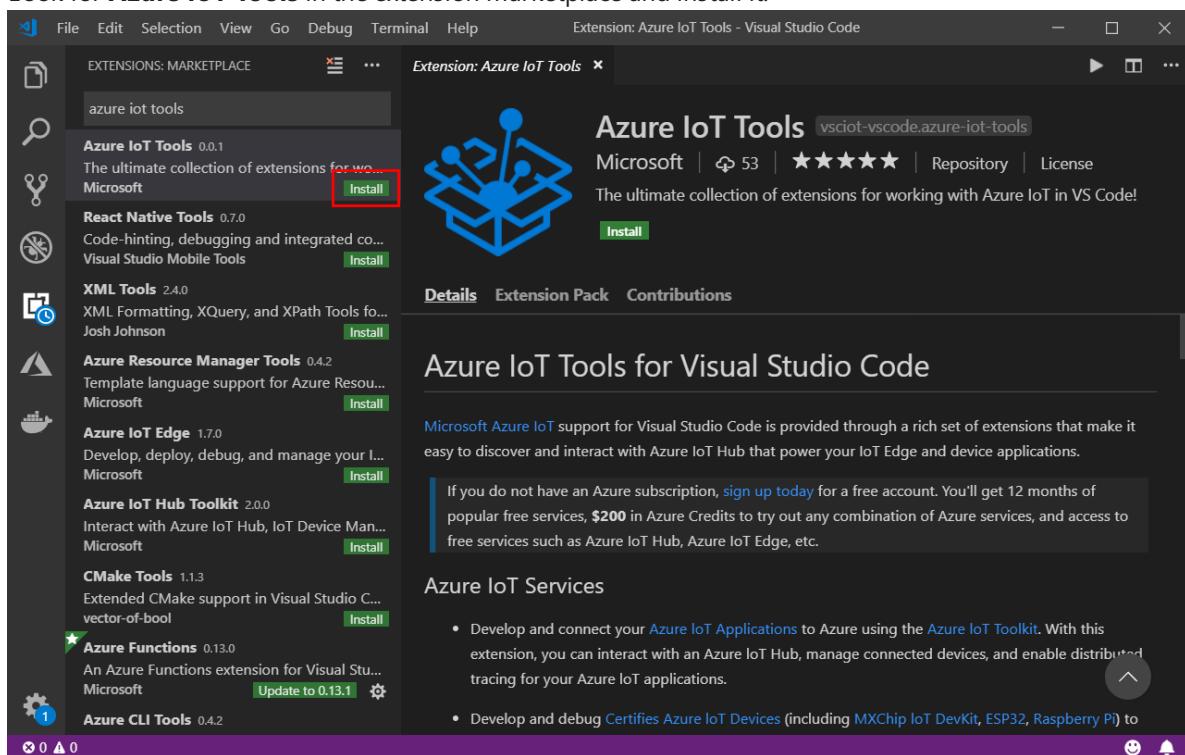
Follow these steps to prepare the development environment for DevKit:

1. Install [Arduino IDE](#). It provides the necessary toolchain for compiling and uploading Arduino code.
 - **Windows:** Use Windows Installer version. Do not install from the app store.
 - **macOS:** Drag and drop the extracted **Arduino.app** into `/Applications` folder.
 - **Ubuntu:** Unzip it into folder such as `$HOME/Downloads/arduino-1.8.8`
2. Install [Visual Studio Code](#), a cross platform source code editor with powerful developer tooling, like IntelliSense code completion and debugging.

3. Launch VS Code, look for **Arduino** in the extension marketplace and install it. This extension provides enhanced experiences for developing on Arduino platform.

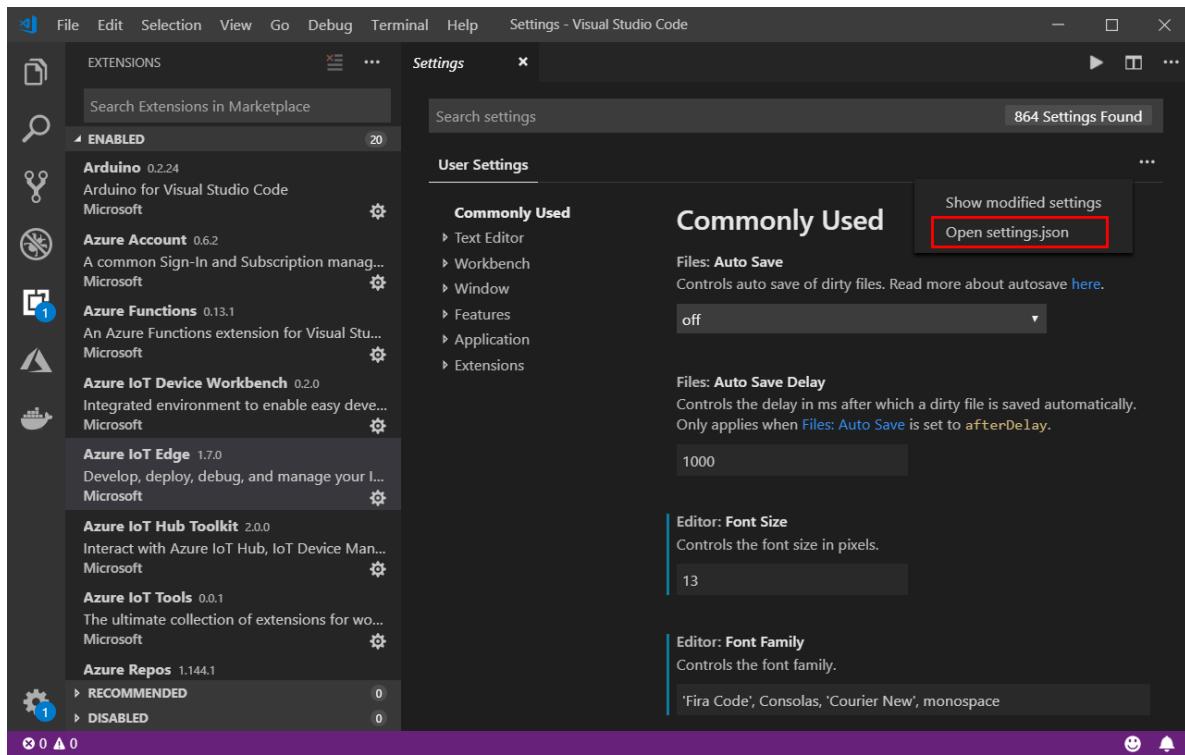


4. Look for **Azure IoT Tools** in the extension marketplace and install it.



5. Configure VS Code with Arduino settings.

In Visual Studio Code, click **File > Preference > Settings**. Then click the ... and **Open settings.json**.



Add following lines to configure Arduino depending on your platform:

- **Windows:**

```
"arduino.path": "C:\\\\Program Files (x86)\\\\Arduino",
"arduino.additionalUrls":
"https://raw.githubusercontent.com/VSChina/azureiotdevkit_tools/master/package_azureboard_index.json"
```

- **macOS:**

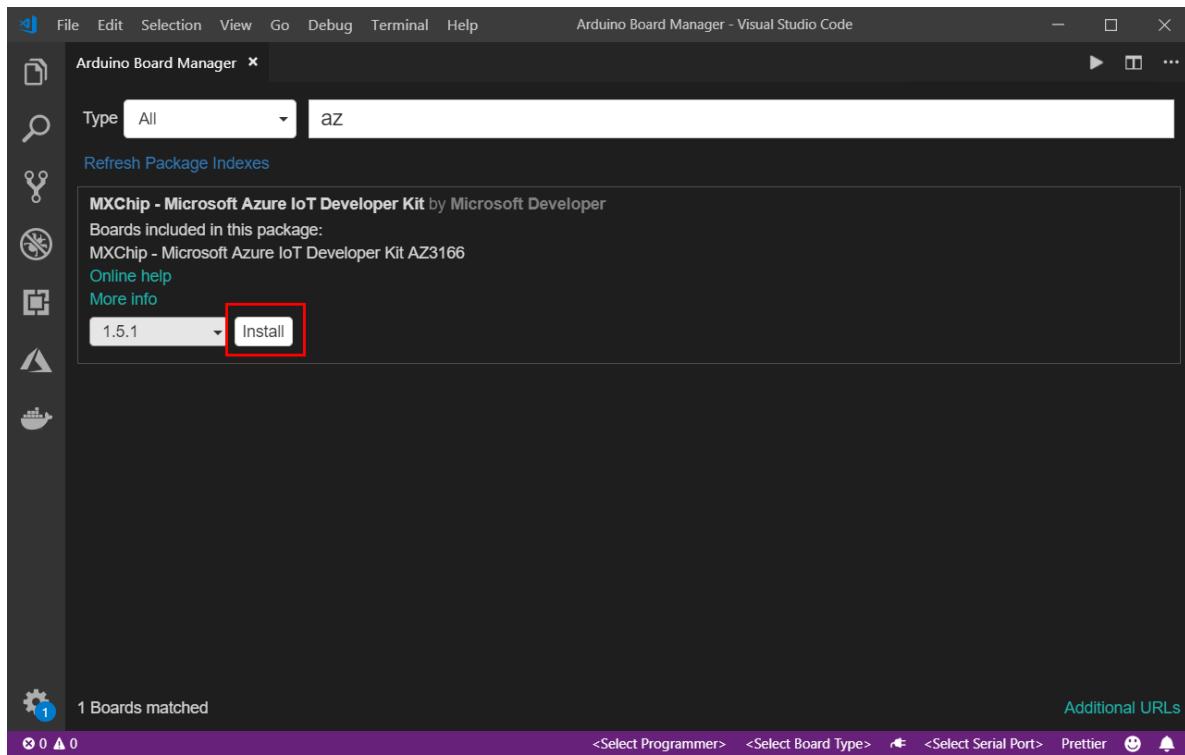
```
"arduino.path": "/Applications",
"arduino.additionalUrls":
"https://raw.githubusercontent.com/VSChina/azureiotdevkit_tools/master/package_azureboard_index.json"
```

- **Ubuntu:**

Replace the **{username}** placeholder below with your username.

```
"arduino.path": "/home/{username}/Downloads/arduino-1.8.8",
"arduino.additionalUrls":
"https://raw.githubusercontent.com/VSChina/azureiotdevkit_tools/master/package_azureboard_index.json"
```

6. Click **F1** to open the command palette, type and select **Arduino: Board Manager**. Search for **AZ3166** and install the latest version.



Install ST-Link drivers

ST-Link/V2 is the USB interface that IoT DevKit uses to communicate with your development machine. You need to install it on Windows to enable flash the compiled device code to the DevKit. Follow the OS-specific steps to allow the machine access to your device.

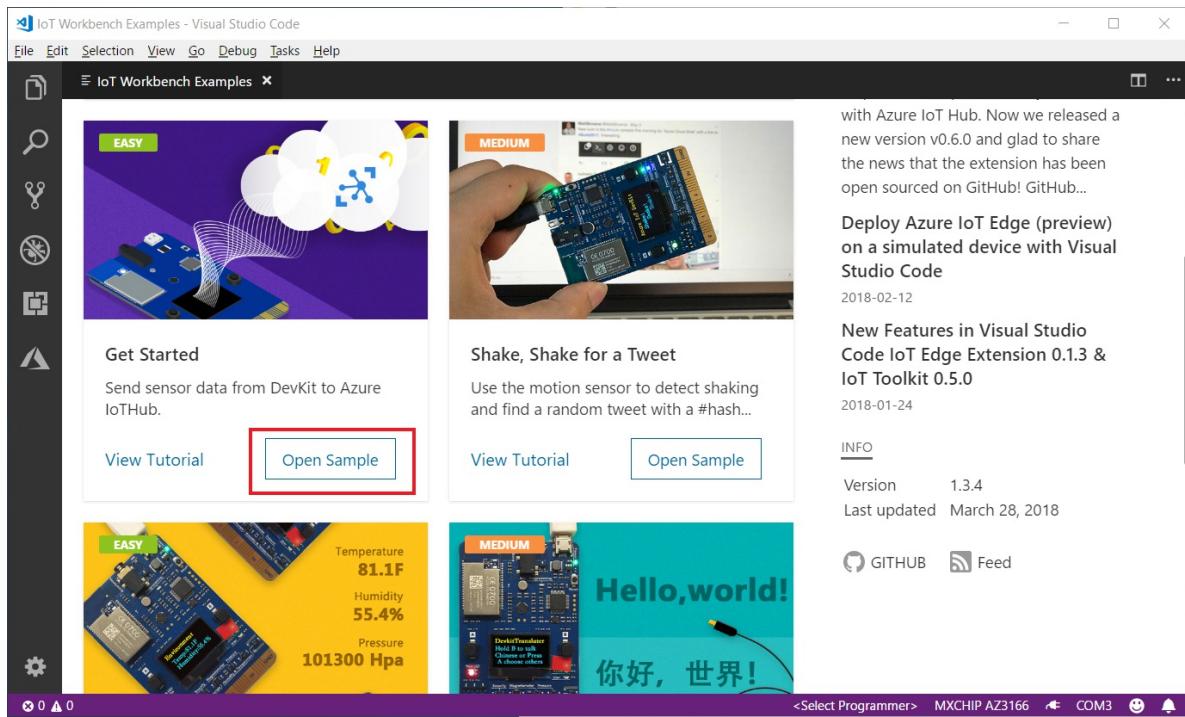
- **Windows:** Download and install USB driver from [STMicroelectronics website](#).
- **macOS:** No driver is required for macOS.
- **Ubuntu:** Run the following in terminal and log out and log in for the group change to take effect:

```
# Copy the default rules. This grants permission to the group 'plugdev'  
sudo cp ~/.arduino15/packages/AZ3166/tools/openocd/0.10.0/linux/contrib/60-openocd.rules  
/etc/udev/rules.d/  
sudo udevadm control --reload-rules  
  
# Add yourself to the group 'plugdev'  
# Logout and log back in for the group to take effect  
sudo usermod -a -G plugdev $(whoami)
```

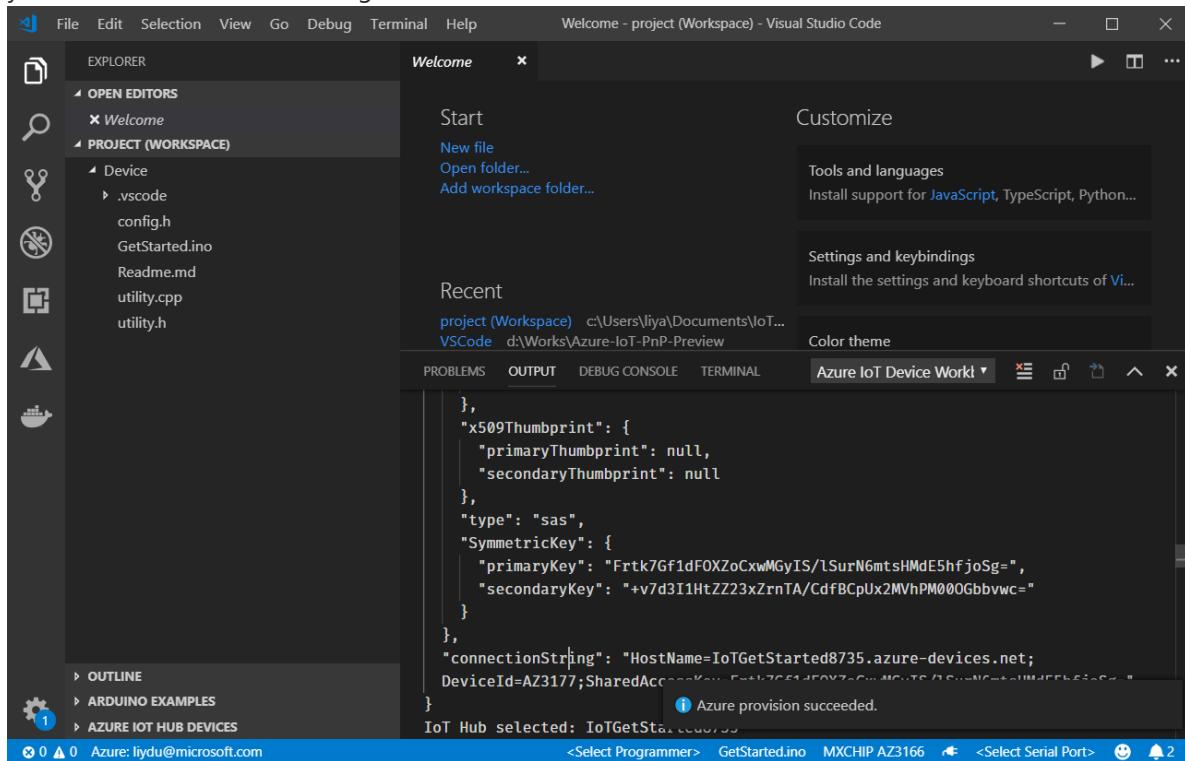
Now you are all set with preparing and configuring your development environment. Let us build the "Hello World" sample for IoT: sending temperature telemetry data to Azure IoT Hub.

Build your first project

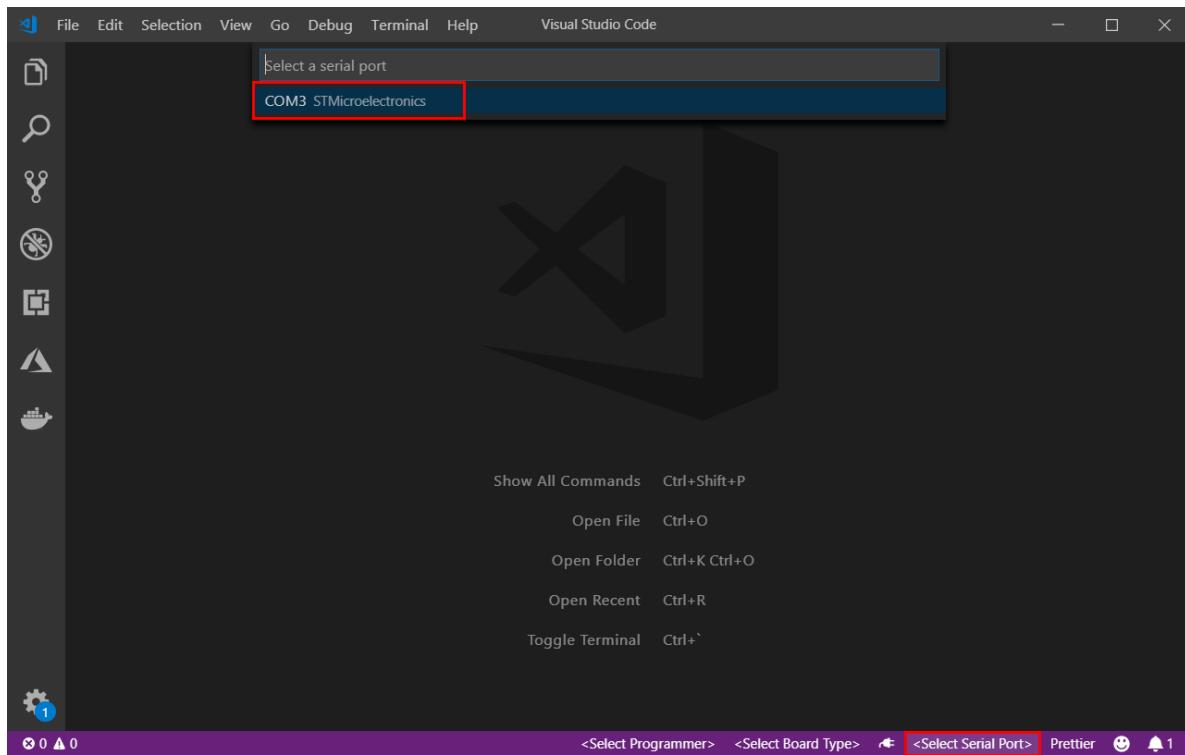
1. Make sure your IoT DevKit is **not connected** to your computer. Start VS Code first, and then connect the DevKit to your computer.
2. Click **F1** to open the command palette, type and select **Azure IoT Device Workbench: Open Examples....** Then select **IoT DevKit** as board.
3. In the IoT Workbench Examples page, find **Get Started** and click **Open Sample**. Then selects the default path to download the sample code.



4. In the new opened project window, click **F1** to open the command palette, type and select **Azure IoT Device Workbench: Provision Azure Services....** Follow the step by step guide to finish provisioning your Azure IoT Hub and creating the IoT Hub device.

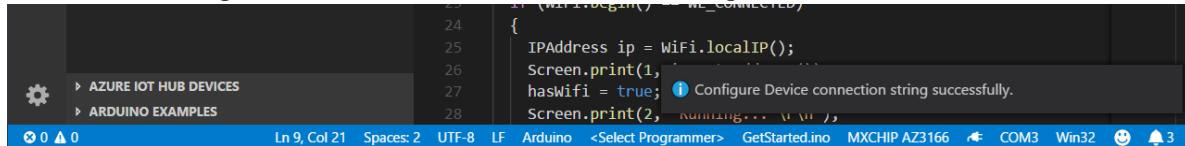


5. In the bottom-right status bar, check the **MXCHIP AZ3166** is shown as selected board and serial port with **STMicroelectronics** is used.

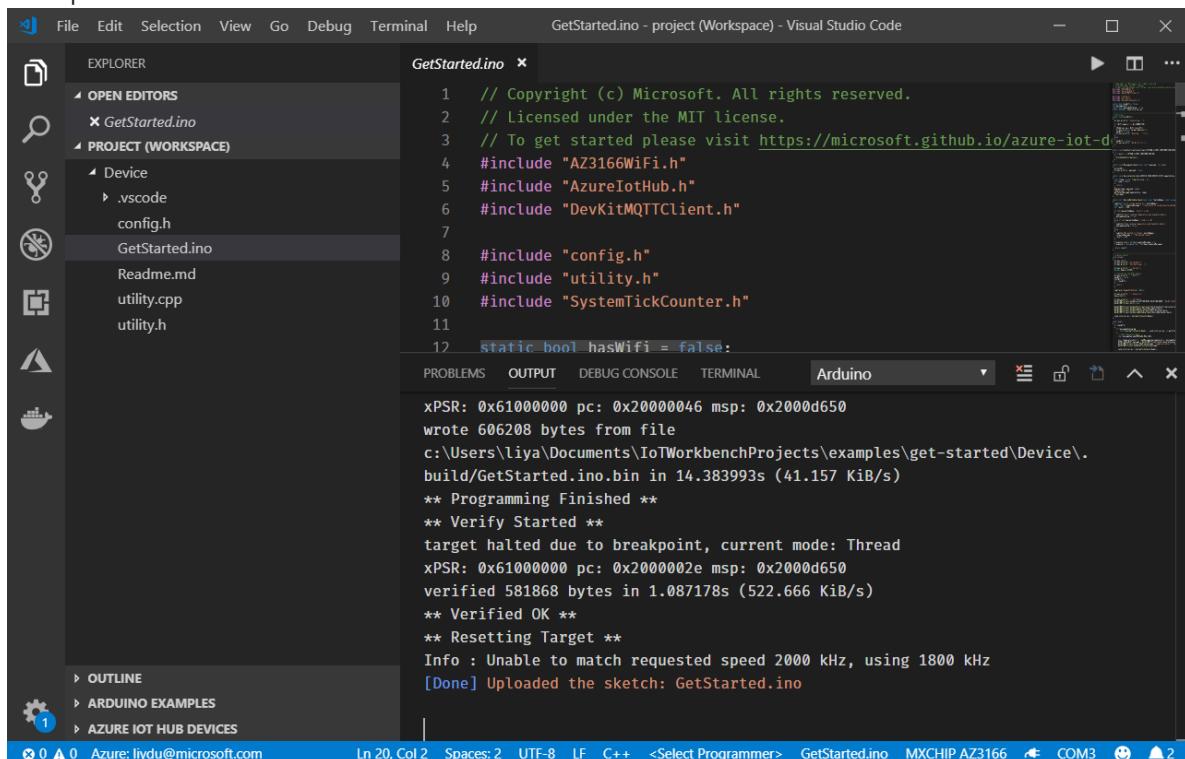


6. Click **F1** to open the command palette, type and select **Azure IoT Device Workbench: Configure Device Settings...**, then select **Config Device Connection String > Select IoT Hub Device Connection String**.

7. On DevKit, hold down **button A**, push and release the **reset** button, and then release **button A**. Your DevKit enters configuration mode and saves the connection string.



8. Click **F1** again, type and select **Azure IoT Device Workbench: Upload Device Code**. It starts compile and upload the code to DevKit.



The DevKit reboots and starts running the code.

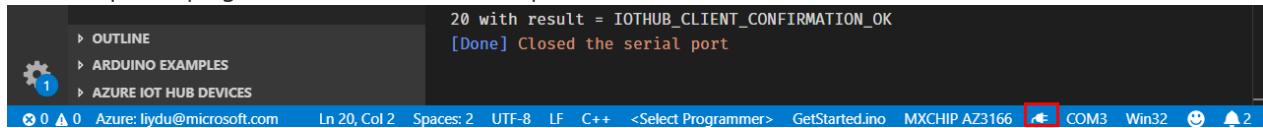
NOTE

If there is any errors or interruptions, you can always recover by running the command again.

Test the project

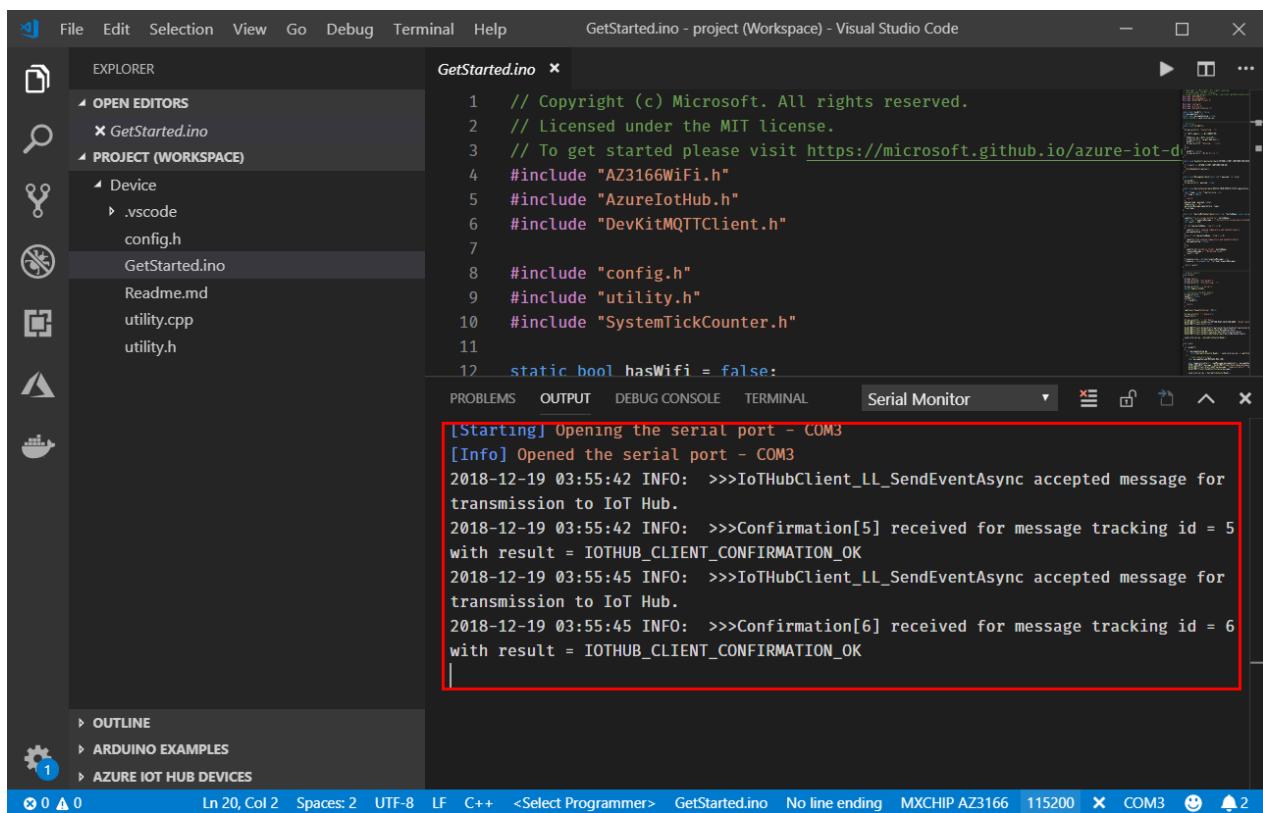
View the telemetry sent to Azure IoT Hub

Click the power plug icon on the status bar to open the Serial Monitor:



The sample application is running successfully when you see the following results:

- The Serial Monitor displays the message sent to the IoT Hub.
- The LED on the MXChip IoT DevKit is blinking.



View the telemetry received by Azure IoT Hub

You can use [Azure IoT Tools](#) to monitor device-to-cloud (D2C) messages in IoT Hub.

1. Sign in [Azure portal](#), find the IoT Hub you created.

2. In the **Shared access policies** pane, click the **iothubowner policy**, and write down the Connection string of your IoT hub.

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

3. In VS Code, click **F1**, type and select **Azure IoT Hub: Set IoT Hub Connection String**. Copy the connection string into it.

```
// To get started please visit https://microsoft.github.io/azure-iot-device-c/
#include "AZ3166WiFi.h"
#include "AzureIotHub.h"
#include "DevKitMQTTClient.h"
```

4. Expand the **AZURE IOT HUB DEVICES** pane on the right, right click on the device name you created and select **Start Monitoring D2C Message**.

The screenshot shows the Visual Studio Code interface with the 'GetStarted.ino - project (Workspace)' tab open. In the center-right area, there is a code editor window displaying C++ code for an Arduino project. A context menu is open over the code, with the 'Start Monitoring D2C Message' option highlighted by a red rectangle. The menu also includes other options like 'Generate Code', 'Send D2C Message to IoT Hub', 'Send C2D Message to Device', etc.

5. In **OUTPUT** pane, you can see the incoming D2C messages to the IoT Hub.

The screenshot shows the Visual Studio Code interface with the 'GetStarted.ino - project (Workspace)' tab open. The 'OUTPUT' pane at the bottom right is highlighted with a red rectangle and displays the following JSON message:
"applicationProperties": {
 "temperatureAlert": "false"
}
}
[IoTHubMonitor] [12:06:55 PM] Message received from [AZ3166]:
{
 "body": {
 "messageId": 244,
 "humidity": 33.29999923706055
 },
 "applicationProperties": {
 "temperatureAlert": "false"
 }
}

Problems and feedback

If you encounter problems, you can check for a solution in the [IoT DevKit FAQ](#) or reach out to us from [Gitter](#). You can also give us feedback by leaving a comment on this page.

Next steps

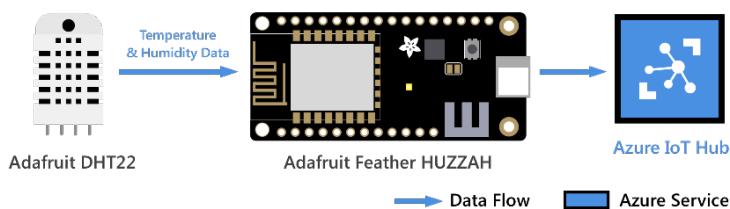
You have successfully connected an MXChip IoT DevKit to your IoT hub, and you have sent the captured sensor data to your IoT hub.

To continue to get started with Azure IoT Hub and to explore other IoT scenarios using IoT DevKit, see the following:

- Connect IoT DevKit to your Azure IoT Central application
- Connect IoT DevKit to Azure IoT Remote Monitoring solution accelerator
- Translate voice message with Azure Cognitive Services
- Retrieve a Twitter message with Azure Functions
- Send messages to an MQTT server using Eclipse Paho APIs
- Monitor the magnetic sensor and send email notifications with Azure Functions

Connect Adafruit Feather HUZZAH ESP8266 to Azure IoT Hub in the cloud

3/6/2019 • 10 minutes to read



What you do

Connect Adafruit Feather HUZZAH ESP8266 to an IoT hub that you create. Then you run a sample application on ESP8266 to collect the temperature and humidity data from a DHT22 sensor. Finally, you send the sensor data to your IoT hub.

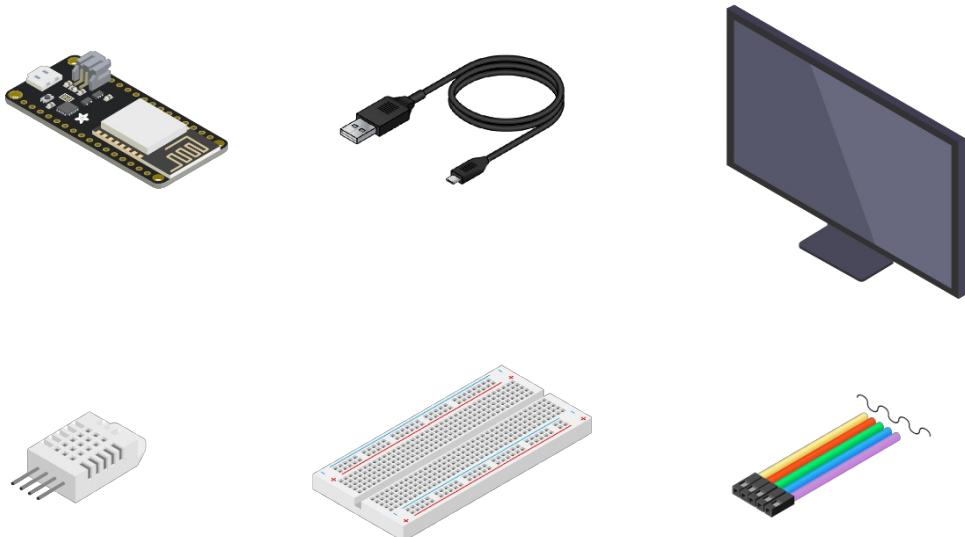
NOTE

If you're using other ESP8266 boards, you can still follow these steps to connect it to your IoT hub. Depending on the ESP8266 board you're using, you might need to reconfigure the `LED_PIN`. For example, if you're using ESP8266 from AI-Thinker, you might change it from `0` to `2`. Don't have a kit yet? Get it from the [Azure website](#).

What you learn

- How to create an IoT hub and register a device for Feather HUZZAH ESP8266
- How to connect Feather HUZZAH ESP8266 with the sensor and your computer
- How to collect sensor data by running a sample application on Feather HUZZAH ESP8266
- How to send the sensor data to your IoT hub

What you need



To complete this operation, you need the following parts from your Feather HUZZAH ESP8266 Starter Kit:

- The Feather HUZZAH ESP8266 board
- A Micro USB to Type A USB cable

You also need the following things for your development environment:

- An active Azure subscription. If you don't have an Azure account, [create a free Azure trial account](#) in just a few minutes.
- A Mac or PC that is running Windows or Ubuntu.
- A wireless network for Feather HUZZAH ESP8266 to connect to.
- An Internet connection to download the configuration tool.
- [Visual Studio Code extension for Arduino](#).

NOTE

The Arduino IDE version used by Visual Studio Code extension for Arduino has to be version 1.6.8 or later. Earlier versions don't work with the AzureIoT library.

The following items are optional in case you don't have a sensor. You also have the option of using simulated sensor data.

- An Adafruit DHT22 temperature and humidity sensor
- A breadboard
- M/M jumper wires

Create an IoT hub

This section describes how to create an IoT hub using the [Azure portal](#).

1. Log in to the [Azure portal](#).
2. Choose **+Create a resource**, then choose **Internet of Things**.
3. Click **IoT Hub** from the list on the right. You see the first screen for creating an IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Review + create

Create an IoT Hub to help you connect, monitor, and manage billions of your IoT assets. [Learn More](#)

PROJECT DETAILS

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

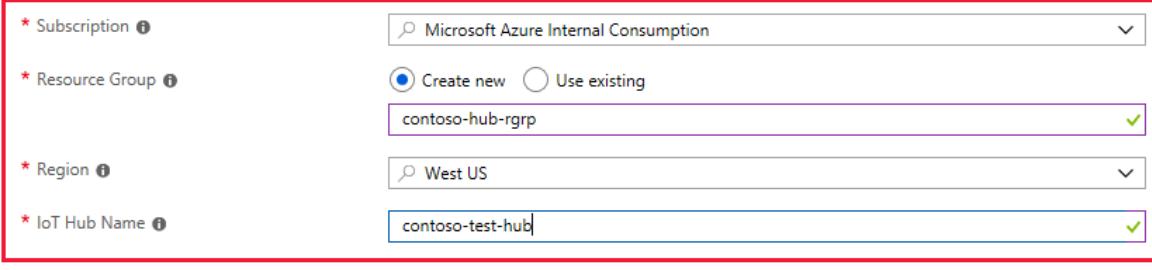
* Subscription Create new Use existing
contoso-hub-rgrp ✓

* Resource Group Create new Use existing
contoso-hub-rgrp ✓

* Region Create new Use existing
contoso-test-hub ✓

* IoT Hub Name Create new Use existing
contoso-test-hub ✓

[Review + create](#) [Next: Size and scale »](#) [Automation options](#)



Fill in the fields.

Subscription: Select the subscription to use for your IoT hub.

Resource Group: You can create a new resource group or use an existing one. To create a new one, click **Create new** and fill in the name you want to use. To use an existing resource group, click **Use existing** and select the resource group from the dropdown list. For more information, see [Manage Azure Resource Manager resource groups](#).

Region: This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.

IoT Hub Name: Put in the name for your IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

4. Click **Next: Size and scale** to continue creating your IoT hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics **Size and scale** Review + create

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS

* Pricing and scale tier [?](#) S1: Standard tier [?](#) [Learn how to choose the right IoT Hub tier for your solution](#)

Number of S1 IoT Hub units [?](#) [1](#) This determines your IoT Hub scale capability and can be changed as your need increases.

Device-to-cloud-messages [?](#) Enabled

Message routing [?](#) Enabled

Cloud-to-device commands [?](#) Enabled

IoT Edge [?](#) Enabled

Device management [?](#) Enabled

Advanced Settings

Device-to-cloud partitions [?](#) [4](#)

[Review + create](#) [« Previous: Basics](#) [Automation options](#)

On this screen, you can take the defaults and just click **Review + create** at the bottom.

Pricing and scale tier: You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free tier is intended for testing and evaluation. It allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the free tier.

IoT Hub units: The number of messages allowed per unit per day depends on your hub's pricing tier. For example, if you want the IoT hub to support ingress of 700,000 messages, you choose two S1 tier units.

For details about the other tier options, see [Choosing the right IoT Hub tier](#).

Advanced / Device-to-cloud partitions: This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most IoT hubs only need four partitions.

5. Click **Review + create** to review your choices. You see something similar to this screen.

Basics **Size and scale** **Review + create**

BASICS

Subscription i	Microsoft Azure Internal Consumption
Resource Group i	contoso-hub-rgrp
Region i	West US
IoT Hub Name i	contoso-test-hub

SIZE AND SCALE

Pricing and scale tier i	S1
Number of S1 IoT Hub units i	1
Messages per day i	400,000
Cost per month	25.00 USD

Create « Previous: Size and scale Automation options

- Click **Create** to create your new IoT hub. Creating the hub takes a few minutes.

Retrieve connection string for IoT hub

After your hub has been created, retrieve the connection string for the hub. This is used to connect devices and applications to your hub.

- Click on your hub to see the IoT Hub pane with Settings, and so on. Click **Shared access policies**.
- In **Shared access policies**, select the **iothubowner** policy.
- Under **Shared access keys**, copy the **Connection string -- primary key** to be used later.

ContosoHub - Shared access policies

iothubowner

Access policy name: iothubowner

Permissions:

- Registry read [i](#)
- Registry write [i](#)
- Service connect [i](#)
- Device connect [i](#)

Shared access keys:

Primary key:	<code>HostName=ContosoHub.azure-devices.net;SharedAccessKey...</code>	
Secondary key:	<code>HostName=ContosoHub.azure-devices.net;SharedAccessKey...</code>	

For more information, see [Access control](#) in the "IoT Hub developer guide."

Register a new device in the IoT hub

In this section, you create a device identity in the identity registry in your IoT hub. A device cannot connect to IoT hub unless it has an entry in the identity registry. For more information, see the "Identity registry" section of the [IoT Hub developer guide](#)

1. In your IoT hub navigation menu, open **IoT Devices**, then click **Add** to register a new device in your IoT hub.

The screenshot shows the 'ContosoHub - IoT devices' blade in the Azure portal. On the left, there's a navigation menu with items like 'Events', 'Shared access policies', 'Pricing and scale', etc., and a red box highlights the 'IoT devices' item under 'EXPLORERS'. At the top, there's a search bar, a 'Refresh' button, and a 'Delete' button. A large red box highlights the '+ Add' button. Below the buttons, there's a message: 'You can use this tool to view, create, update, and delete devices on your IoT Hub.' There's also a 'Query' section with a query editor containing: 'Query ⓘ SELECT * FROM devices WHERE optional (e.g. tags.location='US')' and an 'Execute' button. At the bottom, there's a table header with columns: DEVICE ID, STATUS, LAST ACTIVITY, LAST STATUS..., AUTHENTICATI..., CLOUD TO DEV... and a row 'No results'.

2. Provide a name for your new device, such as **myDeviceId**, and click **Save**. This action creates a new device identity for your IoT hub.



Create a device

□ X



Learn more about creating devices



* Device ID i

myDeviceId



Authentication type i

Symmetric key X.509 Self-Signed X.509 CA Signed

* Primary key i

Enter your primary key

* Secondary key i

Enter your secondary key

Auto-generate keys i



Connect this device to an IoT hub i

Enable Disable

Parent device (Preview) i

No parent device

[Set a parent device](#)

[Save](#)

IMPORTANT

The device ID may be visible in the logs collected for customer support and troubleshooting, so make sure to avoid any sensitive information while naming it.

3. After the device is created, open the device from the list in the **IoT devices** pane. Copy the **Connection string---primary key** to use later.

The screenshot shows the 'Device details' page for a device named 'myDeviceId'. It includes fields for Device Id, Primary key, Secondary key, Connection string (primary key), and Connection string (secondary key). The 'Connection string (primary key)' field contains the value 'HostName=ContosoHub.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=<Primary Key>'. The 'Connection string (secondary key)' field contains the value 'HostName=ContosoHub.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=<Secondary Key>'. There are also sections for connecting to an IoT hub and setting a parent device.

NOTE

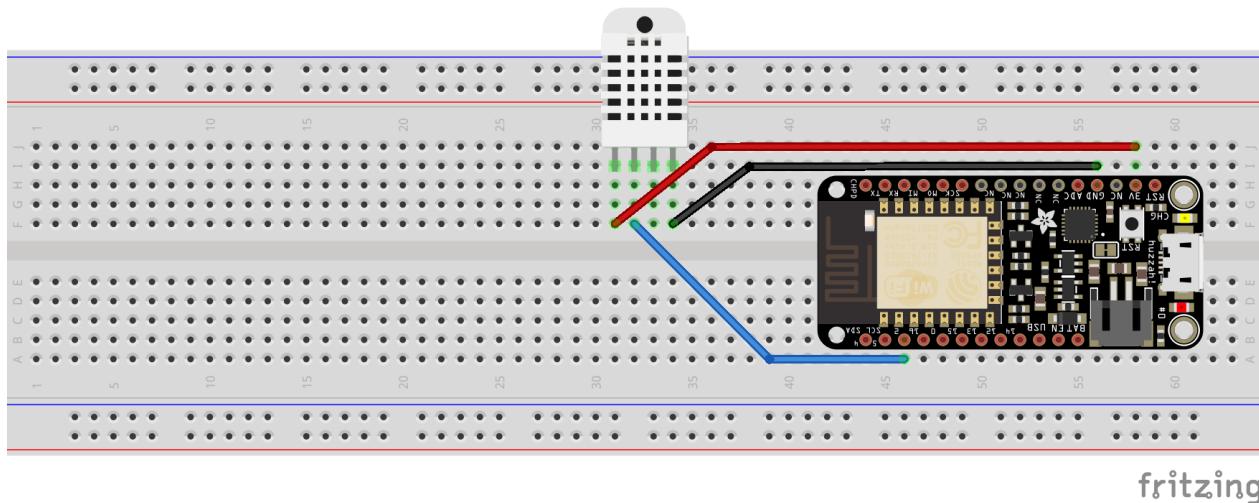
The IoT Hub identity registry only stores device identities to enable secure access to the IoT hub. It stores device IDs and keys to use as security credentials, and an enabled/disabled flag that you can use to disable access for an individual device. If your application needs to store other device-specific metadata, it should use an application-specific store. For more information, see [IoT Hub developer guide](#).

Connect Feather HUZZAH ESP8266 with the sensor and your computer

In this section, you connect the sensors to your board. Then you plug in your device to your computer for further use.

Connect a DHT22 temperature and humidity sensor to Feather HUZZAH ESP8266

Use the breadboard and jumper wires to make the connection as follows. If you don't have a sensor, skip this section because you can use simulated sensor data instead.



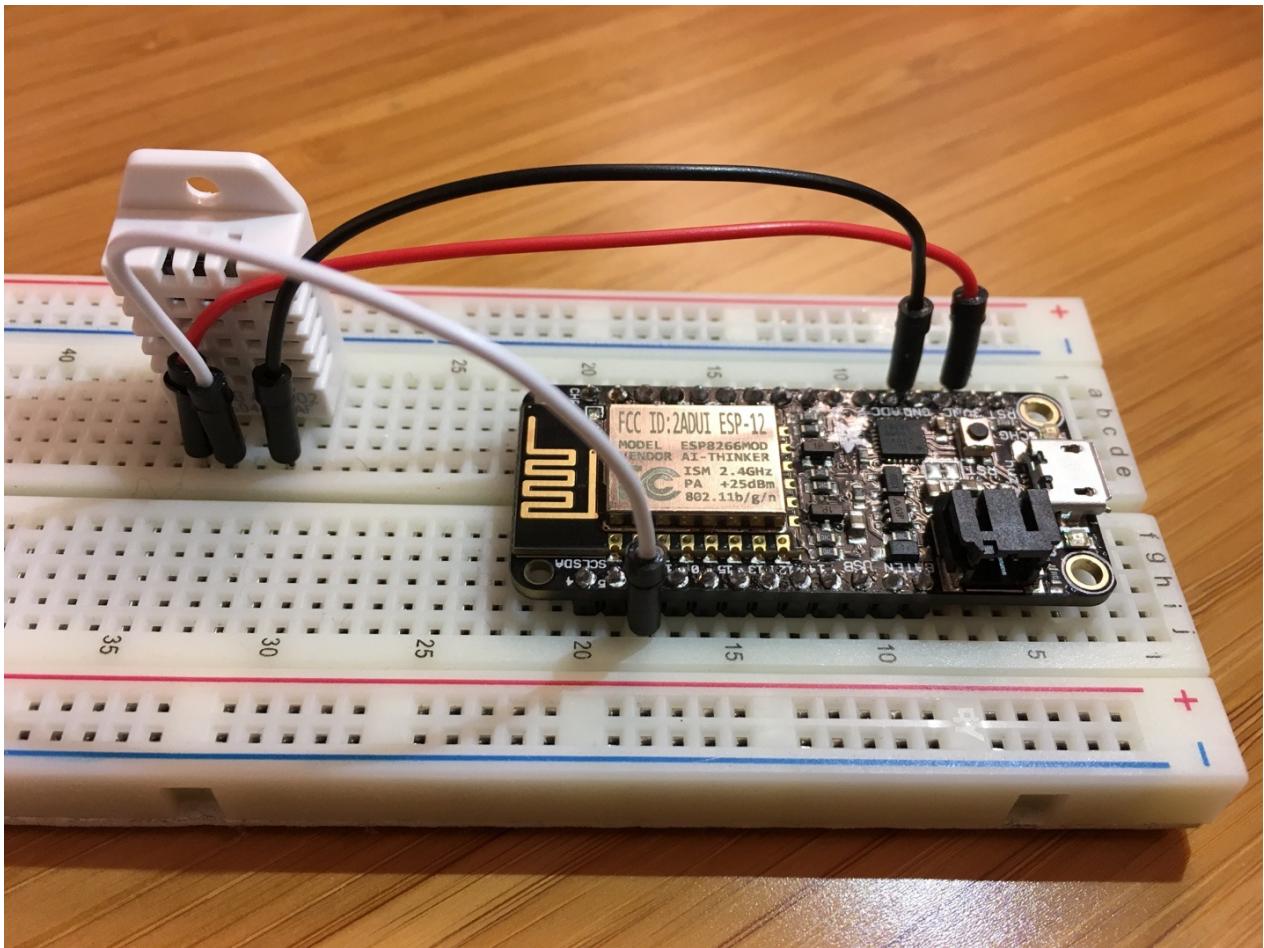
For sensor pins, use the following wiring:

START (SENSOR)	END (BOARD)	CABLE COLOR
VDD (Pin 31F)	3V (Pin 58H)	Red cable
DATA (Pin 32F)	GPIO 2 (Pin 46A)	Blue cable

START (SENSOR)	END (BOARD)	CABLE COLOR
GND (Pin 34F)	GND (Pin 56I)	Black cable

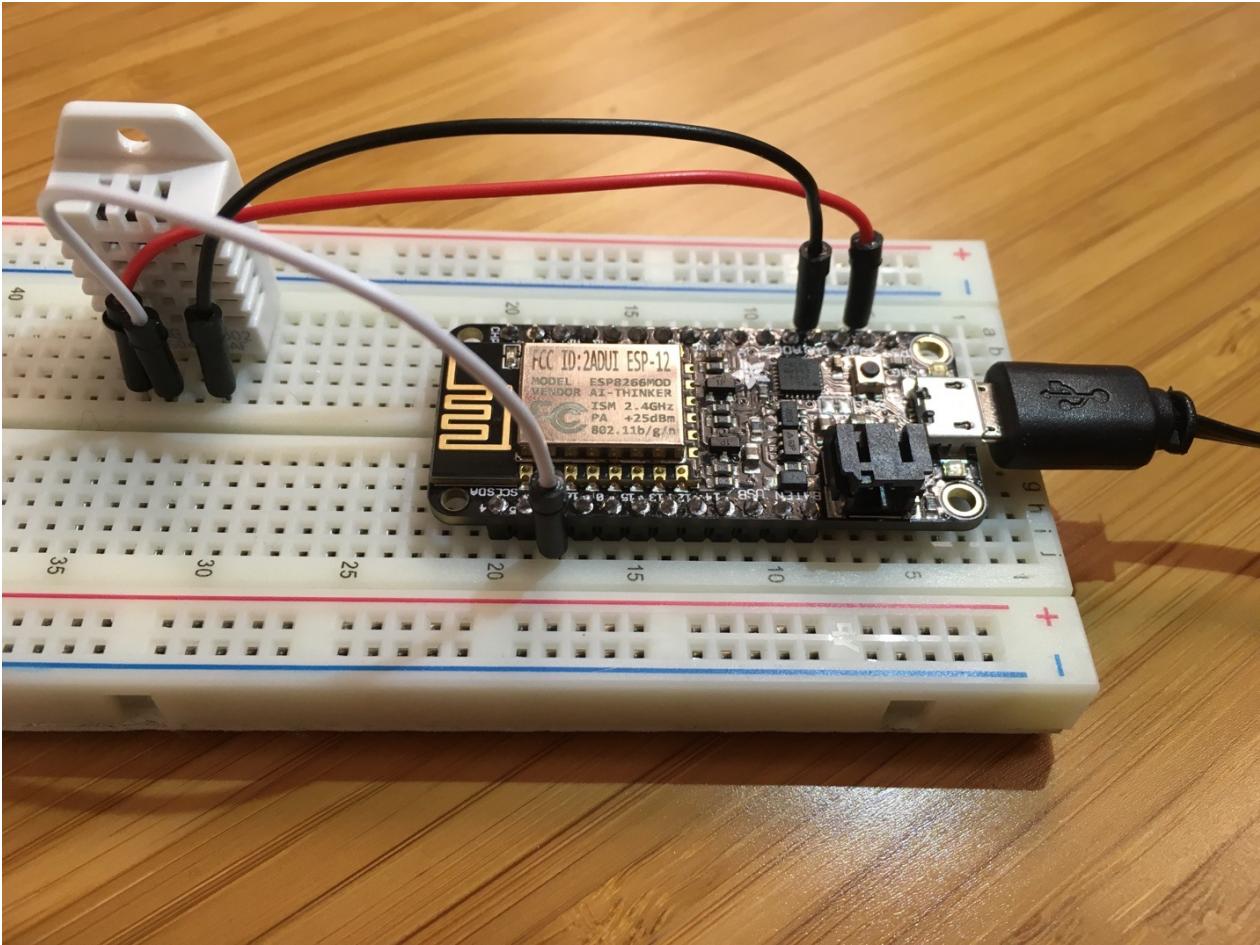
For more information, see [Adafruit DHT22 sensor setup](#) and [Adafruit Feather HUZZAH Esp8266 Pinouts](#).

Now your Feather Huzzah ESP8266 should be connected with a working sensor.



Connect Feather HUZZAH ESP8266 to your computer

As shown next, use the Micro USB to Type A USB cable to connect Feather HUZZAH ESP8266 to your computer.



Add serial port permissions (Ubuntu only)

If you use Ubuntu, make sure you have the permissions to operate on the USB port of Feather HUZZAH ESP8266. To add serial port permissions, follow these steps:

1. Run the following commands at a terminal:

```
ls -l /dev/ttyUSB*
ls -l /dev/ttyACM*
```

You get one of the following outputs:

- crw-rw---- 1 root uucp xxxxxxxx
- crw-rw---- 1 root dialout xxxxxxxx

In the output, notice that `uucp` or `dialout` is the group owner name of the USB port.

2. Add the user to the group by running the following command:

```
sudo usermod -a -G <group-owner-name> <username>
```

`<group-owner-name>` is the group owner name you obtained in the previous step. `<username>` is your Ubuntu user name.

3. Sign out of Ubuntu, and then sign in again for the change to appear.

Collect sensor data and send it to your IoT hub

In this section, you deploy and run a sample application on Feather HUZZAH ESP8266. The sample application blinks the LED on Feather HUZZAH ESP8266, and sends the temperature and humidity data collected from the

DHT22 sensor to your IoT hub.

Get the sample application from GitHub

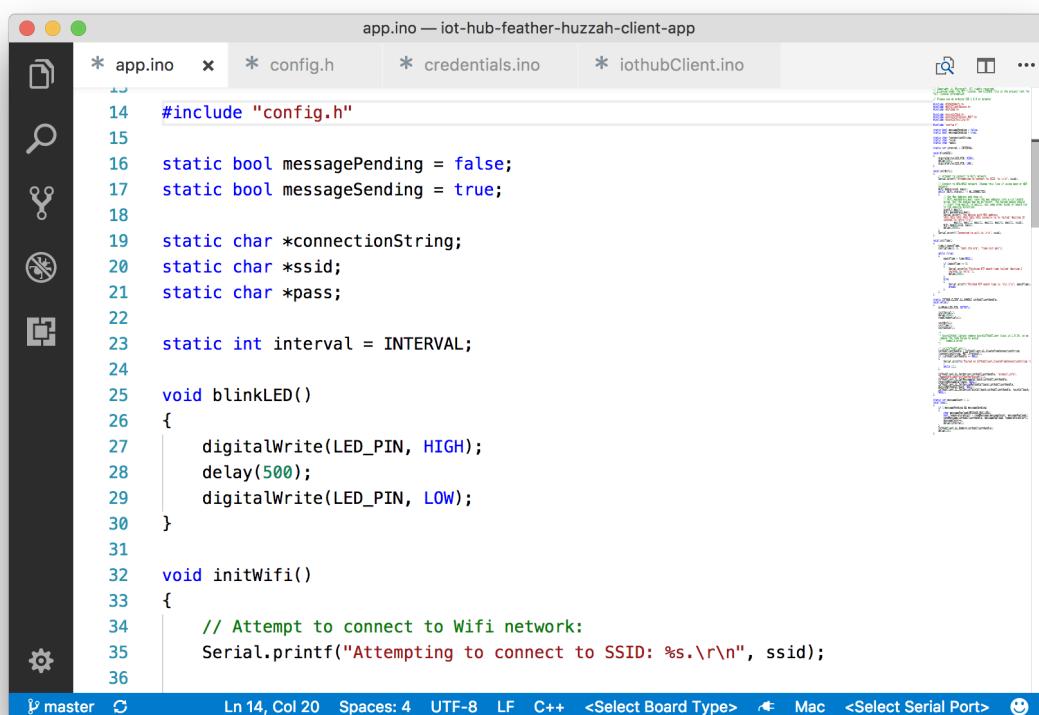
The sample application is hosted on GitHub. Clone the sample repository that contains the sample application from GitHub. To clone the sample repository, follow these steps:

1. Open a command prompt or a terminal window.
2. Go to a folder where you want the sample application to be stored.
3. Run the following command:

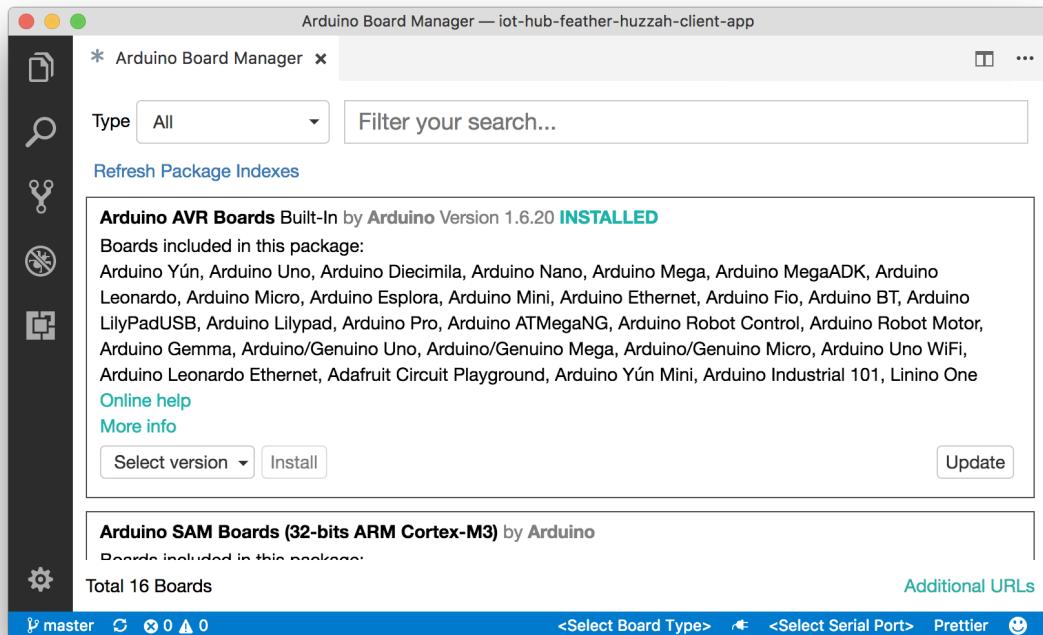
```
git clone https://github.com/Azure-Samples/iot-hub-feather-huzzah-client-app.git
```

Next, install the package for Feather HUZZAH ESP8266 in Visual Studio Code.

4. Open the folder where the sample application is stored.
5. Open the app.ino file in the app folder in the Visual Studio Code.

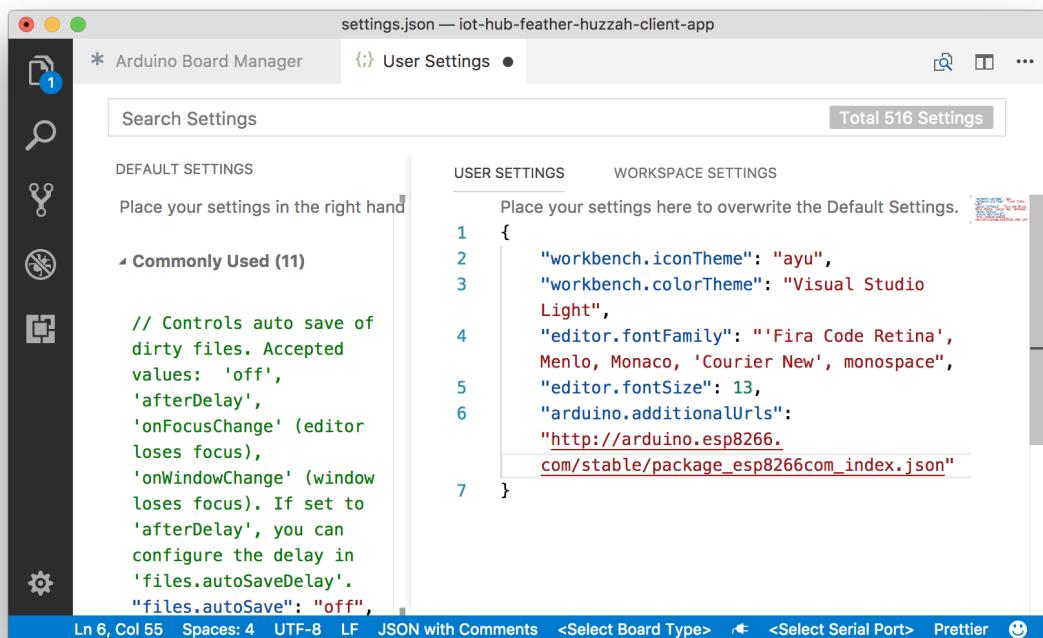


6. In the Visual Studio Code, enter **F1**.
7. Type **Arduino** and select **Arduino: Board Manager**.
8. In the **Arduino Board Manager** tab, click **Additional URLs**.



9. In the **User Settings** window, copy and paste the following at the end of the file

```
"arduino.additionalUrls": "http://arduino.esp8266.com/stable/package_esp8266com_index.json"
```

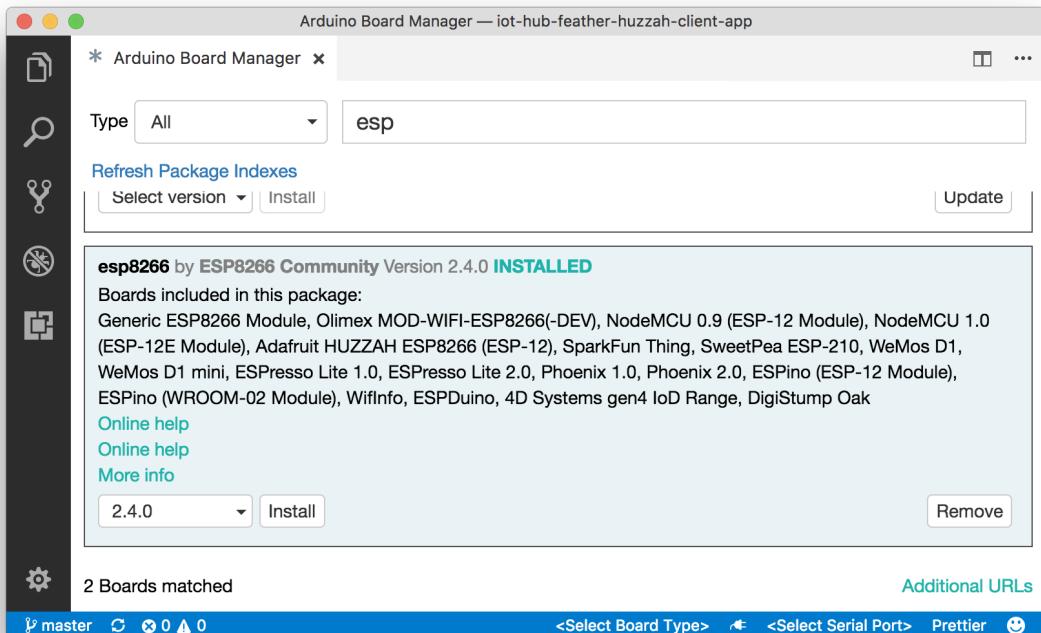


10. Save the file and close the **User Settings** tab.

11. Click **Refresh Package Indexes**. After the refresh finishes, search for **esp8266**.

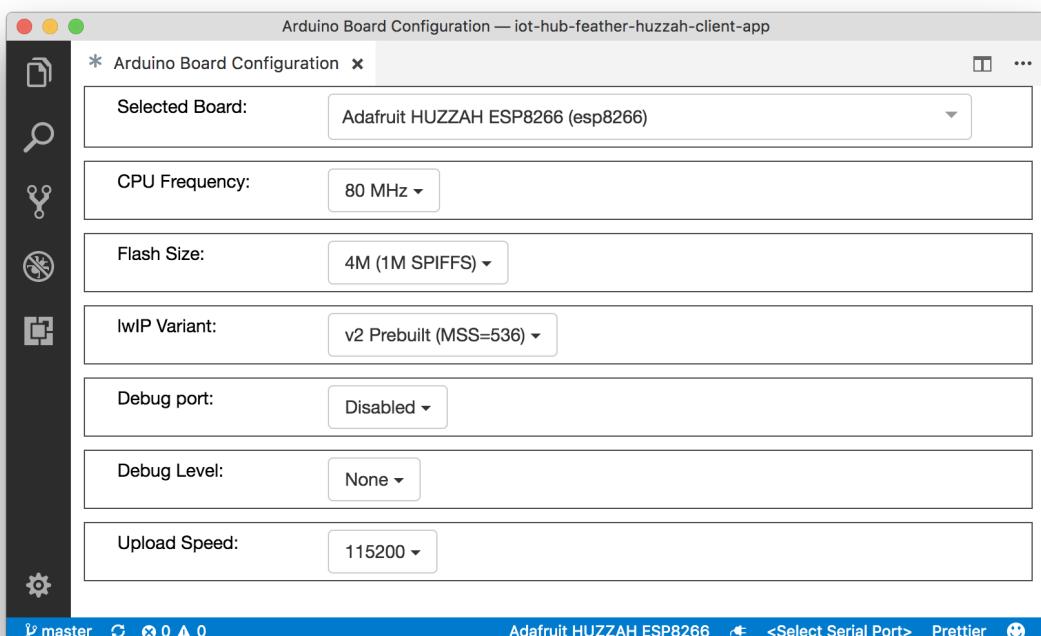
12. Click **Install** button for esp8266.

Boards Manager indicates that ESP8266 with a version of 2.2.0 or later is installed.



13. Enter **F1**, then type **Arduino** and select **Arduino: Board Config**.

14. Click box for **Selected Board:** and type **esp8266**, then select **Adafruit HUZZAH ESP8266 (esp8266)**.



Install necessary libraries

1. In the Visual Studio Code, enter **F1**, then type **Arduino** and select **Arduino: Library Manager**.

2. Search for the following library names one by one. For each library that you find, click **Install**.

- **AzureIoTHub**
- **AzureIoTUtility**
- **AzureIoTProtocol_MQTT**

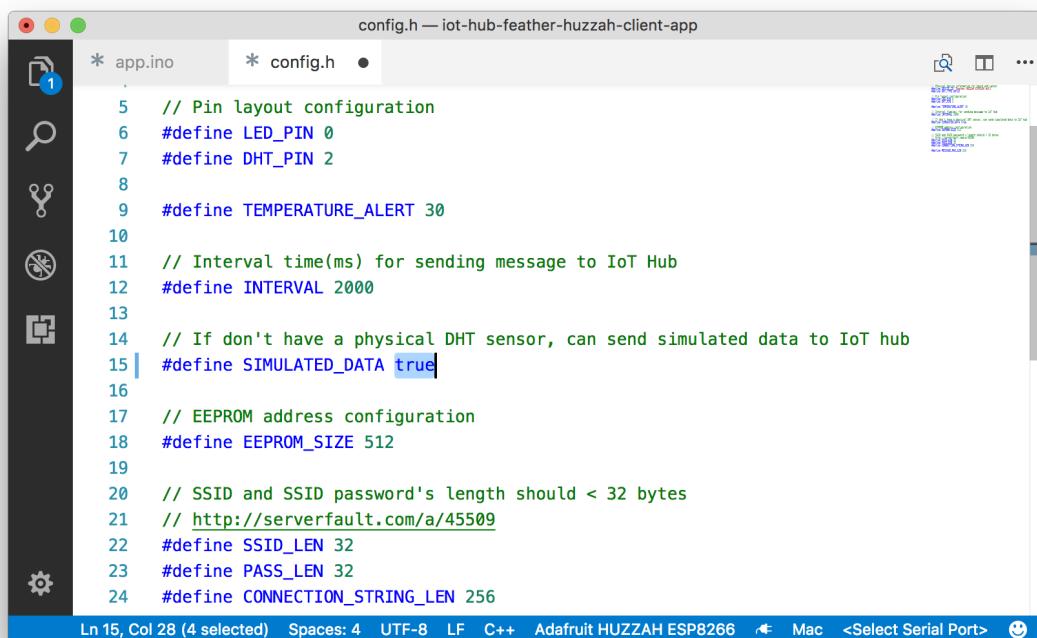
- ArduinoJson
- DHT sensor library
- Adafruit Unified Sensor

Don't have a real DHT22 sensor?

The sample application can simulate temperature and humidity data in case you don't have a real DHT22 sensor. To set up the sample application to use simulated data, follow these steps:

1. Open the `config.h` file in the `app` folder.
2. Locate the following line of code and change the value from `false` to `true`:

```
define SIMULATED_DATA true
```



```
config.h — iot-hub-feather-huzzah-client-app
* app.ino      * config.h •
.
5 // Pin layout configuration
6 #define LED_PIN 0
7 #define DHT_PIN 2
8
9 #define TEMPERATURE_ALERT 30
10
11 // Interval time(ms) for sending message to IoT Hub
12 #define INTERVAL 2000
13
14 // If don't have a physical DHT sensor, can send simulated data to IoT hub
15 #define SIMULATED_DATA true
16
17 // EEPROM address configuration
18 #define EEPROM_SIZE 512
19
20 // SSID and SSID password's length should < 32 bytes
21 // http://serverfault.com/a/45509
22 #define SSID_LEN 32
23 #define PASS_LEN 32
24 #define CONNECTION_STRING_LEN 256

Ln 15, Col 28 (4 selected)  Spaces: 4  UTF-8  LF  C++  Adafruit HUZZAH ESP8266  Mac  <Select Serial Port>  ☺
```

3. Save the file.

Deploy the sample application to Feather HUZZAH ESP8266

1. In the Visual Studio Code, click  on the status bar, and then click the serial port for Feather HUZZAH ESP8266.
2. Enter `F1`, then type **Arduino** and select **Arduino: Upload** to build and deploy the sample application to Feather HUZZAH ESP8266.

Enter your credentials

After the upload completes successfully, follow these steps to enter your credentials:

1. Open Arduino IDE, click **Tools** > **Serial Monitor**.
2. In the serial monitor window, notice the two drop-down lists in the lower-right corner.
3. Select **No line ending** for the left drop-down list.
4. Select **115200 baud** for the right drop-down list.
5. In the input box located at the top of the serial monitor window, enter the following information if you are

asked to provide them, and then click **Send**.

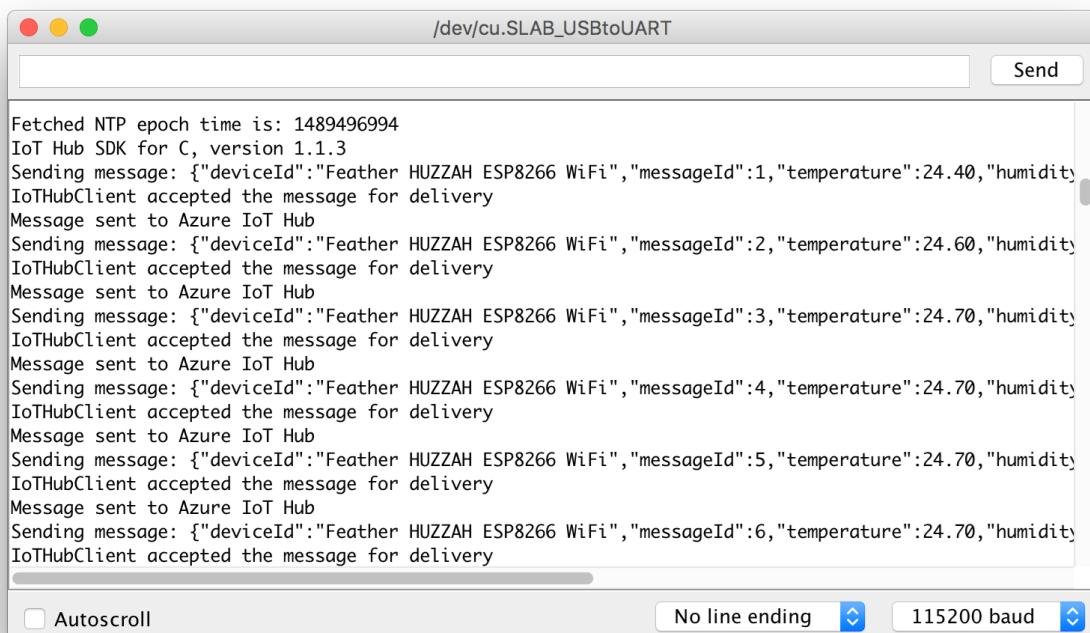
- Wi-Fi SSID
- Wi-Fi password
- Device connection string

NOTE

The credential information is stored in the EEPROM of Feather HUZZAH ESP8266. If you click the reset button on the Feather HUZZAH ESP8266 board, the sample application asks if you want to erase the information. Enter **y** to have the information erased. You are asked to provide the information a second time.

Verify the sample application is running successfully

If you see the following output from the serial monitor window and the blinking LED on Feather HUZZAH ESP8266, the sample application is running successfully.



```
Fetched NTP epoch time is: 1489496994
IoT Hub SDK for C, version 1.1.3
Sending message: {"deviceId":"Feather HUZZAH ESP8266 WiFi","messageId":1,"temperature":24.40,"humidity":50}
IoTHubClient accepted the message for delivery
Message sent to Azure IoT Hub
Sending message: {"deviceId":"Feather HUZZAH ESP8266 WiFi","messageId":2,"temperature":24.60,"humidity":50}
IoTHubClient accepted the message for delivery
Message sent to Azure IoT Hub
Sending message: {"deviceId":"Feather HUZZAH ESP8266 WiFi","messageId":3,"temperature":24.70,"humidity":50}
IoTHubClient accepted the message for delivery
Message sent to Azure IoT Hub
Sending message: {"deviceId":"Feather HUZZAH ESP8266 WiFi","messageId":4,"temperature":24.70,"humidity":50}
IoTHubClient accepted the message for delivery
Message sent to Azure IoT Hub
Sending message: {"deviceId":"Feather HUZZAH ESP8266 WiFi","messageId":5,"temperature":24.70,"humidity":50}
IoTHubClient accepted the message for delivery
Message sent to Azure IoT Hub
Sending message: {"deviceId":"Feather HUZZAH ESP8266 WiFi","messageId":6,"temperature":24.70,"humidity":50}
IoTHubClient accepted the message for delivery
```

Next steps

You have successfully connected a Feather HUZZAH ESP8266 to your IoT hub, and sent the captured sensor data to your IoT hub.

To continue to get started with Azure IoT Hub and to explore all extended IoT scenarios, see the following:

- [Manage cloud device messaging with Azure IoT Hub Toolkit extension for Visual Studio Code](#)
- [Manage devices with Azure IoT Hub Toolkit extension for Visual Studio Code](#)
- [Set up message routing](#)
- [Use Power BI to visualize real-time sensor data from your IoT hub](#)
- [Use the Web Apps feature of Azure App Service to visualize real-time sensor data from your IoT hub](#)
- [Forecast weather by using the sensor data from your IoT hub in Azure Machine Learning](#)

- Use Logic Apps for remote monitoring and notifications

Use IoT DevKit AZ3166 with Azure Functions and Cognitive Services to make a language translator

3/6/2019 • 3 minutes to read

In this article, you learn how to make IoT DevKit as a language translator by using [Azure Cognitive Services](#). It records your voice and translates it to English text shown on the DevKit screen.

The [MXChip IoT DevKit](#) is an all-in-one Arduino compatible board with rich peripherals and sensors. You can develop for it using [Azure IoT Device Workbench](#) or [Azure IoT Tools](#) extension pack in Visual Studio Code. The [projects catalog](#) contains sample applications to help you prototype IoT solutions.

Before you begin

To complete the steps in this tutorial, first do the following tasks:

- Prepare your DevKit by following the steps in [Connect IoT DevKit AZ3166 to Azure IoT Hub in the cloud](#).

Create Azure Cognitive Service

1. In the Azure portal, click **Create a resource** and search for **Speech**. Fill out the form to create the Speech Service.

The screenshot shows the Azure portal's search interface. On the left is a sidebar with navigation links like Home, Dashboard, All services, and Favorites. The main area has a search bar at the top with 'speech' typed in. Below the search bar are filters for Pricing (All), Operating System (All), and Publisher (All). The results section shows a table with columns for NAME, PUBLISHER, and CATEGORY. The first result, 'Speech', is highlighted with a red box. Other results include 'Speaker Recognition (preview)', 'Voxibot 1.0.187', and 'QnA Maker'.

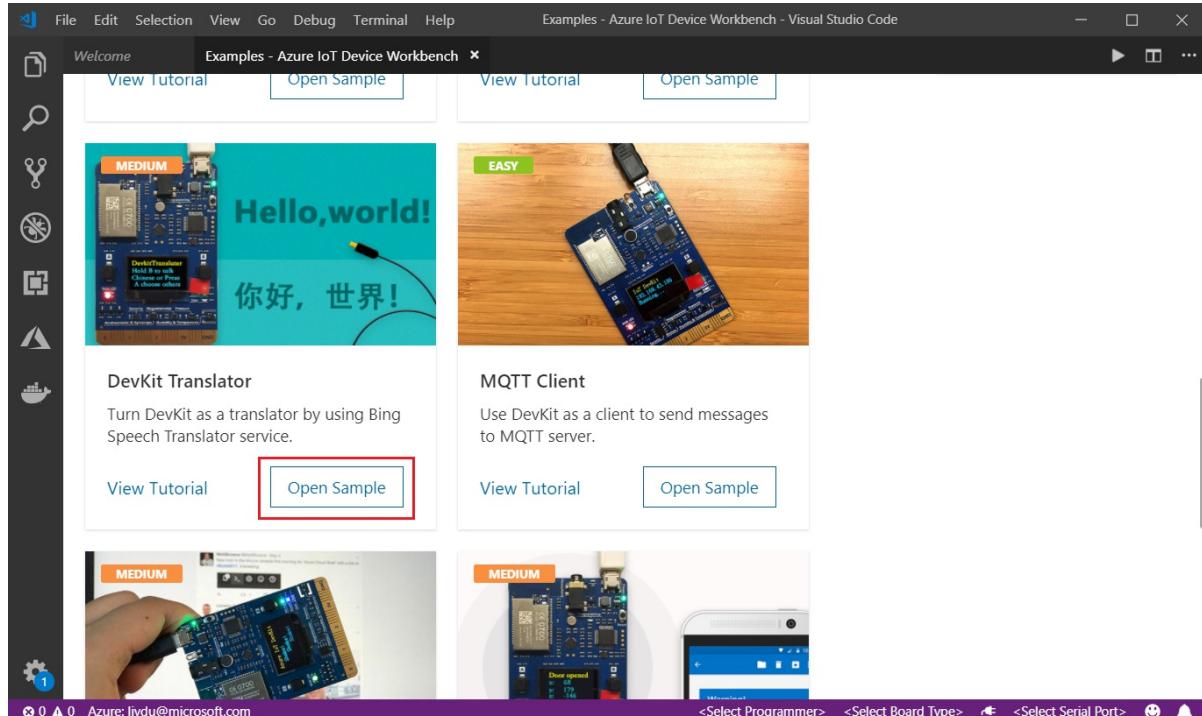
NAME	PUBLISHER	CATEGORY
Speech	Microsoft	AI + Machine Learning
Speaker Recognition (preview)	Microsoft	AI + Machine Learning
Voxibot 1.0.187	Ulex Innovative Systems	Compute
QnA Maker	Microsoft	AI + Machine Learning

2. Go to the Speech service you just created, click **Keys** section to copy and note down the **Key1** for DevKit accessing to it.

The screenshot shows the 'DevKit-test - Keys' page in the Azure portal. The left sidebar includes links for Overview, Activity log, Access control (IAM), Tags, and Diagnose and solve problems. Under RESOURCE MANAGEMENT, 'Keys' is selected. The main area shows two key sections: 'KEY 1' and 'KEY 2'. Both sections contain placeholder text: '<key1>' in KEY 1 and '<key2>' in KEY 2. Each placeholder is highlighted with a red box.

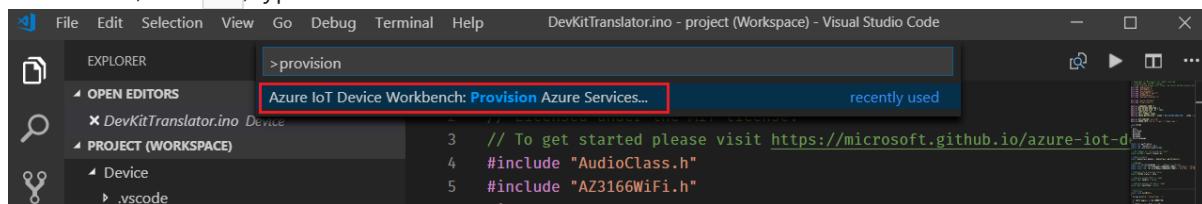
Open sample project

1. Make sure your IoT DevKit is **not connected** to your computer. Start VS Code first, and then connect the DevKit to your computer.
2. Click **F1** to open the command palette, type and select **Azure IoT Device Workbench: Open Examples....** Then select **IoT DevKit** as board.
3. In the IoT Workbench Examples page, find **DevKit Translator** and click **Open Sample**. Then selects the default path to download the sample code.

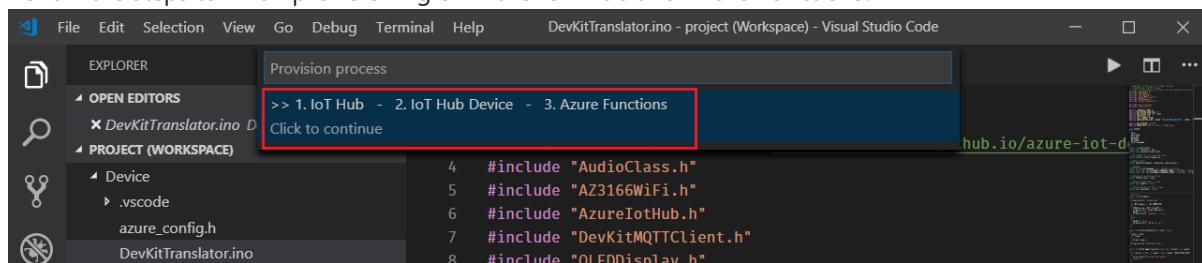


Use Speech Service with Azure Functions

1. In VS Code, click **F1**, type and select **Azure IoT Device Workbench: Provision Azure Services....**



2. Follow the steps to finish provisioning of Azure IoT Hub and Azure Functions.



Take a note of the Azure IoT Hub device name you created.

3. Open `Functions\DevKitTranslatorFunction.cs` and update the following lines of code with the device name and Speech Service key you noted down.

```

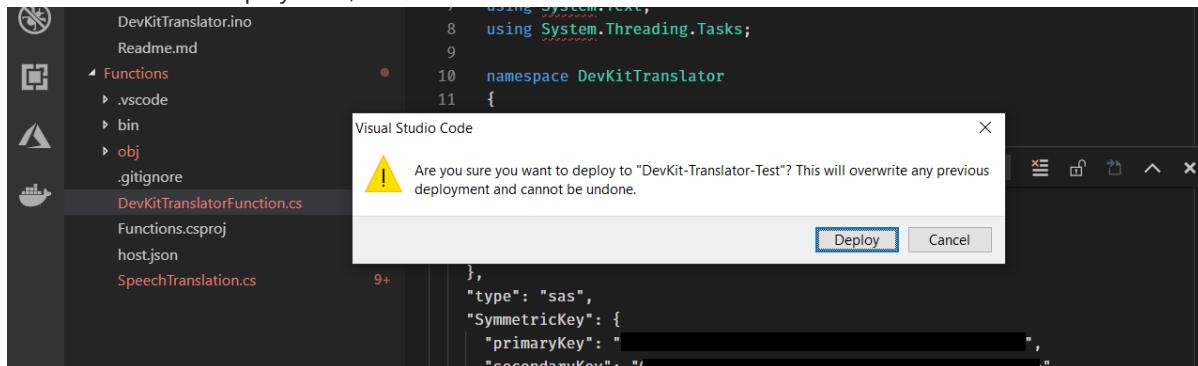
// Subscription Key of Speech Service
const string speechSubscriptionKey = "";

// Region of the speech service, see https://docs.microsoft.com/azure/cognitive-services/speech-
// service/regions for more details.
const string speechServiceRegion = "";

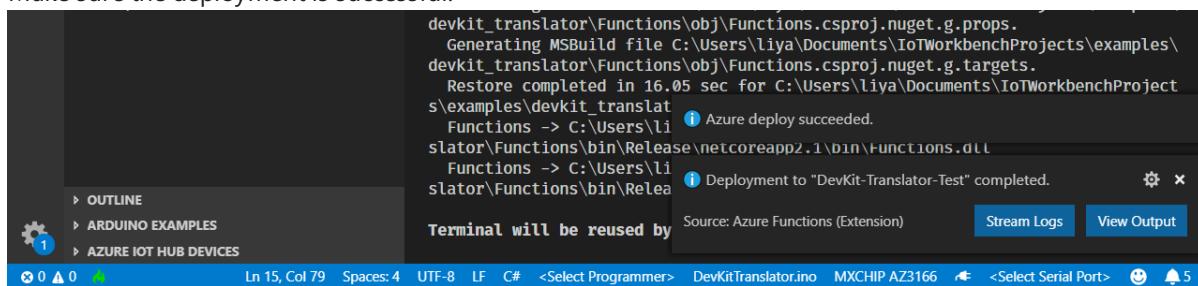
// Device ID
const string deviceName = "";

```

4. Click **F1**, type and select **Azure IoT Device Workbench: Deploy to Azure...**. If VS Code asks for confirmation for redeployment, click **Yes**.

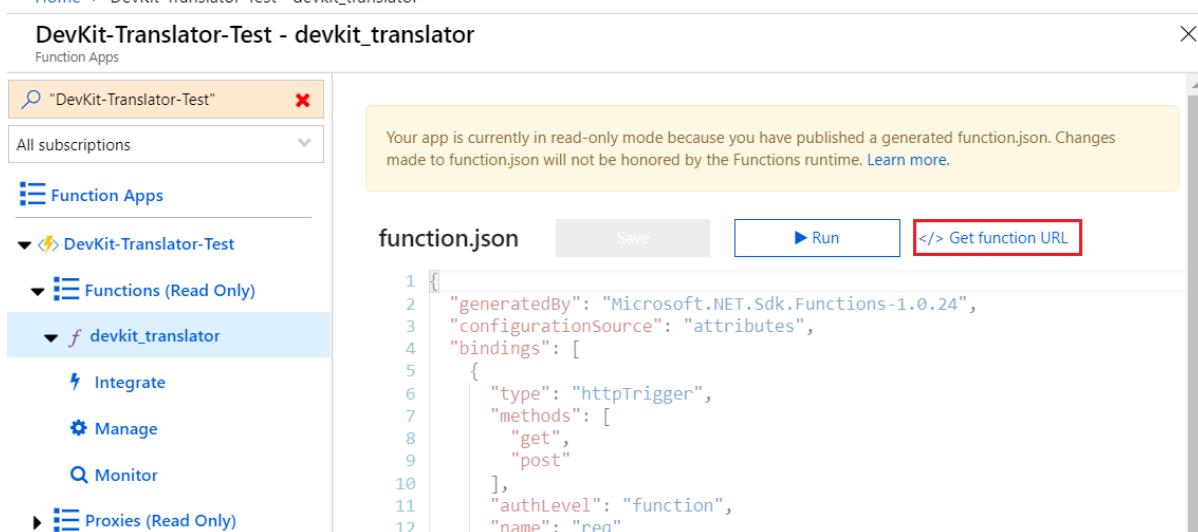


5. Make sure the deployment is successful.



6. In Azure portal, go to **Functions Apps** section, find the Azure Function app just created. Click

devkit_translator, then click **</> Get Function URL** to copy the URL.



7. Paste the URL into **azure_config.h** file.

The screenshot shows the Visual Studio Code interface with two open files: `DevKitTranslatorFunction.cs` and `azure_config.h`. The `DevKitTranslatorFunction.cs` file contains C# code for an Azure Function. The `azure_config.h` file contains C preprocessor definitions, including the `AZURE_FUNCTION_URL` definition set to a specific Azure website URL.

```
#define AZURE_FUNCTION_URL  
"https://devkit-translator-test.azurewebsites.net/api/devkit_translator?  
code:"
```

NOTE

If the Function app does not work properly, check this [FAQ](#) section to resolve it.

Build and upload device code

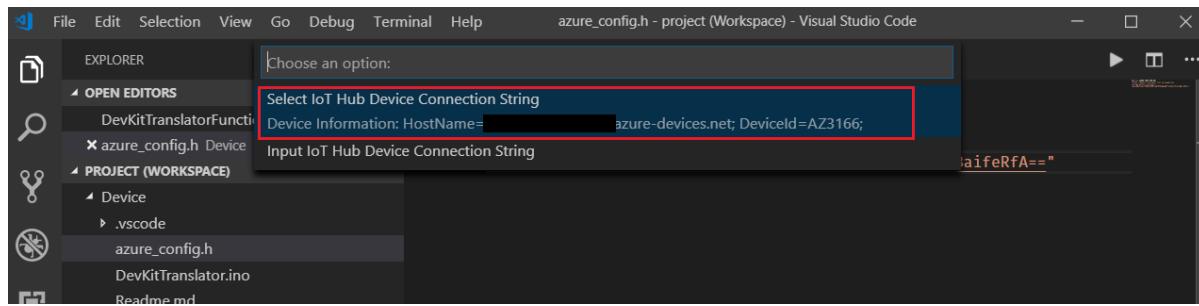
1. Switch the DevKit to **configuration mode** by:

- Hold down button **A**.
- Press and release **Reset** button.

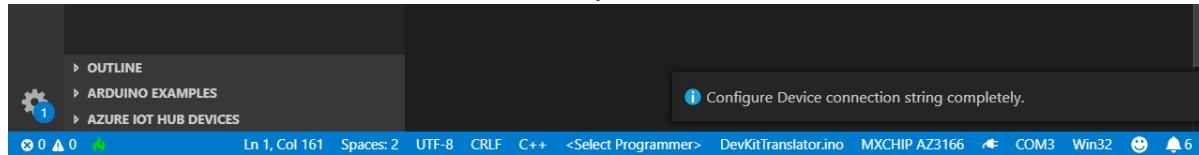
You will see the screen displays the DevKit ID and **Configuration**.



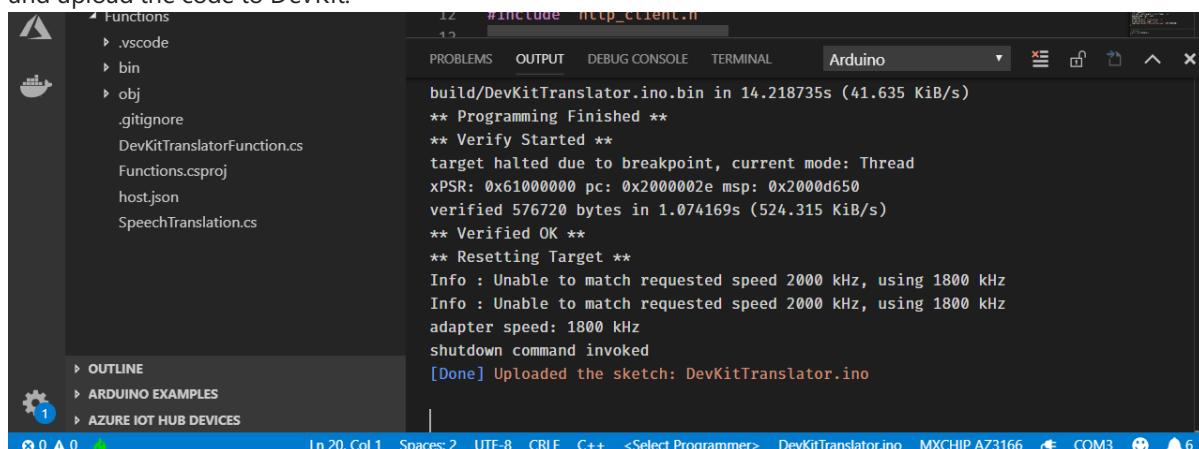
2. Click **F1**, type and select **Azure IoT Device Workbench: Configure Device Settings... > Config Device Connection String**. Select **Select IoT Hub Device Connection String** to configure it to the DevKit.



3. You will see the notification once it's done successfully.



4. Click **F1** again, type and select **Azure IoT Device Workbench: Upload Device Code**. It starts compile and upload the code to DevKit.

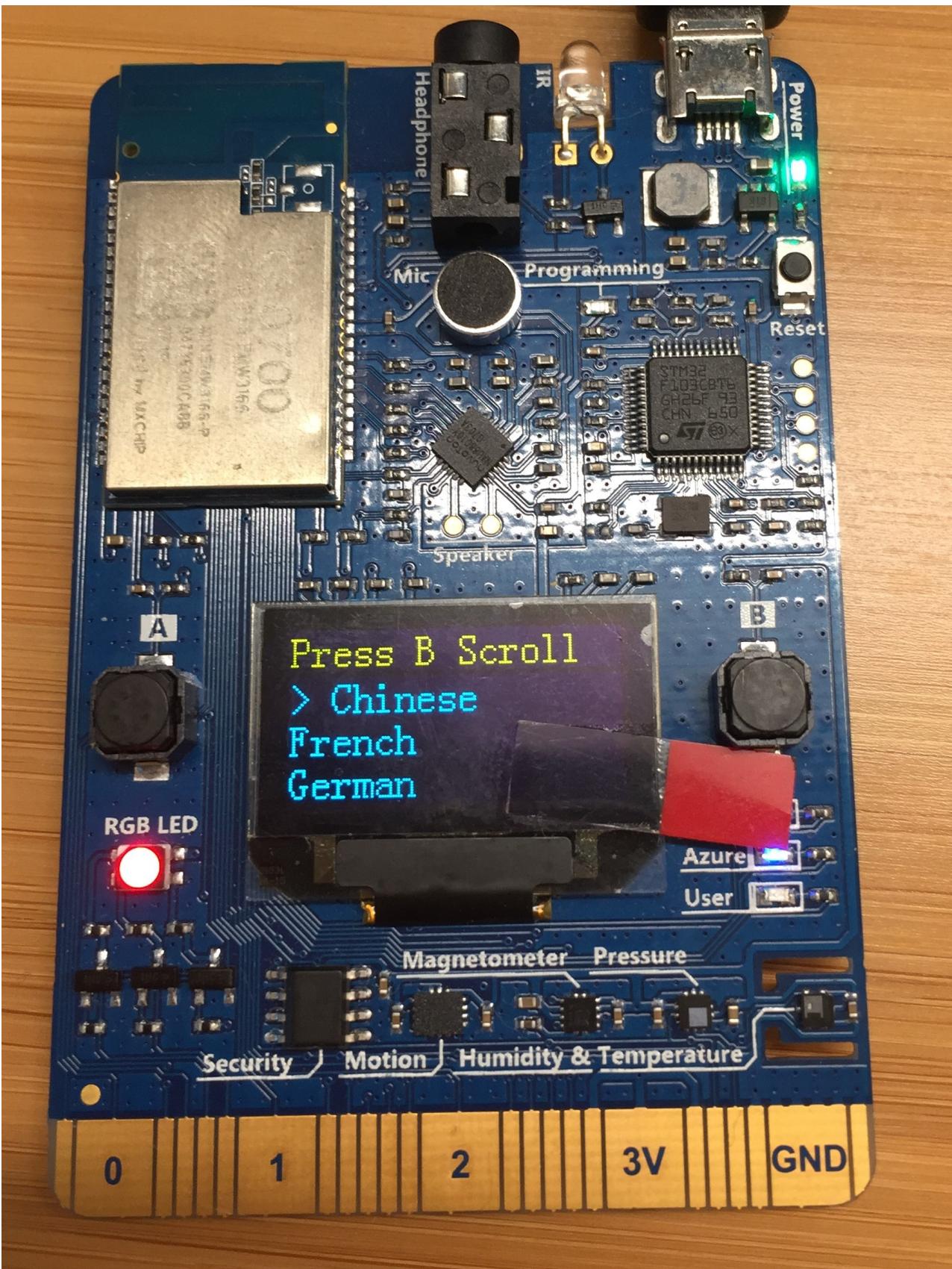


Test the project

After app initialization, follow the instructions on the DevKit screen. The default source language is Chinese.

To select another language for translation:

1. Press button A to enter setup mode.
2. Press button B to scroll all supported source languages.
3. Press button A to confirm your choice of source language.
4. Press and hold button B while speaking, then release button B to initiate the translation.
5. The translated text in English shows on the screen.

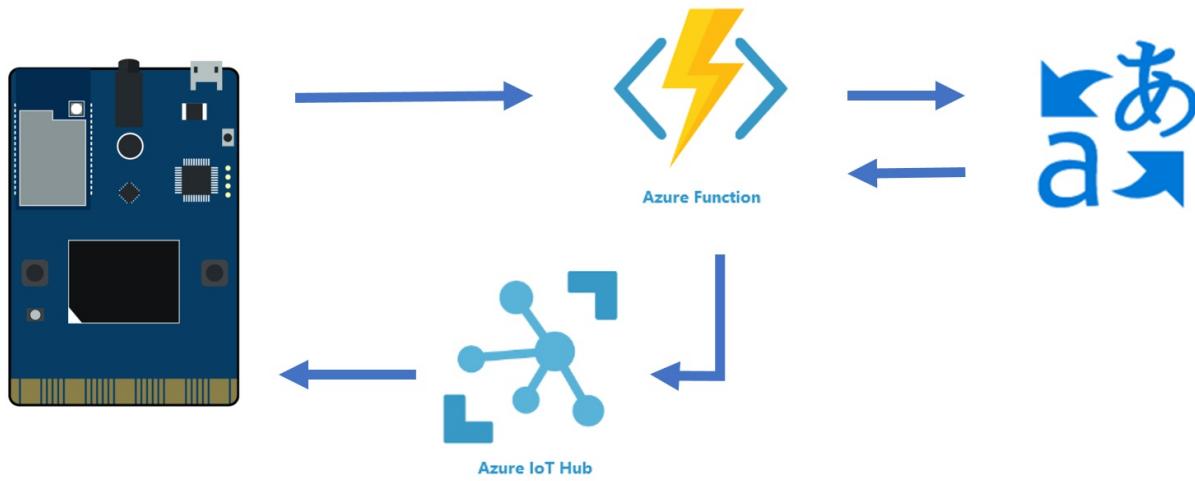




On the translation result screen, you can:

- Press buttons A and B to scroll and select the source language.
- Press the B button to talk. To send the voice and get the translation text, release the B button.

How it works



The IoT DevKit records your voice then posts an HTTP request to trigger Azure Functions. Azure Functions calls the cognitive service speech translator API to do the translation. After Azure Functions gets the translation text, it sends a C2D message to the device. Then the translation is displayed on the screen.

Problems and feedback

If you encounter problems, refer to the [IoT DevKit FAQ](#) or reach out to us using the following channels:

- [Gitter.im](#)
- [Stack Overflow](#)

Next steps

You have learned how to use the IoT DevKit as a translator by using Azure Functions and Cognitive Services. In this how-to, you learned how to:

- Use Visual Studio Code task to automate cloud provisions
- Configure Azure IoT device connection string
- Deploy the Azure Function
- Test the voice message translation

Advance to the other tutorials to learn:

[Connect IoT DevKit AZ3166 to Azure IoT Remote Monitoring solution accelerator](#)

Shake, Shake for a Tweet -- Retrieve a Twitter message with Azure Functions

3/6/2019 • 5 minutes to read

In this project, you learn how to use the motion sensor to trigger an event using Azure Functions. The app retrieves a random tweet with a #hashtag you configure in your Arduino sketch. The tweet displays on the DevKit screen.

What you need

Finish the [Getting Started Guide](#) to:

- Have your DevKit connected to Wi-Fi.
- Prepare the development environment.

An active Azure subscription. If you don't have one, you can register via one of these methods:

- Activate a [free 30-day trial Microsoft Azure account](#)
- Claim your [Azure credit](#) if you are an MSDN or Visual Studio subscriber

Open the project folder

Start by opening the project folder.

Start VS Code

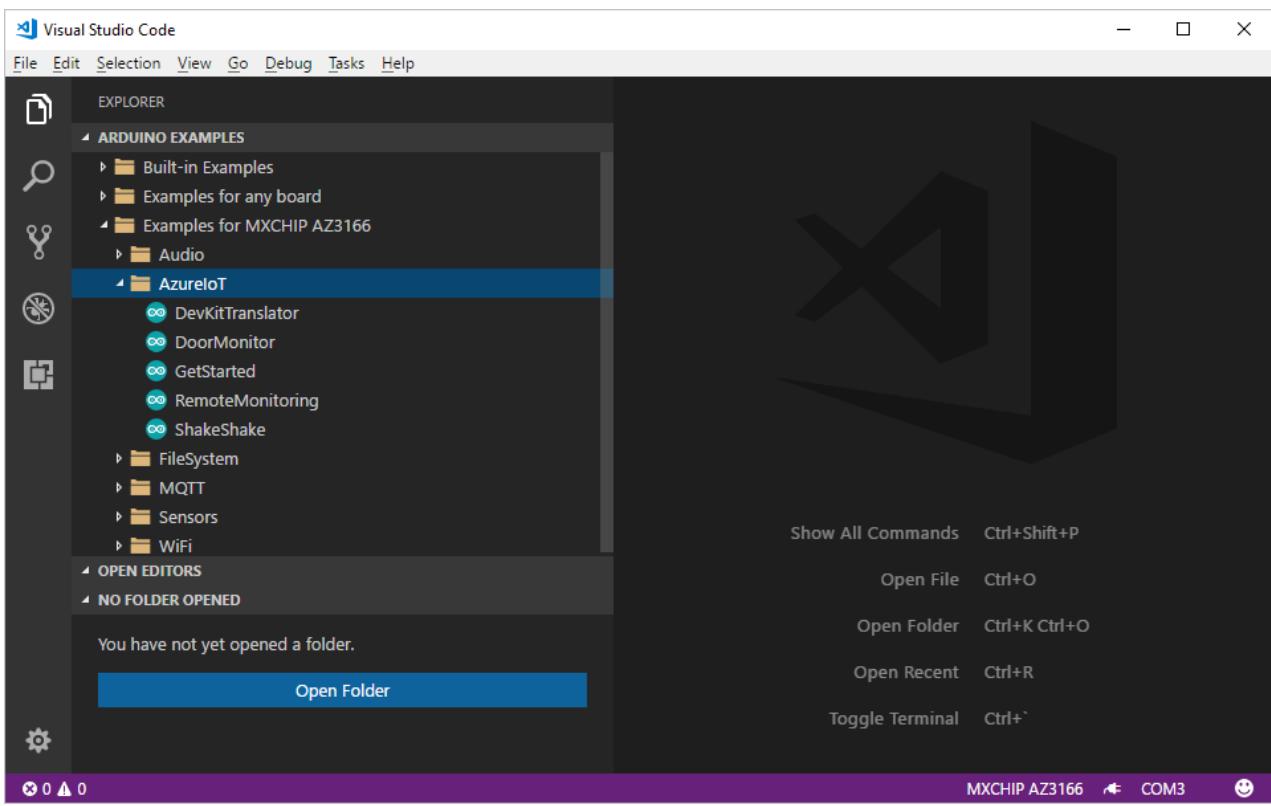
- Make sure your DevKit is connected to your computer.
- Start VS Code.
- Connect the DevKit to your computer.

NOTE

When launching VS Code, you may receive an error message that the Arduino IDE or related board package can't be found. If this error occurs, close VS Code and launch the Arduino IDE again. VS Code should now locate the Arduino IDE path correctly.

Open the Arduino Examples folder

Expand the left side **ARDUINO EXAMPLES** section, browse to **Examples for MXCHIP AZ3166 > AzureIoT**, and select **ShakeShake**. A new VS Code window opens, displaying the project folder. If you can't see the MXCHIP AZ3166 section, make sure your device is properly connected and restart Visual Studio Code.
the

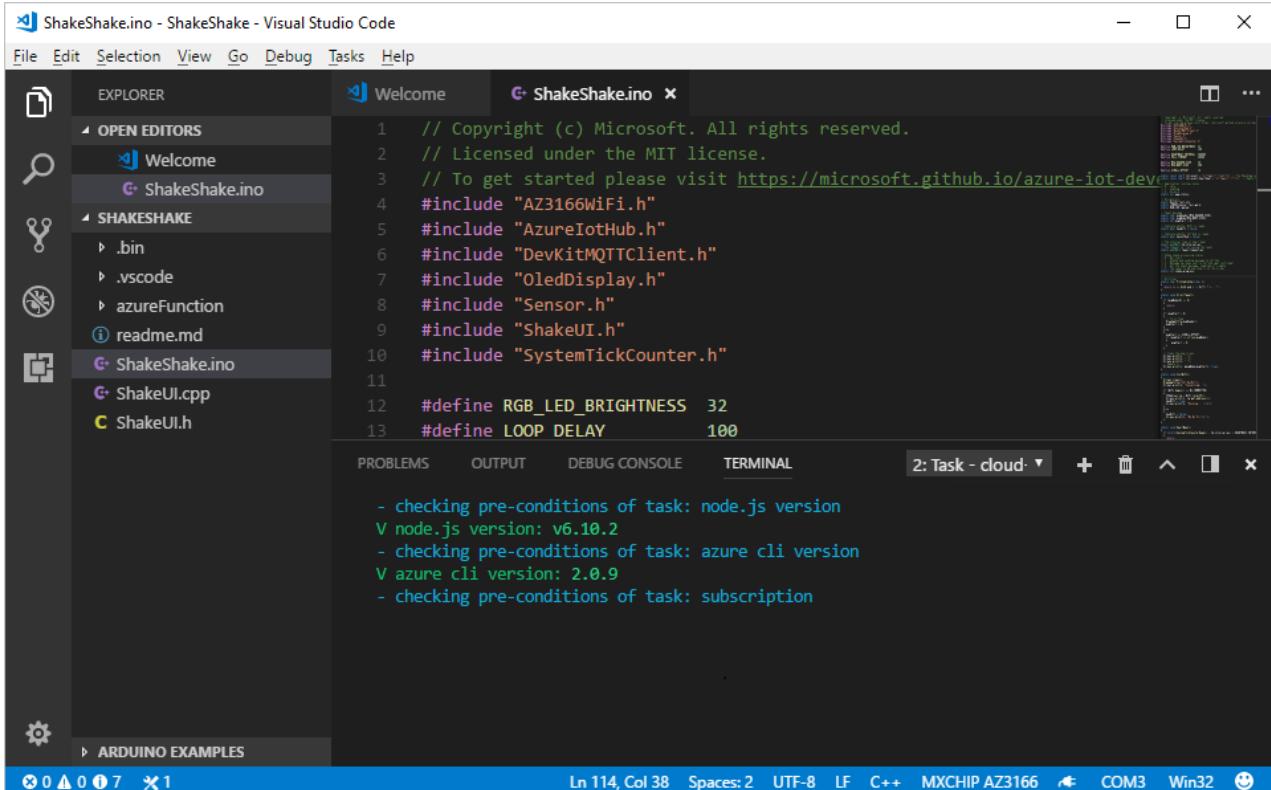


You can also open the sample project from command palette. Click `Ctrl+Shift+P` (macOS: `Cmd+Shift+P`) to open the command palette, type **Arduino**, and then find and select **Arduino: Examples**.

Provision Azure services

In the solution window, run your task through `Ctrl+P` (macOS: `Cmd+P`) by entering `task cloud-provision`.

In the VS Code terminal, an interactive command line guides you through provisioning the required Azure services:



NOTE

If the page hangs in the loading status when trying to sign in to Azure, refer to the "["login page hangs" step in the IoT DevKit FAQ](#).

Modify the #hashtag

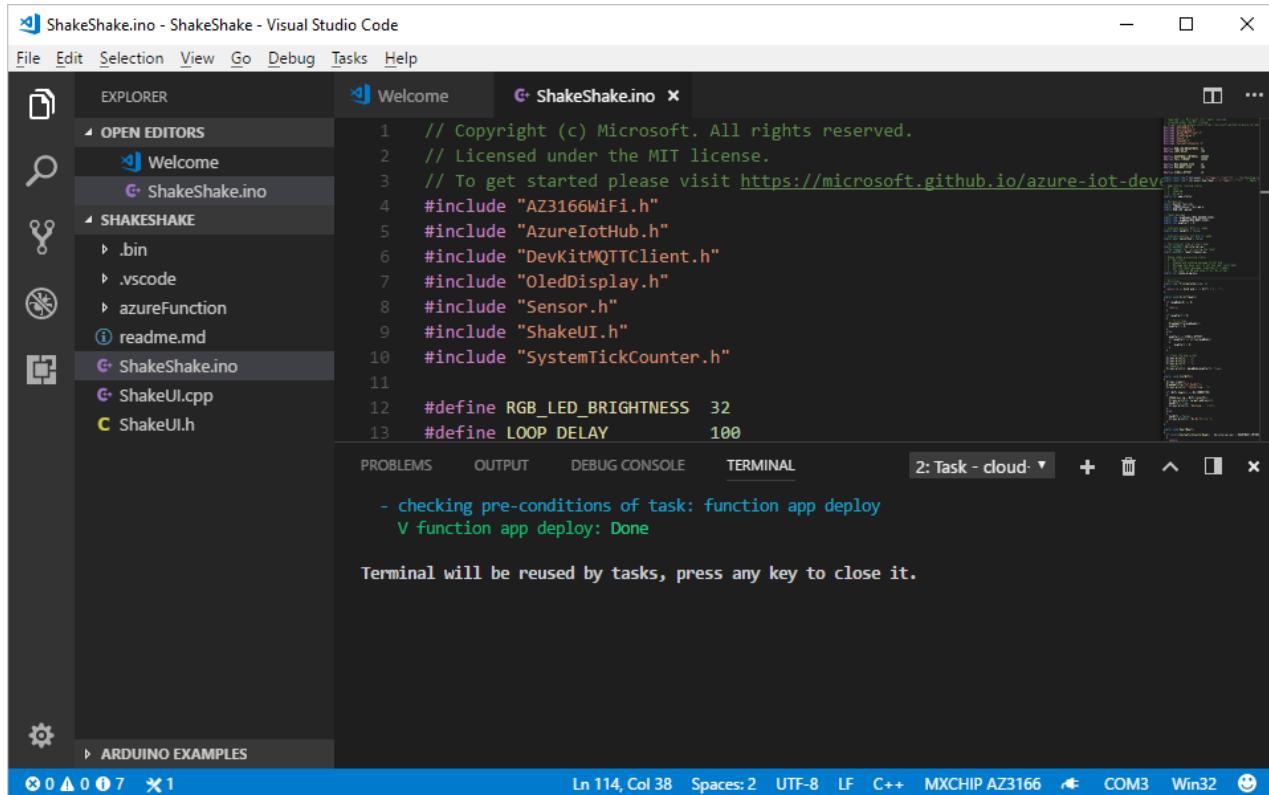
Open `ShakeShake.ino` and look for this line of code:

```
static const char* iot_event = "{\"topic\":\"iot\"}";
```

Replace the string `iot` within the curly braces with your preferred hashtag. The DevKit later retrieves a random tweet that includes the hashtag you specify in this step.

Deploy Azure Functions

Use `Ctrl+P` (macOS: `Cmd+P`) to run `task cloud-deploy` to start deploying the Azure Functions code:



```
// Copyright (c) Microsoft. All rights reserved.
// Licensed under the MIT license.
// To get started please visit https://microsoft.github.io/azure-iot-devkit
#include "AZ3166WiFi.h"
#include "AzureIoTHub.h"
#include "DevKitMQTTClient.h"
#include "OledDisplay.h"
#include "Sensor.h"
#include "ShakeUI.h"
#include "SystemTickCounter.h"

#define RGB_LED_BRIGHTNESS 32
#define LOOP_DELAY 100
```

NOTE

Occasionally, the Azure Function may not work properly. To resolve this issue when it occurs, check the "["compilation error" section of the IoT DevKit FAQ](#)".

Build and upload the device code

Next, build and upload the device code.

Windows

1. Use `Ctrl+P` to run `task device-upload`.
2. The terminal prompts you to enter configuration mode. To do so:

- Hold down button A
 - Push and release the reset button.
3. The screen displays the DevKit ID and 'Configuration'.

macOS

1. Put the DevKit into configuration mode:

Hold down button A, then push and release the reset button. The screen displays 'Configuration'.

2. Use `Cmd+P` to run `task device-upload` to set the connection string that is retrieved from the `task cloud-provision` step.

Verify, upload, and run

Now the connection string is set, it verifies and uploads the app, then runs it.

1. VS Code starts verifying and uploading the Arduino sketch to your DevKit:

```

VS Code - ShakeShake.ino - ShakeShake - Visual Studio Code
File Edit Selection View Go Debug Tasks Help
EXPLORER Welcome ShakeShake.ino
OPEN EDITORS
  Welcome
  ShakeShake.ino
SHAKESHAKE
  .bin
  .vscode
  azureFunction
  readme.md
  ShakeShake.ino
  ShakeUI.cpp
  ShakeUI.h
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
2: Task - device
- checking pre-conditions of task: Check Arduino Board
V Check Arduino Board: MXCHIP_AZ3166 as MXCHIP AZ3166
- checking pre-conditions of task: Build & Upload Sketch
Deleting d:/VSIoT/temp/ShakeShake/.build/ShakeShake.ino.bin
C:\Program Files (x86)\Arduino\arduino_debug.exe --upload --board AZ3166:stm32f4:MXCHIP_AZ3166
--preferences-file d:/VSIoT/temp/ShakeShake\.build/pref.txt --pref compiler.warning_level=none --
pref.build.path=d:/VSIoT/temp/ShakeShake\.build d:/VSIoT/temp\ShakeShake\ShakeShake.ino
Loading configuration...
Initializing packages...
Preparing boards...
Verifying...
Ln 114, Col 38 Spaces:2 UTF-8 LF C++ MXCHIP AZ3166 COM3 Win32

```

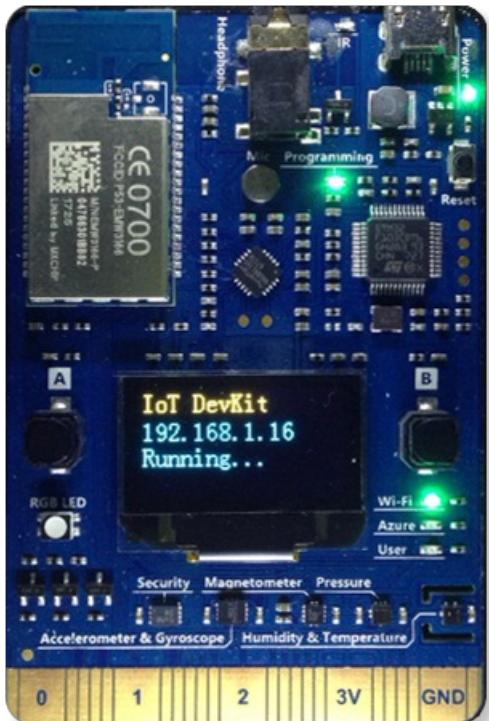
2. The DevKit reboots and starts running the code.

You may get an "Error: AZ3166: Unknown package" error message. This error occurs when the board package index isn't refreshed correctly. To resolve this issue, check the ["unknown package" error in the IoT DevKit FAQ](#).

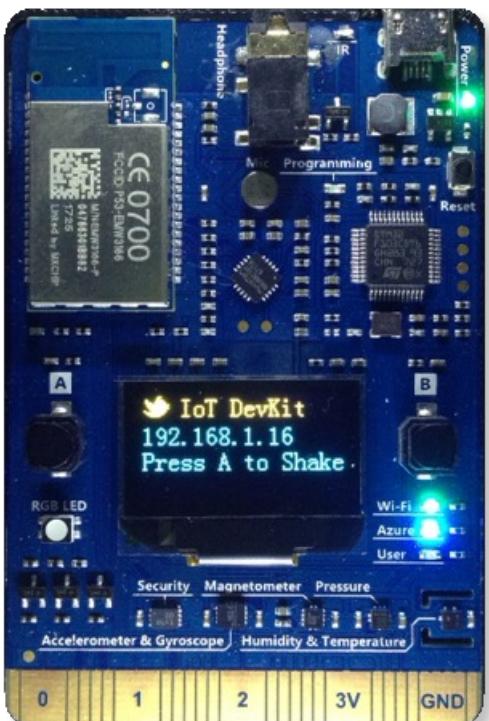
Test the project

After app initialization, click and release button A, then gently shake the DevKit board. This action retrieves a random tweet, which contains the hashtag you specified earlier. Within a few seconds, a tweet displays on your DevKit screen:

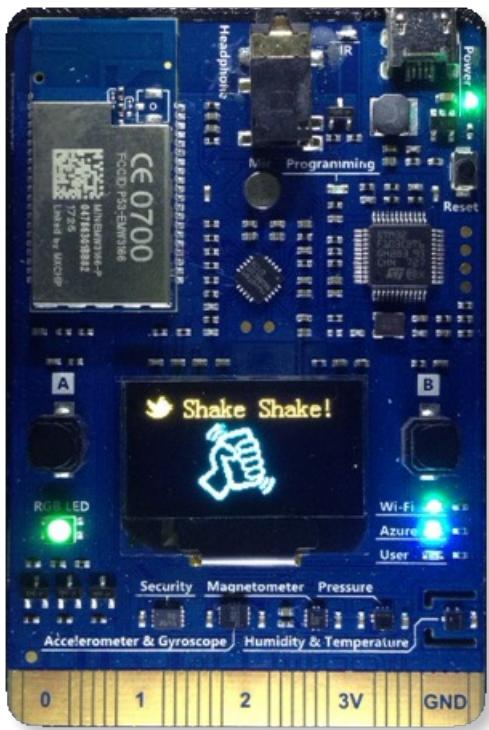
Arduino application initializing...



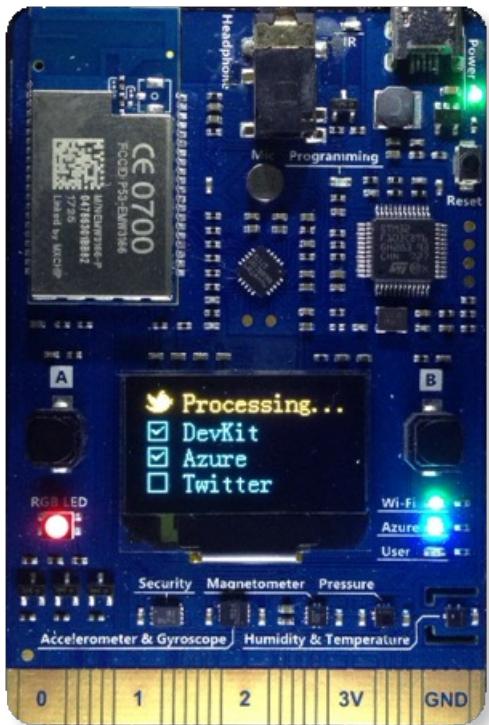
Press A to shake...



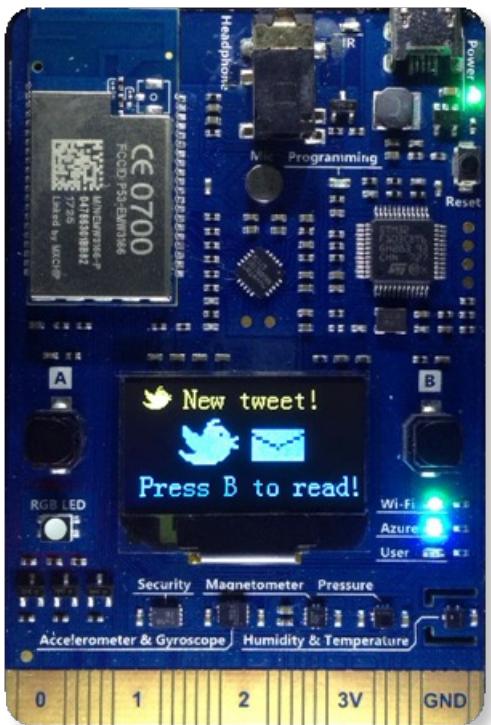
Ready to shake...



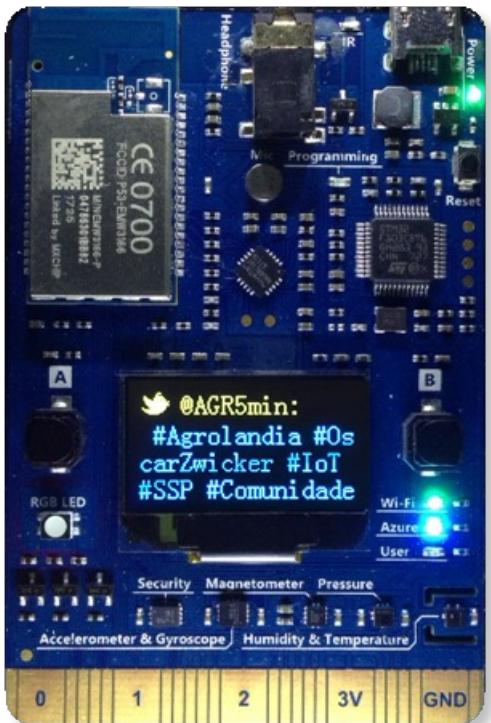
Processing...



Press B to read...

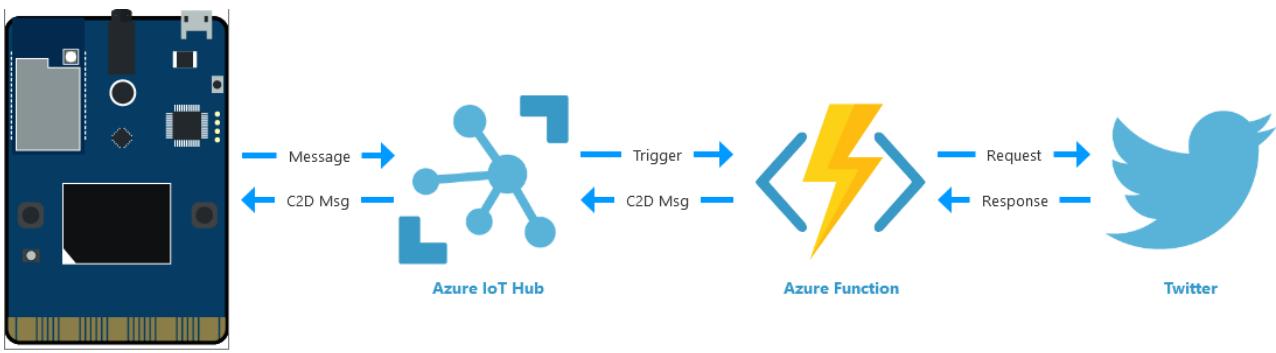


Display a random tweet...



- Press button A again, then shake for a new tweet.
- Press button B to scroll through the rest of the tweet.

How it works



The Arduino sketch sends an event to the Azure IoT Hub. This event triggers the Azure Functions app. The Azure Functions app contains the logic to connect to Twitter's API and retrieve a tweet. It then wraps the tweet text into a C2D (Cloud-to-device) message and sends it back to the device.

Optional: Use your own Twitter bearer token

For testing purposes, this sample project uses a pre-configured Twitter bearer token. However, there is a [rate limit](#) for every Twitter account. If you want to consider using your own token, follow these steps:

1. Go to [Twitter Developer portal](#) to register a new Twitter app.
2. [Get Consumer Key and Consumer Secrets](#) of your app.
3. Use [some utility](#) to generate a Twitter bearer token from these two keys.
4. In the [Azure portal](#) (:target="_blank"), get into the **Resource Group** and find the Azure Function (Type: App Service) for your "Shake, Shake" project. The name always contains 'shake...' string.

The screenshot shows the Azure portal interface with the following details:

- Tags:** Tags section.
- SETTINGS:** Quickstart, Resource costs, Deployments.
- Filter by name...**: Search bar.
- All types**, **All locations**, **No grouping** dropdowns.
- 6 items** listed in a table:

NAME	TYPE	LOCATION
devkit-iot-hub-iohub-b0a4	IoT Hub	West US
devkit-iot-hub-iohub-b0a4-shakeae27	App Service plan	West US
devkit-iot-hub-iohub-b0a4-shakeae27	App Service	West US

5. Update the code for `run.csx` within **Functions > shakeshake-cs** with your own token:

The screenshot shows the Azure Functions blade with the following details:

- Visual Studio Enterprise** dropdown.
- Function Apps** list: devkit-iot-hub-iohub-b0a4, shakeshake-cs (selected).
- Functions** list: shakeshake-cs (selected).
- Integrate**, **Manage**, **Monitor** buttons.
- Proxies (preview)** and **Slots (preview)** sections.
- Code Editor** for `run.csx`:


```

string authHeader = "Bearer " + "[your own token]";

42   throw new ShakeShakeException($"Failed to deserialize message:{myEventHubMessage}");
43 }
44 }
45
46 if (String.IsNullOrEmpty(deviceObject.topic))
47 {
48     // No hash tag or this is a heartbeat package
49     return;
50 }
51
52 string message = string.Empty;
53 try
54 {
55     string tweet = string.Empty;
56     string url = "https://api.twitter.com/1.1/search/tweets.json" + "?count=3&q=%20";
57     string authHeader = "Bearer " + "AAAAAAAAAAAAAAAAGVU0AAAAAAUpcxA9aXc2T06";
58
59     HttpWebRequest request = (HttpWebRequest)WebRequest.Create(url);
60     request.Headers.Add("Authorization", authHeader);
61     request.Method = "GET";
62     request.ContentType = "application/x-www-form-urlencoded; charset=UTF-8";
63
64     var response = (HttpWebResponse)request.GetResponse();
65     var reader = new StreamReader(response.GetResponseStream());
66     string objText = reader.ReadToEnd();
67 }
```

6. Save the file and click **Run**.

Problems and feedback

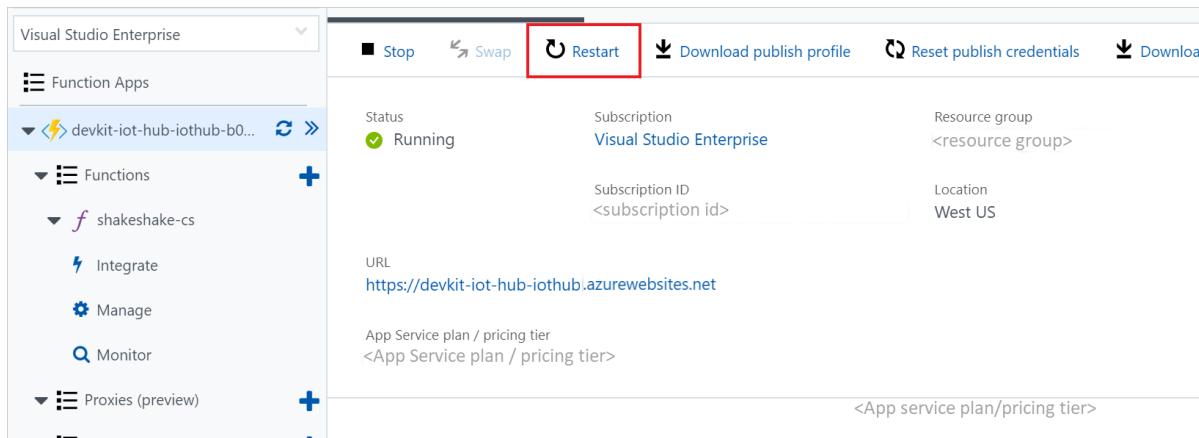
How to troubleshoot problems or provide feedback.

Problems

One problem you could see if the screen displays 'No Tweets' while every step has run successfully. This condition normally happens the first time you deploy and run the sample because the function app requires anywhere from a couple of seconds to as much as one minute to cold start the app.

Or, when running the code, there are some blips that cause a restarting of the app. When this condition happens, the device app can get a timeout for fetching the tweet. In this case, you may try one or both of these methods to solve the issue:

1. Click the reset button on the DevKit to run the device app again.
2. In the [Azure portal](#), find the Azure Functions app you created and restart it:



The screenshot shows the Azure portal interface for managing Azure Functions. On the left, there's a sidebar with 'Visual Studio Enterprise' selected under 'Function Apps'. Below it, a list includes 'devkit-iot-hub-iothub-b0...' and 'shakeshake-cs'. To the right of the sidebar is a main panel for the 'shakeshake-cs' function. At the top of this panel are several buttons: 'Stop', 'Swap', 'Restart' (which is highlighted with a red box), 'Download publish profile', 'Reset publish credentials', and 'Download'. Below these buttons, the function details are listed: Status (Running), Subscription (Visual Studio Enterprise), Resource group (<resource group>), Subscription ID (<subscription id>), Location (West US), URL (<https://devkit-iot-hub-iothub.azurewebsites.net>), App Service plan / pricing tier (<App Service plan / pricing tier>), and <App service plan/pricing tier>.

Feedback

If you experience other problems, refer to the [IoT DevKit FAQ](#) or contact us using the following channels:

- [Gitter.im](#)
- [Stack Overflow](#)

Next steps

Now that you have learned how to connect a DevKit device to your Azure IoT Remote Monitoring solution accelerator and retrieve a tweet, here are the suggested next steps:

- [Azure IoT Remote Monitoring solution accelerator overview](#)

Send messages to an MQTT server

3/6/2019 • 2 minutes to read

Internet of Things (IoT) systems often deal with intermittent, poor quality, or slow internet connections. MQTT is a machine-to-machine (M2M) connectivity protocol, which was developed with such challenges in mind.

The MQTT client library used here is part of the [Eclipse Paho](#) project, which provides APIs for using MQTT over multiple means of transport.

What you learn

In this project, you learn:

- How to use the MQTT Client library to send messages to an MQTT broker.
- How to configure your MXChip IoT DevKit as an MQTT client.

What you need

Finish the [Getting Started Guide](#) to:

- Have your DevKit connected to Wi-Fi
- Prepare the development environment

Open the project folder

1. If the DevKit is already connect to your computer, disconnect it.
2. Start VS Code.
3. Connect the DevKit to your computer.

Open the MQTTClient Sample

Expand left side **ARDUINO EXAMPLES** section, browse to **Examples for MXCHIP AZ3166 > MQTT**, and select **MQTTClient**. A new VS Code window opens with a project folder in it.

NOTE

You can also open example from command palette. Use `Ctrl+Shift+P` (macOS: `Cmd+Shift+P`) to open the command palette, type **Arduino**, and then find and select **Arduino: Examples**.

Build and upload the Arduino sketch to the DevKit

Type `Ctrl+P` (macOS: `Cmd+P`) to run `task device-upload`. Once the upload is completed, DevKit restarts and runs the sketch.

```

auto erase enabled
Info : device id = 0x30006441
Info : flash size = 1024kbytes
target halted due to breakpoint, current mode: Thread
xPSR: 0x61000000 pc: 0x20000046 msp: 0x2000e784
wrote 360448 bytes from file C:\Users\dol\Documents\Arduino\generated_examples\MQTTClient_4\.build/MQTTClient.ino.bin in 8
.525967s (41.286 KiB/s)
** Programming Finished **
** Verify Started **
target halted due to breakpoint, current mode: Thread
xPSR: 0x61000000 pc: 0x2000002e msp: 0x2000e784
verified 353284 bytes in 0.721824s (477.961 KiB/s)
** Verified OK **
** Resetting Target **
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
arduino debug.exe exited.
  V Build & Upload Sketch: success
Press any key to close the terminal

```

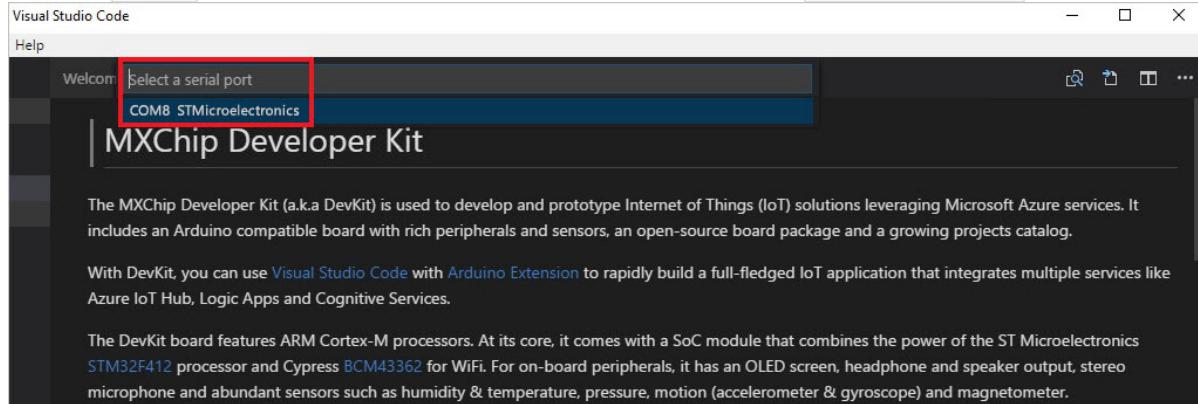
NOTE

You may receive an "Error: AZ3166: Unknown package" error message. This error occurs when the board package index is not refreshed correctly. To resolve this error, refer to the [development section of the IoT DevKit FAQ](#).

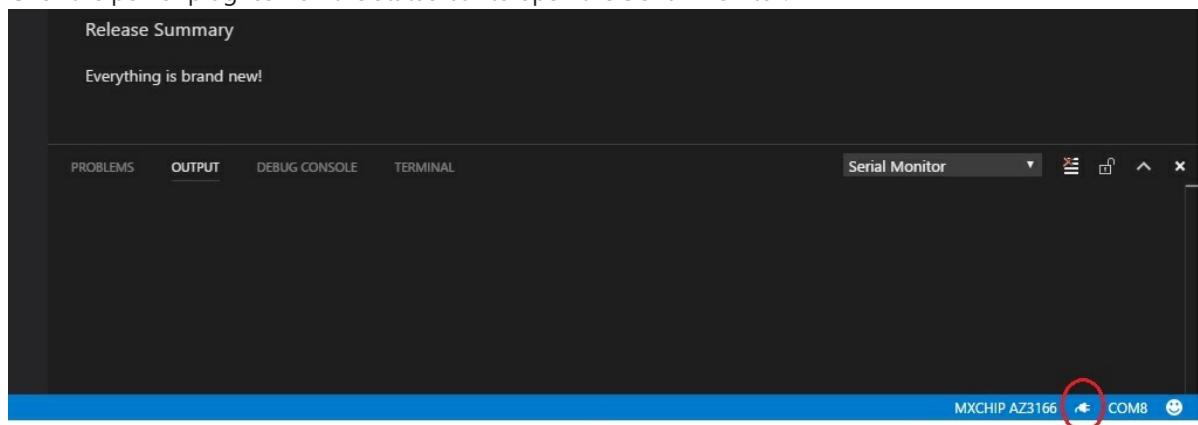
Test the project

In VS Code, follow this procedure to open and set up the Serial Monitor:

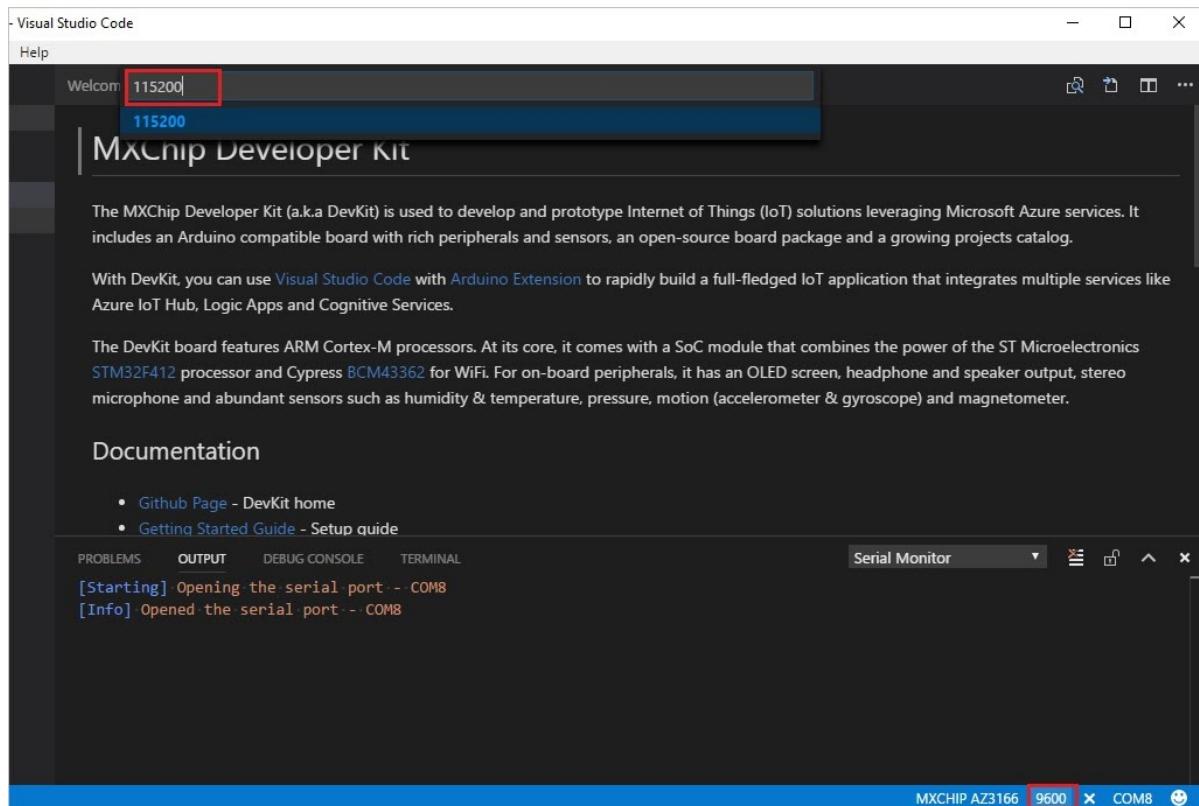
1. Click the `COM[X]` word on the status bar to set the right COM port with `STMicroelectronics`:



2. Click the power plug icon on the status bar to open the Serial Monitor:



3. On the status bar, click the number that represents the Baud Rate and set it to `115200`:



The Serial Monitor displays all the messages sent by the sample sketch. The sketch connects the DevKit to Wi-Fi. Once the Wi-Fi connection is successful, the sketch sends a message to the MQTT broker. After that, the sample repeatedly sends two "iot.eclipse.org" messages using QoS 0 and QoS 1, respectively.

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
*****
** MXChip - Microsoft Azure IoT Developer Kit **
*****
You can 1. press Button A and reset to enter configuration mode.
..... 2. press Button B and reset to enter AP mode.

Attempting to connect to Wi-Fi, SSID: Ruff_R0101965
Time is now (UTC): Tue Jun 13 09:52:34 2017

>>>Enter Loop
Connecting to MQTT server iot.eclipse.org:1883
Connected to MQTT server successfully
Message arrived: qos 0, retained 0, dup 0, packetid 0
Payload: QoS 0 message from AZ3166!
Message arrived: qos 1, retained 0, dup 0, packetid 1
Payload: QoS 1 message from AZ3166!
Finish message count: 2

>>>Enter Loop
Connecting to MQTT server iot.eclipse.org:1883
Connected to MQTT server successfully
Message arrived: qos 0, retained 0, dup 0, packetid 0
```

Problems and feedback

If you encounter problems, refer to the [IoT DevKit FAQ](#) or connect using the following channels:

- [Gitter.im](#)
- [Stack Overflow](#)

See also

- [Connect IoT DevKit AZ3166 to Azure IoT Hub in the cloud](#)
- [Shake, Shake for a Tweet](#)

Next steps

Now that you have learned how to configure your MXChip IoT DevKit as an MQTT client and use the MQTT Client library to send messages to an MQTT broker, here are the suggested next steps:

- [Azure IoT Remote Monitoring solution accelerator overview](#)
- [Connect an MXChip IoT DevKit device to your Azure IoT Central application](#)

Door Monitor

3/6/2019 • 5 minutes to read

The MXChip IoT DevKit contains a built-in magnetic sensor. In this project, you detect the presence or absence of a nearby strong magnetic field -- in this case, coming from a small, permanent magnet.

What you learn

In this project, you learn:

- How to use the MXChip IoT DevKit's magnetic sensor to detect the movement of a nearby magnet.
- How to use the SendGrid service to send a notification to your email address.

NOTE

For a practical use of this project, perform the following tasks:

- Mount a magnet to the edge of a door.
- Mount the DevKit on the door jamb close to the magnet. Opening or closing the door will trigger the sensor, resulting in your receiving an email notification of the event.

What you need

Finish the [Getting Started Guide](#) to:

- Have your DevKit connected to Wi-Fi
- Prepare the development environment

An active Azure subscription. If you do not have one, you can register via one of these methods:

- Activate a [free 30-day trial Microsoft Azure account](#).
- Claim your [Azure credit](#) if you are an MSDN or Visual Studio subscriber.

Deploy the SendGrid service in Azure

[SendGrid](#) is a cloud-based email delivery platform. This service will be used to send email notifications.

NOTE

If you have already deployed a SendGrid service, you may proceed directly to [Deploy IoT Hub in Azure](#).

SendGrid Deployment

To provision Azure services, use the **Deploy to Azure** button. This button enables quick and easy deployment of your open-source projects to Microsoft Azure.

Click the **Deploy to Azure** button below.



If you are not already signed into your Azure account, sign in now.

You now see the SendGrid sign-up form.

Custom deployment
Deploy from a custom template

BASICS

* Subscription: IoT Tooling Tests with TTL = 7 Days

* Resource group: Create new (radio button selected) / Use existing

* Location: West US

SETTINGS

* Name: [Input field]

* Password: [Input field]

Plan_name: free - \$0 / month

* Email: [Input field]

* First Name: [Input field]

* Last Name: [Input field]

Company: [Input field]

Website: [Input field]

Pin to dashboard

Purchase

Complete the sign-up form:

- **Resource group:** Create a resource group to host the SendGrid service, or use an existing one. See [Using resource groups to manage your Azure resources](#).
- **Name:** The name for your SendGrid service. Choose a unique name, differing from other services you may have.
- **Password:** The service requires a password, which will not be used for anything in this project.
- **Email:** The SendGrid service will send verification to this email address.

Check the **Pin to dashboard** option to make this application easier to find in the future, then click **Purchase** to submit the sign-in form.

SendGrid API Key creation

After the deployment completes, click it and then click the **Manage** button. Your SendGrid account page appears, where you need to verify your email address.

The screenshot shows the Azure portal interface for a SendGrid account named 'DoorMonitor'. The left sidebar lists account details: Resource group 'DoorMonitor', Status 'Running', Location 'West US', Subscription name 'IoT Tooling Tests with TTL = 7 Days', and Subscription ID '<subscription-id>'. A red box highlights the 'Manage' button. The main content area shows 'Pricing Information' with a 'FREE' tier and 25,000 emails/month. A red box highlights the 'All settings →' button. The right sidebar is titled 'Settings' and contains sections for 'SUPPORT + TROUBLESHOOTING' (Activity log), 'GENERAL' (Properties, Configurations, Contact Information), and 'RESOURCE MANAGEMENT' (Tags, Locks, Users, Automation script).

On the SendGrid page, click **Settings** > **API Keys** > **Create API Key**.

The screenshot shows the SendGrid 'API Keys' page. The left sidebar has a 'Door Monitor' header and links for Dashboard, Marketing, Templates, Stats, Activity, Suppressions, Settings (with 'API Keys' highlighted by a red box), Account Details, Alert Settings, and API Keys. The main content area is titled 'API Keys' and features a 'Create API Key' button (also highlighted by a red box). Below it is a section titled 'Get started creating API Keys' with a key icon and a description: 'API keys help protect the sensitive areas of your SendGrid account (e.g. contacts and account settings). To control and limit access of API users, you can create multiple API keys, each with different permissions.'

On the **Create API Key** page, input the **API Key Name** and click **Create & View**.

Create API Key

API Key Name • (i)

API Key Permissions • (i)

 **Full Access**
Allows the API key to access GET, PATCH, PUT, DELETE, and POST endpoints for all parts of your account, excluding billing.

 **Restricted Access**
Customize levels of access for all parts of your account, excluding billing.

 **Billing Access**
Allows the API key to access billing endpoints for the account. (This is especially useful for Enterprise or Partner customers looking for more advanced account management.)

Cancel Create & View

Your API key is displayed only one time. Be sure to copy and store it safely, as it is used in the next step.

Deploy IoT Hub in Azure

The following steps will provision other Azure IoT related services and deploy Azure Functions for this project.

Click the **Deploy to Azure** button below.



The sign-up form appears.

Custom deployment

Deploy from a custom template

BASICS

* Subscription

* Resource group Create new Use existing

* Location

SETTINGS

* IoT Hub Name

IoT Hub Sku

IoT Hub Units

IoT Hub Partitions

* Send Grid Api Key

* To Email

* From Email

Repo Url

Pin to dashboard

Purchase

Fill in the fields on the sign-up form.

- **Resource group:** Create a resource group to host the SendGrid service, or use an existing one. See [Using resource groups to manage your Azure resources](#).
- **IoT Hub Name:** The name for your IoT hub. Choose a unique name, differing from other services you may have.
- **IoT Hub Sku:** F1 (limited to one per subscription) is free. You can see more pricing information on the [pricing page](#).
- **From Email:** This field should be the same email address you used when setting up the SendGrid service.

Check the **Pin to dashboard** option to make this application easier to find in the future, then click **Purchase** when you're ready to continue to the next step.

Build and upload the code

Next, load the sample code in VS Code and provision the necessary Azure services.

Start VS Code

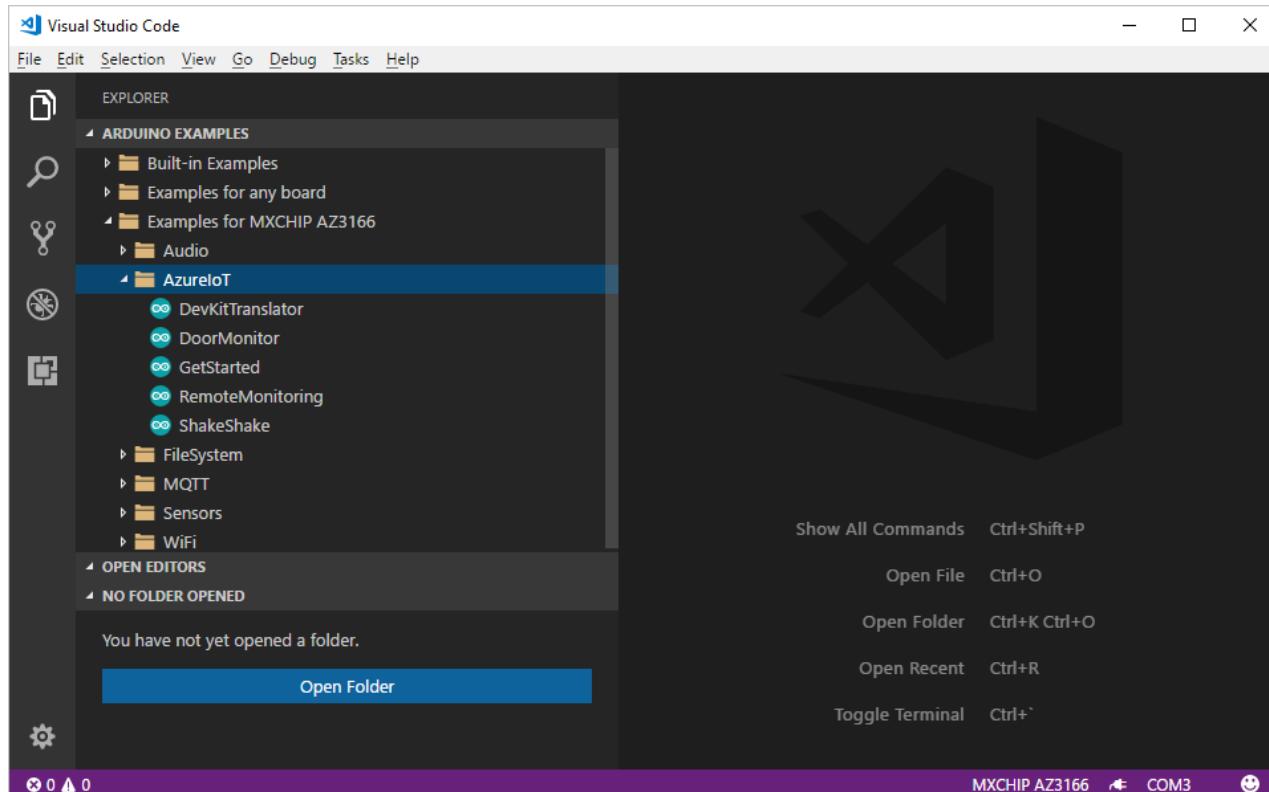
- Make sure your DevKit is **not** connected to your computer.
- Start VS Code.
- Connect the DevKit to your computer.

NOTE

When you launch VS Code, you may receive an error message stating that it cannot find the Arduino IDE or related board package. If you receive this error, close VS Code, launch the Arduino IDE again, and VS Code should locate the Arduino IDE path correctly.

Open Arduino Examples folder

Expand the left side **ARDUINO EXAMPLES** section, browse to **Examples for MXCHIP AZ3166 > AzureIoT**, and select **DoorMonitor**. This action opens a new VS Code window with a project folder in it.



You can also open the example app from the command palette. Use `Ctrl+Shift+P` (macOS: `Cmd+Shift+P`) to open the command palette, type **Arduino**, and then find and select **Arduino: Examples**.

Provision Azure services

In the solution window, run the cloud provisioning task:

- Type `Ctrl+P` (macOS: `Cmd+P`).
- Enter `task cloud-provision` in the provided text box.

In the VS Code terminal, an interactive command line guides you through provisioning the required Azure services. Select all of the same items from the prompted list that you previously provisioned in [Deploy IoT Hub in Azure](#).

```
// Copyright (c) Microsoft. All rights reserved.
// Licensed under the MIT license.
// To get started please visit https://microsoft.github.io/azure-iot-devkit
#include "AZ3166WiFi.h"
#include "AzureIotHub.h"
#include "DevKitMQTTClient.h"
#include "LIS2MDLSensor.h"
#include "OledDisplay.h"

#define APP_VERSION      "ver=1.0"
#define LOOP_DELAY       1000
#define EXPECTED_COUNT   5

- checking pre-conditions of task: node.js version
V node.js version: v6.10.2
- checking pre-conditions of task: azure cli version
V azure cli version: 2.0.9
- checking pre-conditions of task: subscription
```

NOTE

If the page hangs in the loading status when trying to sign in to Azure, refer to the "["page hangs when logging in"](#) section of the [IoT DevKit FAQ](#) to resolve this issue.

Build and upload the device code

Next, upload the code for the device.

Windows

1. Use `Ctrl+P` to run `task device-upload`.
2. The terminal prompts you to enter configuration mode. To do so, hold down button A, then push and release the reset button. The screen displays the DevKit identification number and the word *Configuration*.

macOS

1. Put the DevKit into configuration mode: Hold down button A, then push and release the reset button. The screen displays 'Configuration'.
2. Click `Cmd+P` to run `task device-upload`.

Verify, upload, and run the sample app

The connection string that is retrieved from the [Provision Azure services](#) step is now set.

VS Code then starts verifying and uploading the Arduino sketch to the DevKit.

DoorMonitor.ino - DoorMonitor - Visual Studio Code

File Edit Selection View Go Debug Tasks Help

EXPLORER Welcome DoorMonitor.ino

```

1 // Copyright (c) Microsoft. All rights reserved.
2 // Licensed under the MIT license.
3 // To get started please visit https://microsoft.github.io/azure-iot-devkit/
4 #include "AZ3166WiFi.h"
5 #include "AzureIotHub.h"
6 #include "DevKitMQTTClient.h"
7 #include "LIS2MDLSensor.h"
8 #include "OledDisplay.h"
9
10 #define APP_VERSION      "ver=1.0"
11 #define LOOP_DELAY        1000
12 #define EXPECTED_COUNT   5
13

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: Task - device ▾ + ⌂ ⌄ ⌁ ×

V Check Arduino IDE Version and Location: 1.8.1 @ C:\Program Files (x86)\Arduino
- checking pre-conditions of task: Check Arduino Board
V Check Arduino Board: MXCHIP_AZ3166 as MXCHIP AZ3166
- checking pre-conditions of task: Build & Upload Sketch
C:\Program Files (x86)\Arduino\arduino_debug.exe --upload --board AZ3166:stm32f4:MXCHIP_AZ3166 -
-preferences-file d:\VS IoT\temp\DoorMonitor\.build/pref.txt --pref compiler.warning_level:none -
-pref build.path=d:\VS IoT\temp\DoorMonitor\.build d:\VS IoT\temp\DoorMonitor\DoorMonitor.ino
Loading configuration...
Initializing packages...
Preparing boards...
Verifying...

ARDUINO EXAMPLES

Ln 18, Col 33 Spaces: 2 UTF-8 CRLF C++ MXCHIP AZ3166 COM3 Win32 ☺

The DevKit reboots and starts running the code.

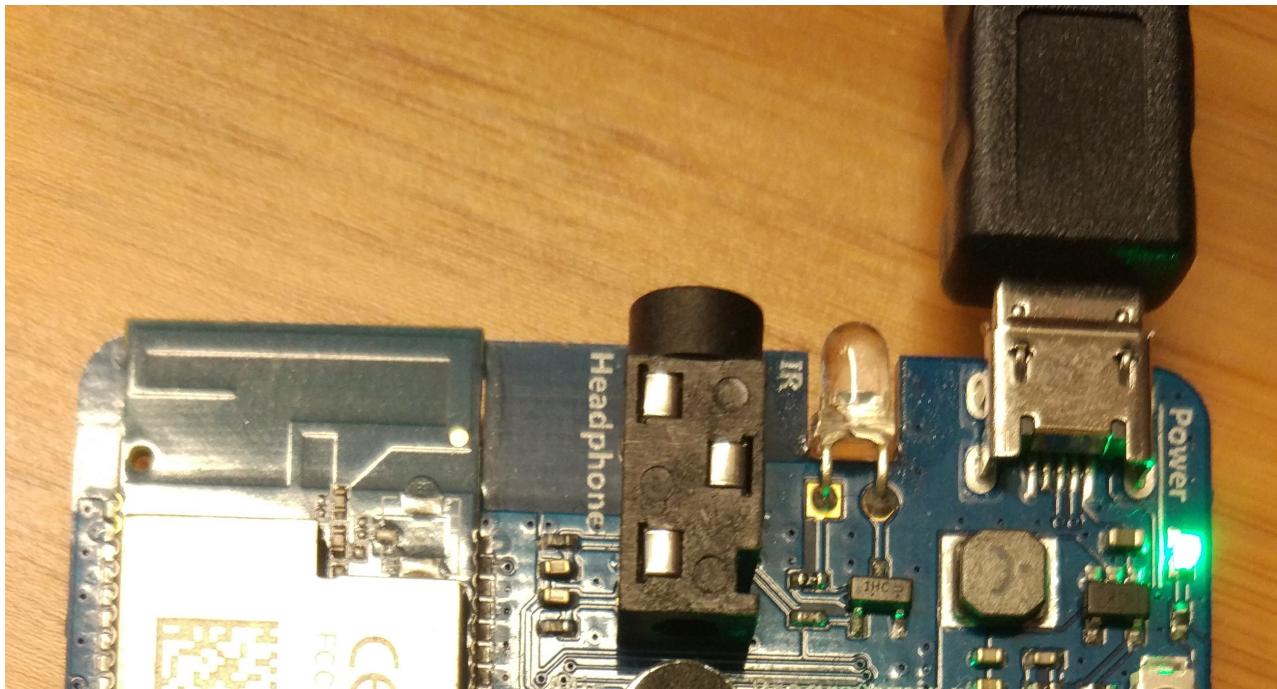
NOTE

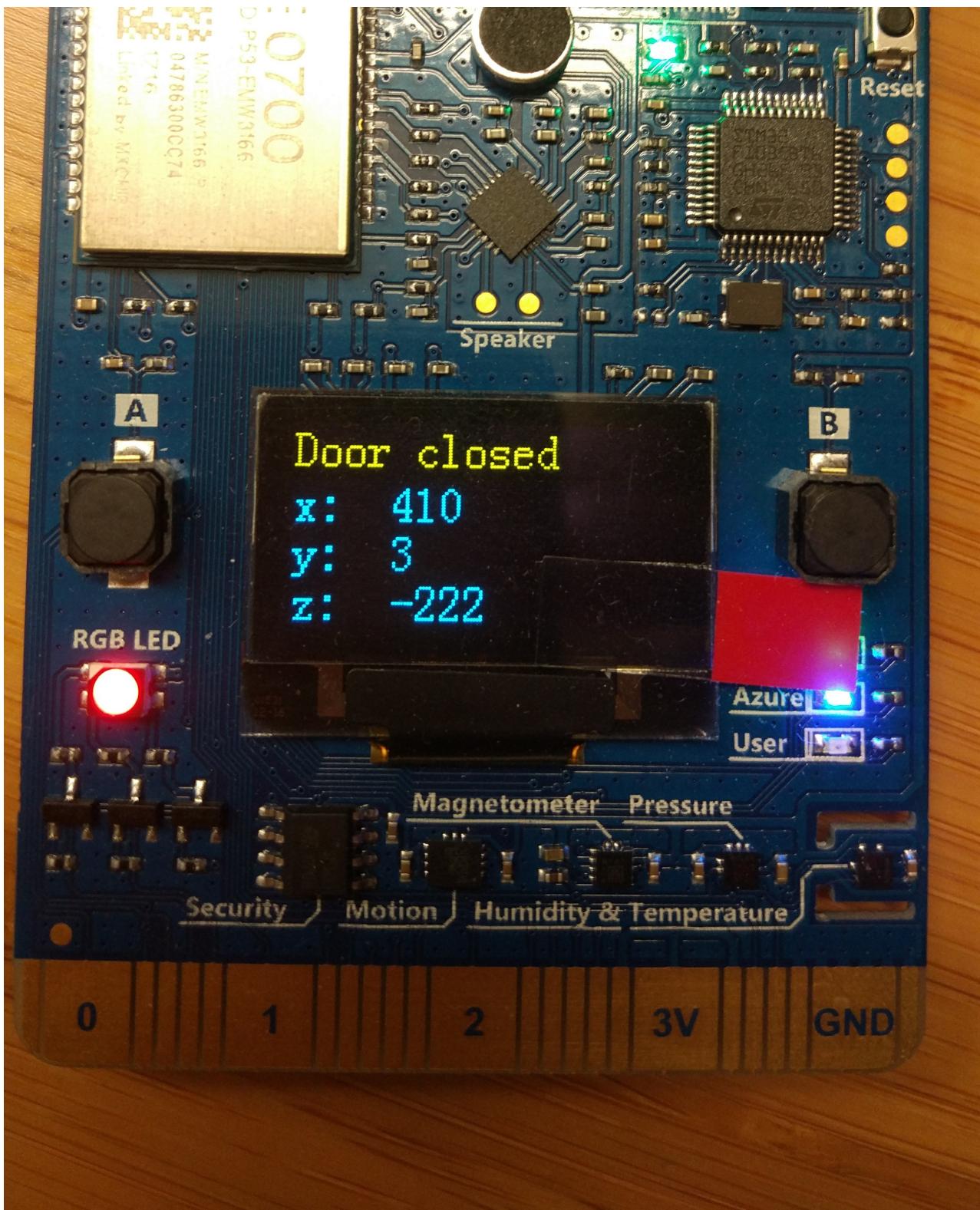
Occasionally, you may receive an "Error: AZ3166: Unknown package" error message. This error occurs when the board package index is not refreshed correctly. To resolve this error, refer to the [development section of the IoT DevKit FAQ](#).

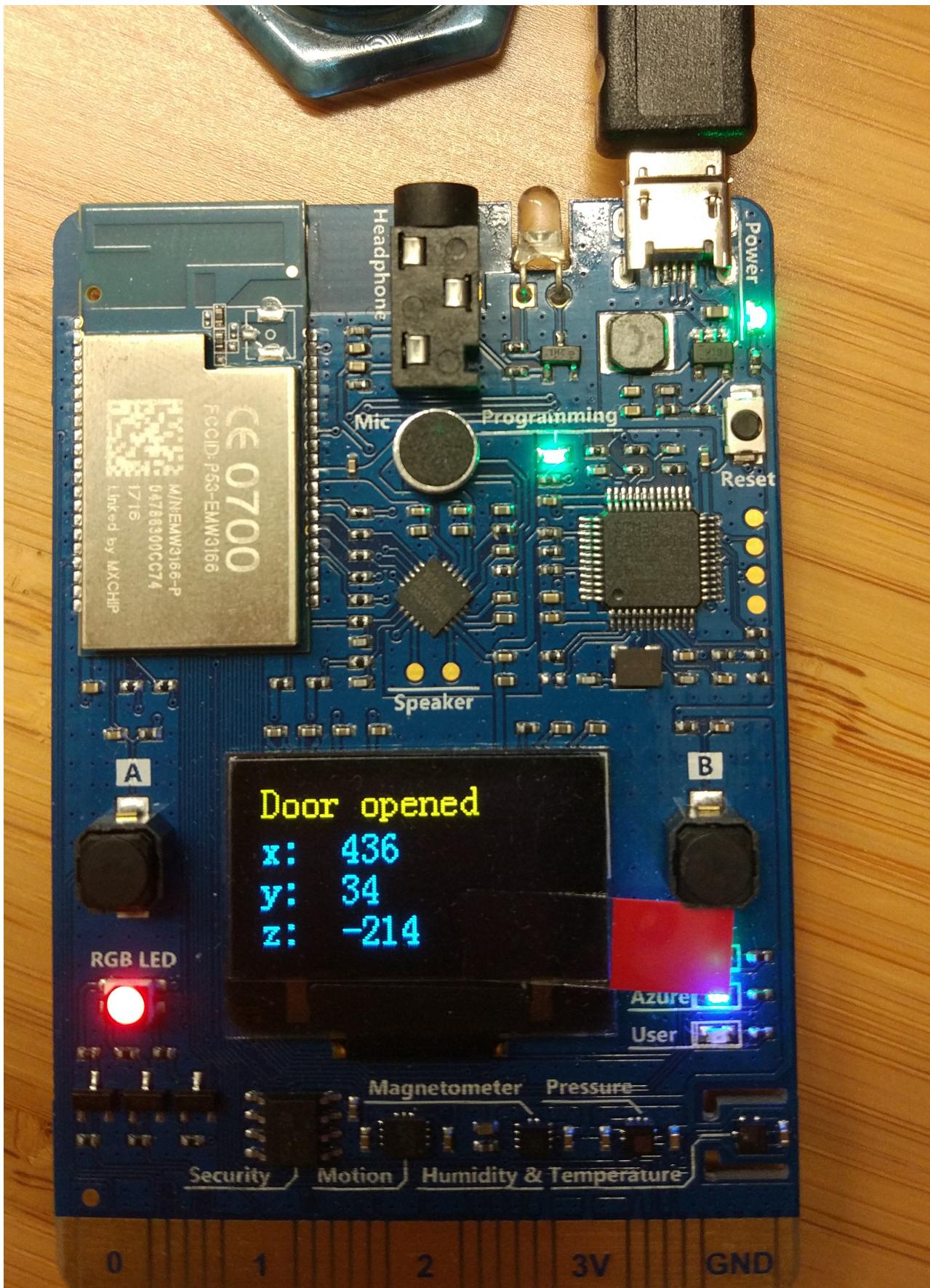
Test the project

The program first initializes when the DevKit is in the presence of a stable magnetic field.

After initialization, `Door closed` is displayed on the screen. When there is a change in the magnetic field, the state changes to `Door opened`. Each time the door state changes, you receive an email notification. (These email messages may take up to five minutes to be received.)







Problems and feedback

If you encounter problems, refer to the [IoT DevKit FAQ](#) or connect using the following channels:

- [Gitter.im](#)
- [Stack Overflow](#)

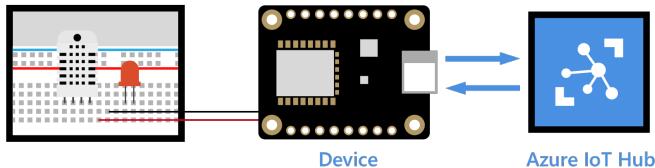
Next steps

You have learned how to connect a DevKit device to your Azure IoT Remote Monitoring solution accelerator and used the SendGrid service to send an email. Here are the suggested next steps:

- [Azure IoT Remote Monitoring solution accelerator overview](#)
- [Connect an MXChip IoT DevKit device to your Azure IoT Central application](#)

Use Azure IoT Tools for Visual Studio Code to send and receive messages between your device and IoT Hub

2/22/2019 • 2 minutes to read



Azure IoT Tools is a useful Visual Studio Code extension that makes IoT Hub management and IoT application development easier. This article focuses on how to use Azure IoT Tools for Visual Studio Code to send and receive messages between your device and your IoT hub.

NOTE

Some of the features mentioned in this article, like cloud-to-device messaging, device twins, and device management, are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

NOTE

This article has been updated to use the new Azure PowerShell Az module. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For installation instructions, see [Install Azure PowerShell](#).

What you will learn

You learn how to use Azure IoT Tools for Visual Studio Code to monitor device-to-cloud messages and to send cloud-to-device messages. Device-to-cloud messages could be sensor data that your device collects and then sends to your IoT hub. Cloud-to-device messages could be commands that your IoT hub sends to your device to blink an LED that is connected to your device.

What you will do

- Use Azure IoT Tools for Visual Studio Code to monitor device-to-cloud messages.
- Use Azure IoT Tools for Visual Studio Code to send cloud-to-device messages.

What you need

- An active Azure subscription.
- An Azure IoT hub under your subscription.

- [Visual Studio Code](#)
- [Azure IoT Tools for VS Code](#)

Sign in to access your IoT hub

1. In **Explorer** view of VS Code, expand **Azure IoT Hub Devices** section in the bottom left corner.
2. Click **Select IoT Hub** in context menu.
3. A pop-up will show in the bottom right corner to let you sign in to Azure for the first time.
4. After you sign in, your Azure Subscription list will be shown, then select Azure Subscription and IoT Hub.
5. The device list will be shown in **Azure IoT Hub Devices** tab in a few seconds.

NOTE

You can also complete the set up by choosing **Set IoT Hub Connection String**. Enter the connection string for the IoT hub that your IoT device connects to in the pop-up window.

Monitor device-to-cloud messages

To monitor messages that are sent from your device to your IoT hub, follow these steps:

1. Right-click your device and select **Start Monitoring D2C Message**.
2. The monitored messages will be shown in **OUTPUT > Azure IoT Hub Toolkit** view.
3. To stop monitoring, right-click the **OUTPUT** view and select **Stop Monitoring D2C Message**.

Send cloud-to-device messages

To send a message from your IoT hub to your device, follow these steps:

1. Right-click your device and select **Send C2D Message to Device**.
2. Enter the message in input box.
3. Results will be shown in **OUTPUT > Azure IoT Hub Toolkit** view.

Next steps

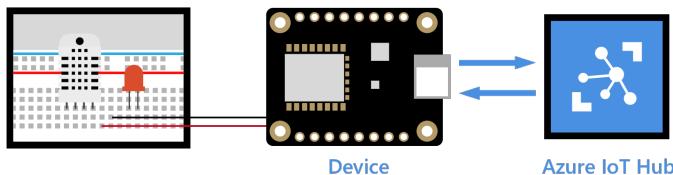
You've learned how to monitor device-to-cloud messages and send cloud-to-device messages between your IoT device and Azure IoT Hub.

To continue to get started with Azure IoT Hub and to explore all extended IoT scenarios, see the following:

- [Manage cloud device messaging with Azure IoT Hub Toolkit extension for Visual Studio Code](#)
- [Manage devices with Azure IoT Hub Toolkit extension for Visual Studio Code](#)
- [Set up message routing](#)
- [Use Power BI to visualize real-time sensor data from your IoT hub](#)
- [Use the Web Apps feature of Azure App Service to visualize real-time sensor data from your IoT hub](#)
- [Forecast weather by using the sensor data from your IoT hub in Azure Machine Learning](#)
- [Use Logic Apps for remote monitoring and notifications](#)

Use Cloud Explorer for Visual Studio to send and receive messages between your device and IoT Hub

1/8/2019 • 2 minutes to read



[Cloud Explorer](#) is a useful Visual Studio extension that enables you to view your Azure resources, inspect their properties and perform key developer actions from within Visual Studio. This article focuses on how to use Cloud Explorer to send and receive messages between your device and your IoT Hub.

NOTE

Some of the features mentioned in this article, like cloud-to-device messaging, device twins, and device management, are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

What you will learn

You will learn how to use Cloud Explorer for Visual Studio to monitor device-to-cloud messages and to send cloud-to-device messages. Device-to-cloud messages could be sensor data that your device collects and then sends to your IoT Hub. Cloud-to-device messages could be commands that your IoT Hub sends to your device. For example, blink an LED that is connected to your device.

What you will do

- Use Cloud Explorer for Visual Studio to monitor device-to-cloud messages.
- Use Cloud Explorer for Visual Studio to send cloud-to-device messages.

What you need

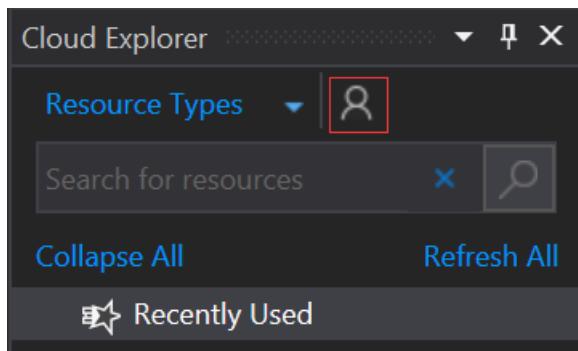
- An active Azure subscription.
- An Azure IoT Hub under your subscription.
- Microsoft Visual Studio 2017 Update 8 or later
- Cloud Explorer component from Visual Studio Installer (selected by default with Azure Workload)

Update Cloud Explorer to latest version

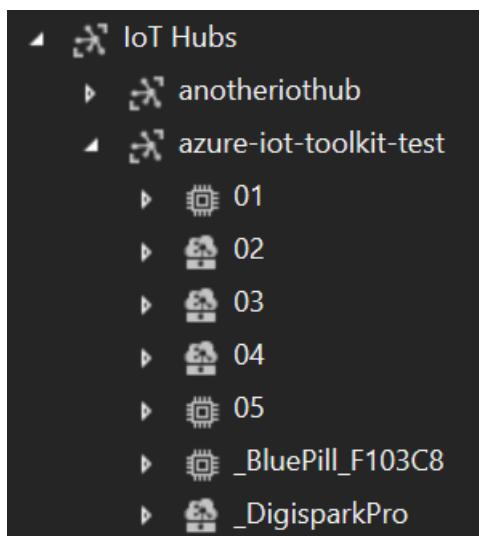
The Cloud Explorer component from Visual Studio Installer only supports monitoring device-to-cloud and cloud-to-device messages. In order to send messages to device or cloud, you download and install the latest [Cloud Explorer](#).

Sign in to access your IoT Hub

1. In Visual Studio **Cloud Explorer** window, click the Account Management icon. You can open the Cloud Explorer window from **View > Cloud Explorer** menu.



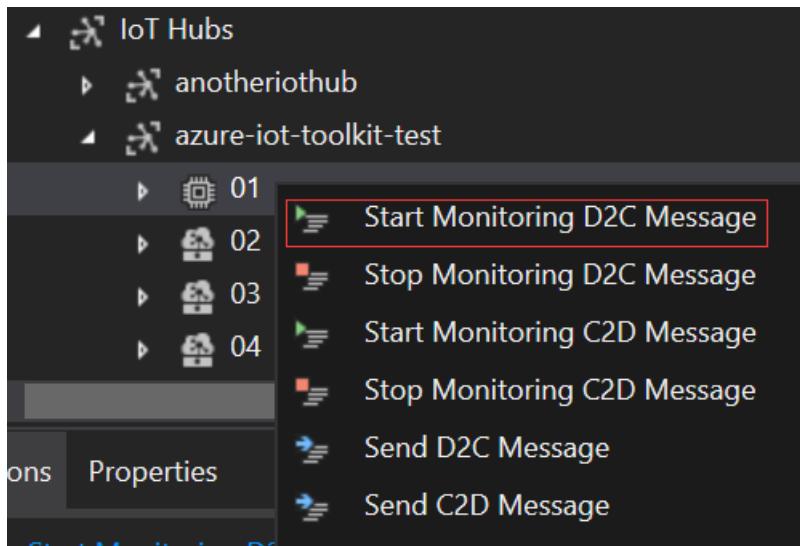
2. Click **Manage Accounts** in Cloud Explorer.
3. Click **Add an account...** in the new window to sign in to Azure for the first time.
4. After you sign in, your Azure subscription list will be shown. Select the Azure subscriptions you want to view and click **Apply**.
5. Expand **Your subscription > IoT Hubs > Your IoT Hub**, the device list will be shown under your IoT Hub node.



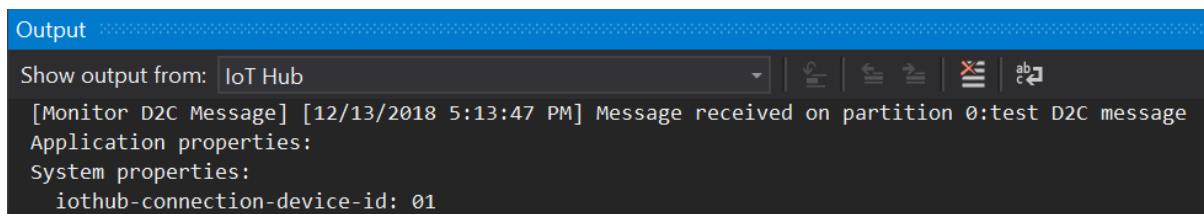
Monitor device-to-cloud messages

To monitor messages that are sent from your device to your IoT Hub, follow these steps:

1. Right-click your IoT Hub or device and select **Start Monitoring D2C Message**.



2. The monitored messages will be shown in the **IoT Hub** output pane.

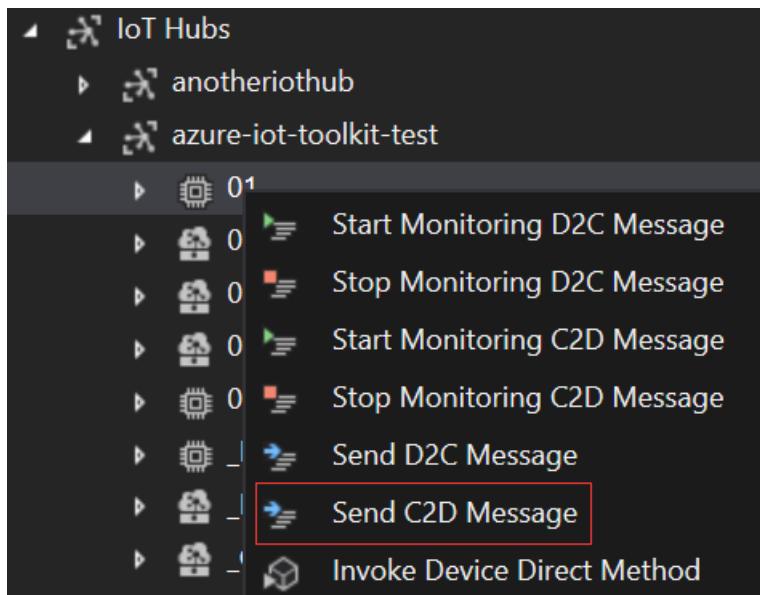


3. To stop monitoring, right-click on any IoT Hub or device and select **Stop Monitoring D2C Message**.

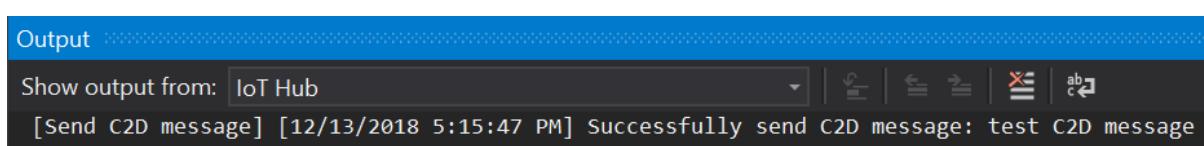
Send cloud-to-device messages

To send a message from your IoT Hub to your device, follow these steps:

1. Right-click your device and select **Send C2D Message**.



2. Enter the message in input box.
3. Results will be shown in the **IoT Hub** output pane.



Next steps

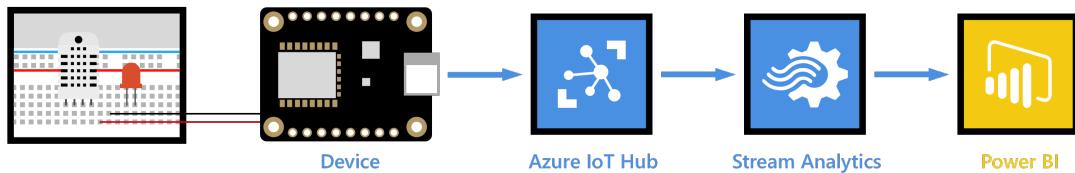
You've learned how to monitor device-to-cloud messages and send cloud-to-device messages between your IoT device and Azure IoT Hub.

To continue to get started with Azure IoT Hub and to explore all extended IoT scenarios, see the following:

- [Manage cloud device messaging with Azure IoT Hub Toolkit extension for Visual Studio Code](#)
- [Manage devices with Azure IoT Hub Toolkit extension for Visual Studio Code](#)
- [Set up message routing](#)
- [Use Power BI to visualize real-time sensor data from your IoT hub](#)
- [Use the Web Apps feature of Azure App Service to visualize real-time sensor data from your IoT hub](#)
- [Forecast weather by using the sensor data from your IoT hub in Azure Machine Learning](#)
- [Use Logic Apps for remote monitoring and notifications](#)

Visualize real-time sensor data from Azure IoT Hub using Power BI

1/18/2019 • 5 minutes to read



NOTE

Before you start this tutorial, [set up your device](#). In the article, you set up your Azure IoT device and IoT hub, and you deploy a sample application to run on your device. The application sends collected sensor data to your IoT hub.

What you learn

You learn how to visualize real-time sensor data that your Azure IoT hub receives by using Power BI. If you want to try to visualize the data in your IoT hub with Web Apps, please see [Use Azure Web Apps to visualize real-time sensor data from Azure IoT Hub](#).

What you do

- Get your IoT hub ready for data access by adding a consumer group.
- Create, configure, and run a Stream Analytics job for data transfer from your IoT hub to your Power BI account.
- Create and publish a Power BI report to visualize the data.

What you need

- Tutorial [Setup your device](#) completed which covers the following requirements:
 - An active Azure subscription.
 - An Azure IoT hub under your subscription.
 - A client application that sends messages to your Azure IoT hub.
- A Power BI account. ([Try Power BI for free](#))

Add a consumer group to your IoT hub

Consumer groups are used by applications to pull data from Azure IoT Hub. In this tutorial, you create a consumer group to be used by a coming Azure service to read data from your IoT hub.

To add a consumer group to your IoT hub, follow these steps:

1. In the [Azure portal](#), open your IoT hub.
2. In the left pane, click **Built-in endpoints**, select **Events** on the top pane, and enter a name under **Consumer groups** on the right pane. Click **Save** after you change the **Default TTL** value and return it back to the original value.

The screenshot shows the Azure portal interface for managing an IoT hub. The left sidebar has 'Resource groups' selected. The main pane shows 'testhub2 - Built-in endpoints'. Under 'Events', there is a consumer group named 'consumer2'. The 'Default TTL' is set to 1 day. Other settings like 'Event Hub-compatible name' (testhub2) and 'Event hub-compatible endpoint' (Endpoint=sb://iothub-ns-evan-test-1173979-8f37235e38.servicebus.windows.net/SharedAccessKeyName=iothubownerSharedAccessKey=...) are also visible.

Create, configure, and run a Stream Analytics job

Let's start by creating a Stream Analytics job. After you create the job, you define the inputs, outputs, and the query used to retrieve the data.

Create a Stream Analytics job

1. In the [Azure portal](#), click **Create a resource** > **Internet of Things** > **Stream Analytics job**.
2. Enter the following information for the job.

Job name: The name of the job. The name must be globally unique.

Resource group: Use the same resource group that your IoT hub uses.

Location: Use the same location as your resource group.

Pin to dashboard: Check this option for easy access to your IoT hub from the dashboard.

The screenshot shows the Azure portal interface for creating a new Stream Analytics job. The left sidebar has 'Resource groups' selected. The main pane shows a 'New Stream Analytics job' dialog. It includes fields for 'Job name' (New Stream Analytics job), 'Subscription' (Internal use), 'Resource group' (resource2), 'Location' (West US), and 'Streaming units' (1 to 120) (3). The 'Cloud' option is selected for the 'Hosting environment'.

3. Click **Create**.

Add an input to the Stream Analytics job

1. Open the Stream Analytics job.
2. Under **Job Topology**, click **Inputs**.
3. In the **Inputs** pane, click **Add stream input**, and then enter the following information:

Input alias: The unique alias for the input and select **Provide IoT Hub settings manually** below.

Source: Select **IoT hub**.

Endpoint: Click **Messaging**.

Consumer group: Select the consumer group you just created.

4. Click **Create**.

The screenshot shows the Azure Stream Analytics job 'streamanalysis2 - Inputs' interface. On the left, the navigation menu includes 'Create a resource', 'Home', 'Dashboard', 'All services', 'FAVORITES' (with 'All resources' selected), 'Resource groups', 'App Services', 'Function Apps', 'SQL databases', 'Azure Cosmos DB', 'Virtual machines', 'Load balancers', 'Storage accounts', 'Virtual networks', 'Azure Active Directory', 'Monitor', 'Advisor', 'Security Center', 'Cost Management + Bill...', and 'Help + support'. The main area shows the 'Inputs' blade with a table:

NAME	SOURCE TYPE	SOURCE
Stream	Stream	IoT Hub

To the right, the 'Input details' pane contains the following fields:

- Input alias:** Stream
- Source type:** Stream
- Source:** IoT Hub
- Endpoint:** Messaging
- Shared access policy name:** iothubowner
- Shared access policy key:** (redacted)

A note at the bottom right of the pane states: "If the chosen resource and the stream analytics job are located in different regions, you will be billed to move data between regions."

Add an output to the Stream Analytics job

1. Under **Job Topology**, click **Outputs**.
2. In the **Outputs** pane, click **Add** and **Power BI**, and then enter the following information:

Output alias: The unique alias for the output.

Group Workspace: Select your target group workspace.

Dataset Name: Enter a dataset name.

Table Name: Enter a table name.

3. Click **Authorize**, and then sign into your Power BI account.
4. Click **Create**.

Configure the query of the Stream Analytics job

1. Under **Job Topology**, click **Query**.
2. Replace **[YourInputAlias]** with the input alias of the job.
3. Replace **[YourOutputAlias]** with the output alias of the job.
4. Click **Save**.

Run the Stream Analytics job

In the Stream Analytics job, click **Start > Now > Start**. Once the job successfully starts, the job status changes from **Stopped** to **Running**.

The screenshot shows the Azure Stream Analytics job configuration interface. The job name is 'streamanalysis2'. It is currently running. There is one input source named 'alskjac123' and one output target named 'Output'. The query is defined as:

```

1 SELECT *
2 *
3 INTO Output
4
5 FROM alskjac123

```

Create and publish a Power BI report to visualize the data

1. Ensure the sample application is running on your device. If not, you can refer to the tutorials under [Setup your device](#).
2. Sign in to your [Power BI](#) account.
3. Click the workspace you used, **My Workspace**.
4. Click **Datasets**.

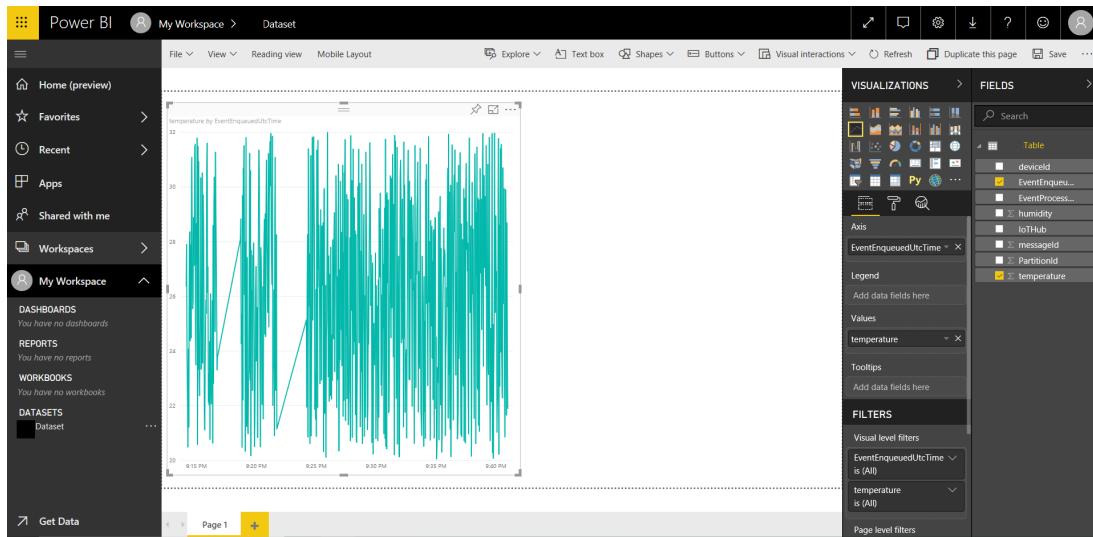
You should see the dataset that you specified when you created the output for the Stream Analytics job.

5. For the dataset you created, click **Add Report** (the first icon to the right of the dataset name).

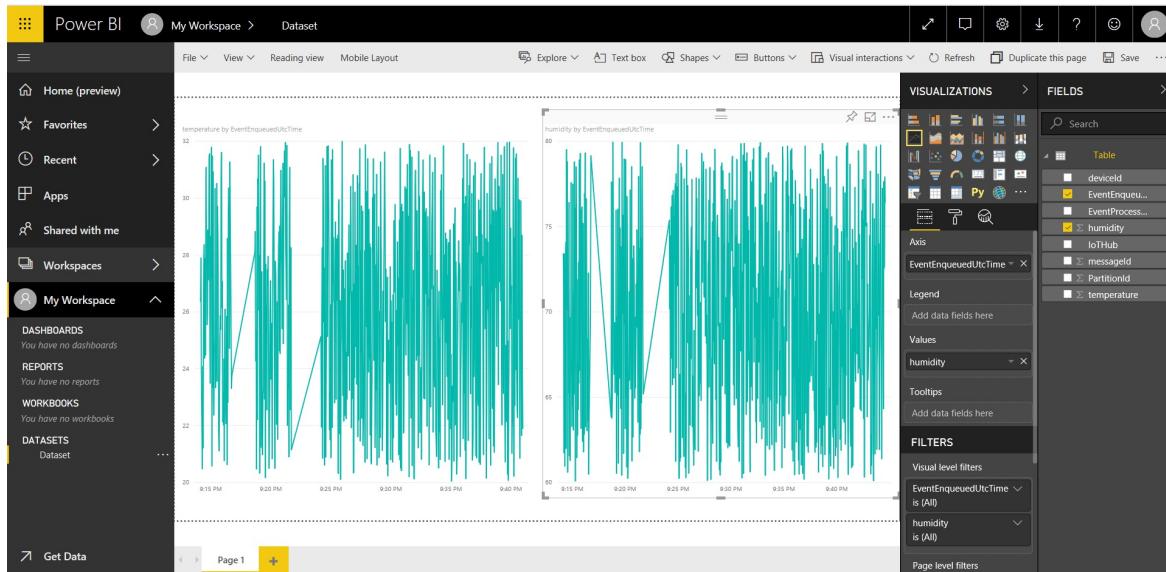
The screenshot shows the Power BI desktop application. On the left, the navigation pane shows 'My Workspace' selected. In the center, the 'Dataset' view is open. The 'Fields' pane on the right lists the fields from the Stream Analytics output: deviceId, EventEnqueuedUtcTime, EventProcessedUtcTime, humidity, IoTHub, messageId, PartitionId, and temperature. The 'Visualizations' pane on the right contains various chart and report options.

6. Create a line chart to show real-time temperature over time.
 - a. On the report creation page, add a line chart.
 - b. On the **Fields** pane, expand the table that you specified when you created the output for the Stream Analytics job.
 - c. Drag **EventEnqueuedUtcTime** to **Axis** on the **Visualizations** pane.
 - d. Drag **temperature** to **Values**.

A line chart is created. The x-axis displays date and time in the UTC time zone. The y-axis displays temperature from the sensor.

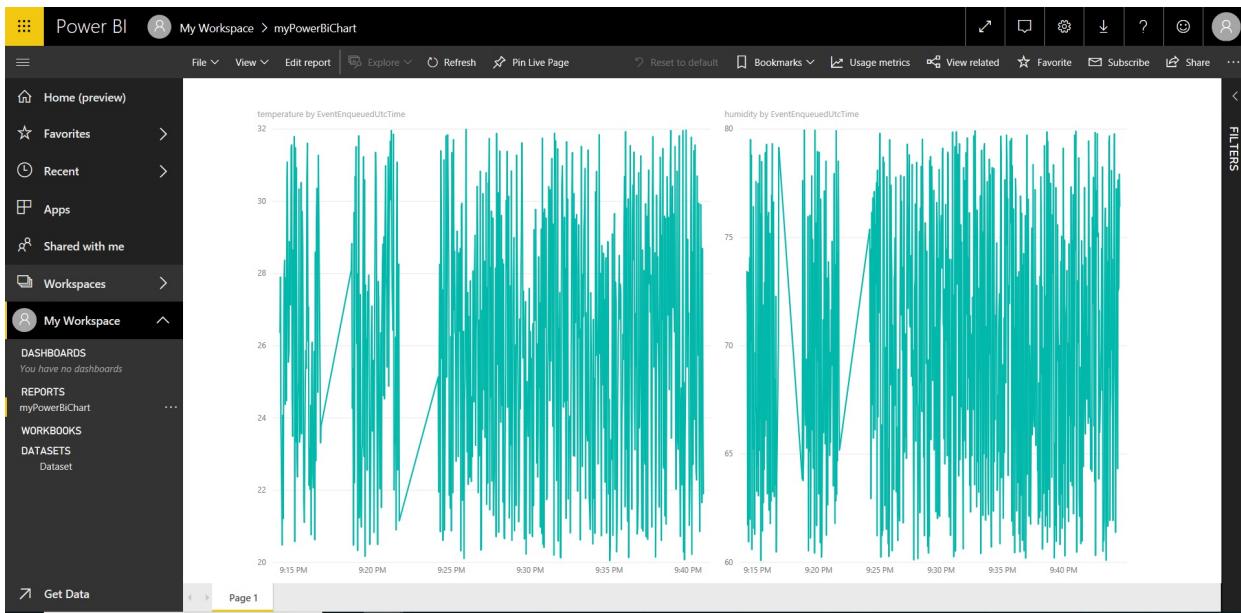


7. Create another line chart to show real-time humidity over time. To do this, follow the same steps above and place **EventEnqueuedUtcTime** on the x-axis and **humidity** on the y-axis.



8. Click **Save** to save the report.
9. Click **Reports** on the left pane, and then click the report that you just created.
10. Click **File > Publish to web**.
11. Click **Create embed code**, and then click **Publish**.

You're provided the report link that you can share with anyone for report access and a code snippet to integrate the report into your blog or website.



Microsoft also offers the [Power BI mobile apps](#) for viewing and interacting with your Power BI dashboards and reports on your mobile device.

Next steps

You've successfully used Power BI to visualize real-time sensor data from your Azure IoT hub.

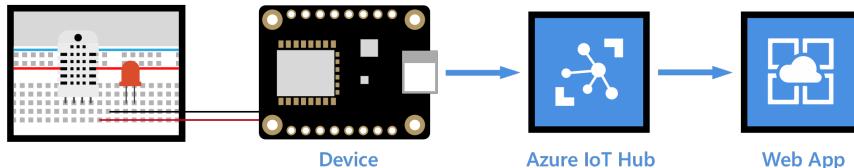
There is an alternate way to visualize data from Azure IoT Hub. See [Use Azure Web Apps to visualize real-time sensor data from Azure IoT Hub](#).

To continue to get started with Azure IoT Hub and to explore all extended IoT scenarios, see the following:

- [Manage cloud device messaging with Azure IoT Hub Toolkit extension for Visual Studio Code](#)
- [Manage devices with Azure IoT Hub Toolkit extension for Visual Studio Code](#)
- [Set up message routing](#)
- [Use Power BI to visualize real-time sensor data from your IoT hub](#)
- [Use the Web Apps feature of Azure App Service to visualize real-time sensor data from your IoT hub](#)
- [Forecast weather by using the sensor data from your IoT hub in Azure Machine Learning](#)
- [Use Logic Apps for remote monitoring and notifications](#)

Visualize real-time sensor data from your Azure IoT hub by using the Web Apps feature of Azure App Service

10/26/2018 • 4 minutes to read



NOTE

Before you start this tutorial, [set up your device](#). In the article, you set up your Azure IoT device and IoT hub, and you deploy a sample application to run on your device. The application sends collected sensor data to your IoT hub.

What you learn

In this tutorial, you learn how to visualize real-time sensor data that your IoT hub receives by running a web application that is hosted on a web app. If you want to try to visualize the data in your IoT hub by using Power BI, see [Use Power BI to visualize real-time sensor data from Azure IoT Hub](#).

What you do

- Create a web app in the Azure portal.
- Get your IoT hub ready for data access by adding a consumer group.
- Configure the web app to read sensor data from your IoT hub.
- Upload a web application to be hosted by the web app.
- Open the web app to see real-time temperature and humidity data from your IoT hub.

What you need

- [Set up your device](#), which covers the following requirements:
 - An active Azure subscription
 - An IoT hub under your subscription
 - A client application that sends messages to your IoT hub
- [Download Git](#)

Create a web app

1. In the [Azure portal](#), click **Create a resource > Web + Mobile > Web App**.
2. Enter a unique job name, verify the subscription, specify a resource group and a location, select **Pin to**

dashboard, and then click **Create**.

We recommend that you select the same location as your resource group.

The screenshot shows the Microsoft Azure portal interface. In the top navigation bar, it says "Microsoft Azure New > Web + Mobile > Web App". There are three main panes: a left sidebar with a "New" section containing a search bar and a "MARKETPLACE" section with various service categories; a middle pane titled "Web + Mobile" showing "FEATURED APPS" with icons and descriptions for "Web App", "Mobile App", "Logic App", "Web App On Linux (preview)", "CDN", and "Media Services"; and a right pane titled "Web App" where the user is creating a new web app named "iot-sample.azurewebsites.net" under the "VSChina IoT Prod" subscription and "iot-sample" resource group. The "App Service plan/Location" dropdown shows "ServicePlan2375834b-af33(South ...)". The "Application Insights" toggle is set to "Off". At the bottom of the right pane are "Create" and "Automation options" buttons.

Add a consumer group to your IoT hub

Consumer groups are used by applications to pull data from Azure IoT Hub. In this tutorial, you create a consumer group to be used by a coming Azure service to read data from your IoT hub.

To add a consumer group to your IoT hub, follow these steps:

1. In the [Azure portal](#), open your IoT hub.
2. In the left pane, click **Built-in endpoints**, select **Events** on the top pane, and enter a name under **Consumer groups** on the right pane. Click **Save** after you change the **Default TTL** value and return it back to the original value.

The screenshot shows the Microsoft Azure portal interface. In the top navigation bar, it says "Microsoft Azure Preview". The left sidebar shows "All services" and "IoT hub". The main area is titled "testhub2 - Built-in endpoints". It has several sections:

- Events**: Shows "Events is the default endpoint, and is used until custom routing rules are created." with a "Partitions" dropdown set to 4, "Event hub-compatible name" as "testhub2", "Event hub-compatible endpoint" as "Endpoint=sb://iothub-test-avan-testh-1173979-8f37235e38.servicebus.windows.net/SharedAccessKeyName=iothubownerSharedAccessKey=...", and "Retain for" set to 1 Day.
- CONSUMER GROUPS**: Shows a list with "Default" and "consumer2". A button "Create new consumer group" is available.
- Cloud to device messaging**: Shows "Control message retention time and retry attempts." with "Default TTL" set to 1 Hours.

Configure the web app to read data from your IoT hub

1. Open the web app you've just provisioned.
2. Click **Application settings**, and then, under **App settings**, add the following key/value pairs:

KEY	VALUE
Azure.IoT.IoTHub.ConnectionString	Obtained from Azure CLI
Azure.IoT.IoTHub.ConsumerGroup	The name of the consumer group that you add to your IoT hub
WEBSITE_NODE_DEFAULT_VERSION	8.9.4

Microsoft Azure xshi - Application settings

App Service

settings

Overview

SETTINGS

Application settings

Authentication / Authorization

Backups

Custom domains

SSL certificates

Networking

Scale up (App Service plan)

Scale out (App Service plan)

Security scanning

WebJobs

Push

MySQL In App (preview)

Properties

Locks

Save Discard

Debugging

Remote debugging Off On

Remote Visual Studio version 2012 2013 2015

App settings

WEBSITE_NODE_DEFAULT_VERSION 6.9.1 Slot setting ...

Azure.IoT.IoTHub.ConnectionString Slot setting ...

Azure.IoT.IoTHub.ConsumerGroup Slot setting ...

Key Value Slot setting ...

Connection strings

No results

Name Value SQL Database Slot setting ...

Default documents

Default.htm Slot setting ...

Default.html Slot setting ...

3. Click **Application settings**, under **General settings**, toggle the **Web sockets** option, and then click **Save**.

Microsoft Azure Resource groups > iot-sample > xshi - Application settings

Report a bug Search resources

Continuous Delivery (Preview)

SETTINGS

Application settings

Authentication / Authorization

Backups

Custom domains

SSL certificates

Networking

Scale up (App Service plan)

Scale out (App Service plan)

Save Discard

General settings

.NET Framework version v4.6

PHP version 5.6

Java version Off

Python version Off

Platform 32-bit 64-bit

Web sockets Off On

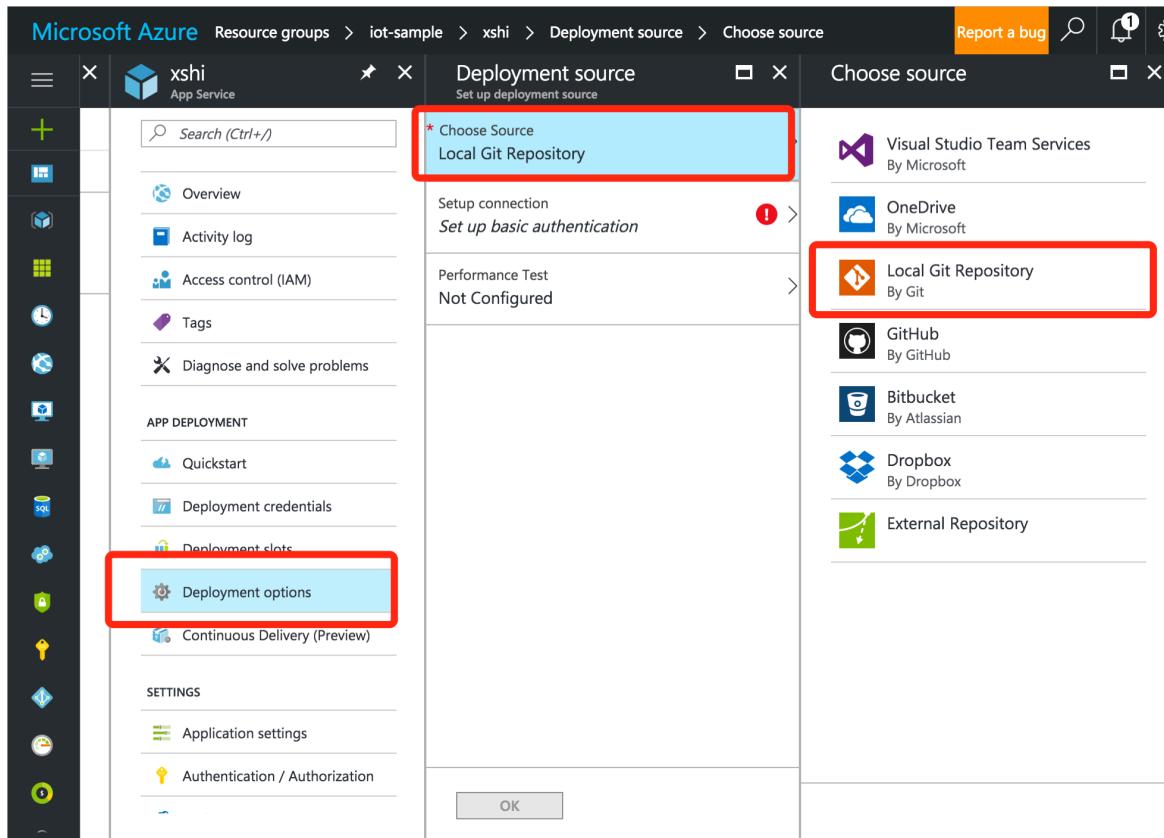
Always On Off On

Managed Pipeline Version Integrated Classic

Upload a web application to be hosted by the web app

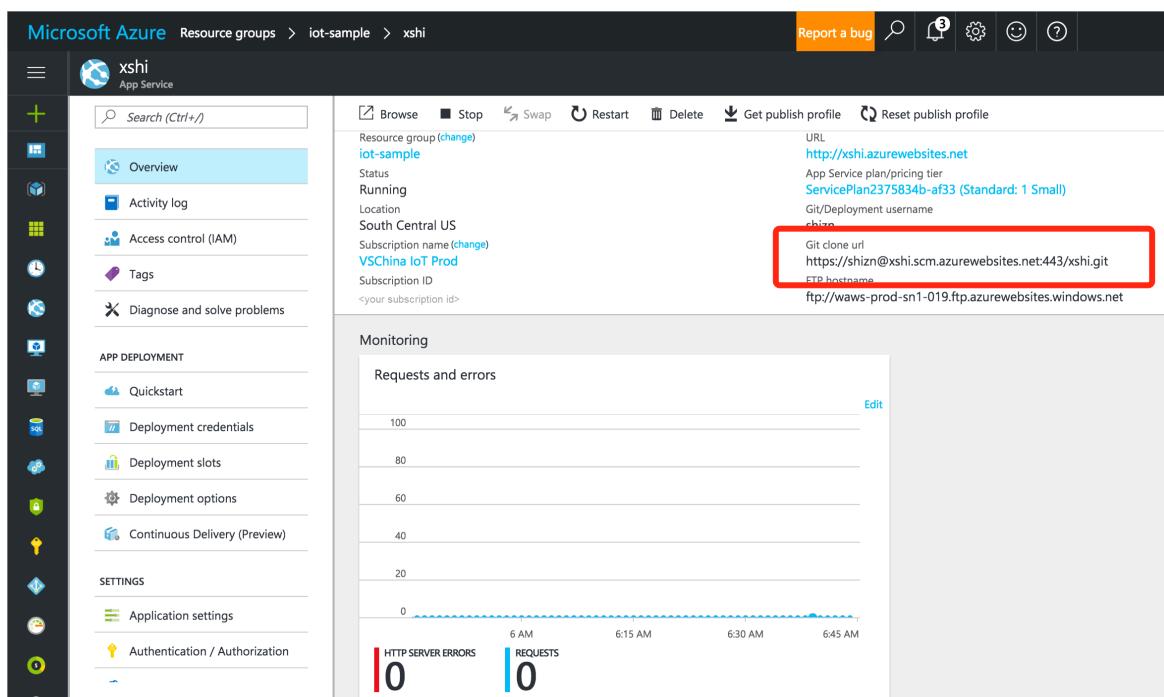
On GitHub, we've made available a web application that displays real-time sensor data from your IoT hub. All you need to do is configure the web app to work with a Git repository, download the web application from GitHub, and then upload it to Azure for the web app to host.

1. In the web app, click **Deployment Options** > **Choose Source** > **Local Git Repository**, and then click **OK**.



2. Click **Deployment Credentials**, create a user name and password to use to connect to the Git repository in Azure, and then click **Save**.

3. Click **Overview**, and note the value of **Git clone url**.



4. Open a command or terminal window on your local computer.

5. Download the web app from GitHub, and upload it to Azure for the web app to host. To do so, run the following commands:

```
git clone https://github.com/Azure-Samples/web-apps-node-iot-hub-data-visualization.git
cd web-apps-node-iot-hub-data-visualization
git remote add webapp <Git clone URL>
git push webapp master:master
```

NOTE

<Git clone URL> is the URL of the Git repository found on the **Overview** page of the web app.

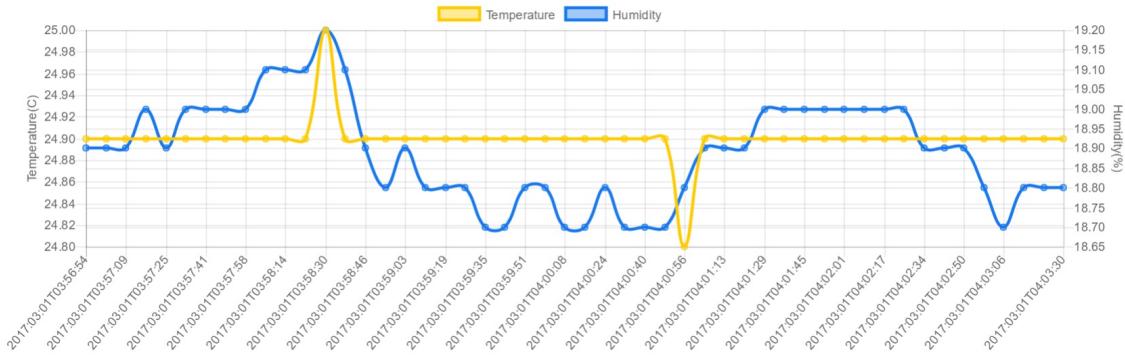
Open the web app to see real-time temperature and humidity data from your IoT hub

On the **Overview** page of your web app, click the URL to open the web app.

The screenshot shows the Microsoft Azure portal interface. On the left, there's a sidebar with various icons for different services like Storage, Database, and Container Registry. The main area is titled 'Microsoft Azure xshi' and shows the 'App Service' blade. The 'Overview' tab is selected. In the center, there's a summary card with 'Requests and errors' data: 100 requests, 0 errors, at 11:15 AM. Below this, the URL 'http://xshi.azurewebsites.net' is listed under 'Essentials' with other details like Resource group (iot-sample), Status (Running), Location (South Central US), Subscription name (VSChina IoT Prod), and Subscription ID (<your subscription id>). A red box highlights the URL. At the top right, there are buttons for 'Report a bug', search, and other account-related options.

You should see the real-time temperature and humidity data from your IoT hub.

Temperature & Humidity Real-time Data



NOTE

Ensure the sample application is running on your device. If not, you will get a blank chart, you can refer to the tutorials under [Setup your device](#).

Next steps

You've successfully used your web app to visualize real-time sensor data from your IoT hub.

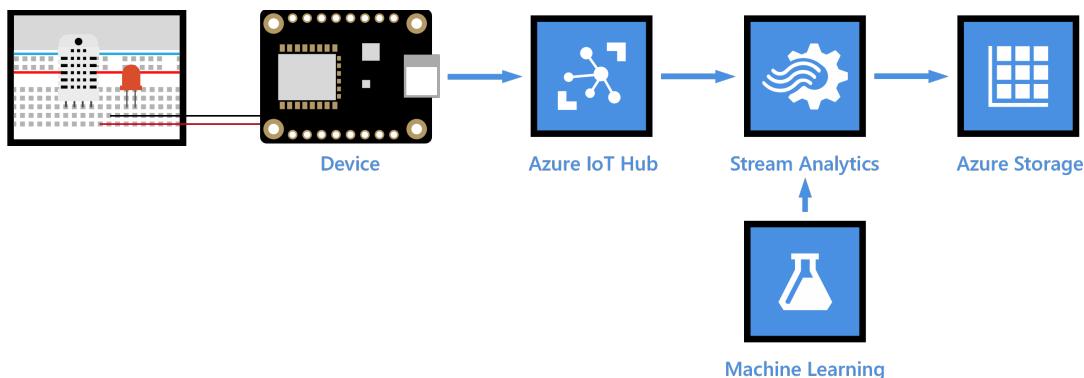
For an alternative way to visualize data from Azure IoT Hub, see [Use Power BI to visualize real-time sensor data from your IoT hub](#).

To continue to get started with Azure IoT Hub and to explore all extended IoT scenarios, see the following:

- [Manage cloud device messaging with Azure IoT Hub Toolkit extension for Visual Studio Code](#)
- [Manage devices with Azure IoT Hub Toolkit extension for Visual Studio Code](#)
- [Set up message routing](#)
- [Use Power BI to visualize real-time sensor data from your IoT hub](#)
- [Use the Web Apps feature of Azure App Service to visualize real-time sensor data from your IoT hub](#)
- [Forecast weather by using the sensor data from your IoT hub in Azure Machine Learning](#)
- [Use Logic Apps for remote monitoring and notifications](#)

Weather forecast using the sensor data from your IoT hub in Azure Machine Learning

3/6/2019 • 5 minutes to read



NOTE

Before you start this tutorial, [set up your device](#). In the article, you set up your Azure IoT device and IoT hub, and you deploy a sample application to run on your device. The application sends collected sensor data to your IoT hub.

Machine learning is a technique of data science that helps computers learn from existing data to forecast future behaviors, outcomes, and trends. Azure Machine Learning is a cloud predictive analytics service that makes it possible to quickly create and deploy predictive models as analytics solutions.

What you learn

You learn how to use Azure Machine Learning to do weather forecast (chance of rain) using the temperature and humidity data from your Azure IoT hub. The chance of rain is the output of a prepared weather prediction model. The model is built upon historic data to forecast chance of rain based on temperature and humidity.

What you do

- Deploy the weather prediction model as a web service.
- Get your IoT hub ready for data access by adding a consumer group.
- Create a Stream Analytics job and configure the job to:
 - Read temperature and humidity data from your IoT hub.
 - Call the web service to get the rain chance.
 - Save the result to an Azure blob storage.
- Use Microsoft Azure Storage Explorer to view the weather forecast.

What you need

- Tutorial [Setup your device](#) completed which covers the following requirements:
 - An active Azure subscription.
 - An Azure IoT hub under your subscription.
 - A client application that sends messages to your Azure IoT hub.
- An Azure Machine Learning Studio account. ([Try Machine Learning Studio for free](#)).

Deploy the weather prediction model as a web service

1. Go to the [weather prediction model page](#).
2. Click **Open in Studio** in Microsoft Azure Machine Learning Studio.

The screenshot shows the Cortana Intelligence Gallery interface. At the top, there's a navigation bar with 'Cortana Intelligence Gallery', a search icon, user icons, and a 'Sign in' link. Below the navigation, there are tabs for 'Browse all', 'Industries', 'Solutions', 'Experiments', and 'More'. The main area is titled 'EXPERIMENT' and contains a lock icon, the title 'Weather prediction model', the author 'Yuwei Zhou' (with a profile icon), and the date 'March 3, 2017'. To the right of the experiment title is a small graphic of a smiling face on a scale. Below the experiment title are two buttons: 'Summary' and 'Description'. On the far right, there's a green button labeled 'Open in Studio' which is highlighted with a red rectangular border. Further down, there's a '+ Add to Collection' button and a small downward arrow icon.

3. Click **Run** to validate the steps in the model. This step might take 2 minutes to complete.

The screenshot shows the Microsoft Azure Machine Learning Studio interface. The top bar includes the title 'Microsoft Azure Machine Learning Studio' and a user profile 'Xin Shi-Free-Workspace'. The main workspace displays the 'Weather prediction model' experiment. The flowchart shows a sequence of steps: 'Weather Dataset' → 'Select Columns in Dataset' → 'Edit Metadata' → 'Clean Missing Data' → 'Multiclass Logistic Regression' (which has a 'Mini Map' preview) → 'Execute R Script' → 'Split Data' → 'Select Columns in Dataset'. On the left, there's a vertical sidebar with icons for Datasets, Modules, Trained Models, and Transforms. On the right, there are sections for 'Properties' (Experiment Properties, Status Code: InDraft, Status Details: None), 'Summary' (with a text input field), 'Description' (with a text input field), and 'Quick Help'. At the bottom, there are buttons for 'NEW', 'RUN HISTORY', 'SAVE', 'SAVE AS', 'DISCARD CHANGES', 'RUN', 'SET UP WEB SERVICE' (which is highlighted with a red box), and 'PUBLISH TO GALLERY'.

4. Click **SET UP WEB SERVICE > Predictive Web Service**.

Microsoft Azure Machine Learning Studio

Xin Shi-Free-Workspace

Training experiment Predictive experiment

Weather prediction... In draft

Draft saved at 3:11:48 PM

Properties Project

Experiment Properties

- START TIME 3/6/2017...
- END TIME 3/6/2017...
- STATUS CODE InDraft
- STATUS DETAILS None

Summary

Enter a few sentences describing your experiment (up to 140 characters).

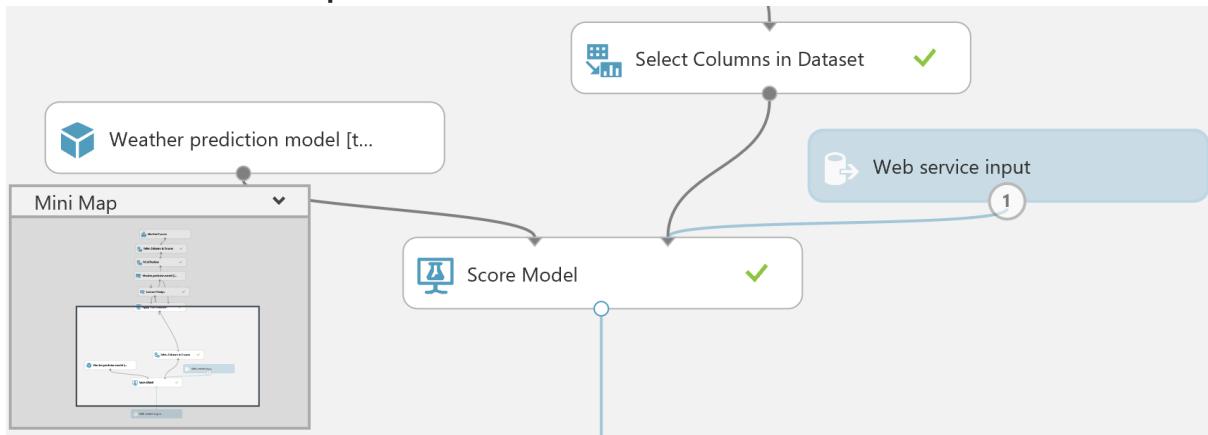
Description

Enter the detailed description for your experiment.

Quick Help

NEW RUN HISTORY SAVE SAVE AS DISCARD CHANGES **RUN** **DEPLOY WEB SERVICE** PUBLISH TO GALLERY

- In the diagram, drag the **Web service input** module somewhere near the **Score Model** module.
- Connect the **Web service input** module to the **Score Model** module.



- Click **RUN** to validate the steps in the model.
- Click **DEPLOY WEB SERVICE** to deploy the model as a web service.
- On the dashboard of the model, download the **Excel 2010 or earlier workbook** for **REQUEST/RESPONSE**.

NOTE

Ensure that you download the **Excel 2010 or earlier workbook** even if you are running a later version of Excel on your computer.

The screenshot shows the Microsoft Azure Machine Learning Studio interface. On the left, there's a sidebar with icons for Dashboard, Configuration, General, Published experiment, Description, API key, Default Endpoint, API Help Page, REQUEST/RESPONSE, BATCH EXECUTION, TEST, APPS, LAST UPDATED, and a search bar. The main area displays a 'weather prediction model [predictive exp.]' configuration. Under 'Description', it says 'No description provided for this web service.' Under 'API key', there's a long URL. Under 'Default Endpoint', the 'REQUEST/RESPONSE' tab is selected, showing 'Test' and 'Test preview'. The 'TEST' tab is highlighted. Below it, under 'BATCH EXECUTION', there's another 'Test preview'. A red box highlights the 'REQUEST/RESPONSE' section. At the bottom, there are 'NEW' and 'DELETE' buttons.

10. Open the Excel workbook, make a note of the **WEB SERVICE URL** and **ACCESS KEY**.

Add a consumer group to your IoT hub

Consumer groups are used by applications to pull data from Azure IoT Hub. In this tutorial, you create a consumer group to be used by a coming Azure service to read data from your IoT hub.

To add a consumer group to your IoT hub, follow these steps:

1. In the [Azure portal](#), open your IoT hub.
2. In the left pane, click **Built-in endpoints**, select **Events** on the top pane, and enter a name under **Consumer groups** on the right pane. Click **Save** after you change the **Default TTL** value and return it back to the original value.

The screenshot shows the Microsoft Azure portal interface. The left sidebar includes 'Create a resource', 'Home', 'Dashboard', 'All services', 'FAVORITES' (with 'Resource groups' selected), 'App Services', 'Function Apps', 'SQL databases', 'Azure Cosmos DB', 'Virtual machines', 'Load balancers', 'Storage accounts', 'Virtual networks', 'Azure Active Directory', 'Monitor', 'Advisor', 'Security Center', 'Cost Management + Billing', and 'Help + support'. The main content area shows 'testhub2 - Built-in endpoints'. The 'Events' section is selected. It displays a message about default endpoints and a table for 'Event Hub-compatible name' (set to 'testhub2') and 'Event Hub-compatible endpoint' (set to 'Endpoint=sb://iothub-ns-**evan-test-1173979-0f37235e38.servicebus.windows.net/SharedAccessKeyName=iothubowner/SharedAccessKey=...'). It also shows a 'Consumer Groups' section with 'CONSUMER GROUPS' (containing 'Default' and 'consumer2') and a 'Cloud to device messaging' section with a 'Default TTL' of 1 hour. There are 'Save' and 'Undo' buttons at the top.**

Create, configure, and run a Stream Analytics job

Create a Stream Analytics job

1. In the [Azure portal](#), click **Create a resource > Internet of Things > Stream Analytics job**.
2. Enter the following information for the job.

Job name: The name of the job. The name must be globally unique.

Resource group: Use the same resource group that your IoT hub uses.

Location: Use the same location as your resource group.

Pin to dashboard: Check this option for easy access to your IoT hub from the dashboard.

The screenshot shows the Azure portal interface for creating a new Stream Analytics job. The left sidebar lists various service categories like Compute, Networking, Storage, etc. The central area shows 'FEATURED APPS' with icons for IoT Hub, Event Hubs, Stream Analytics job (selected), Machine Learning Workspace, and Machine Learning Web Service. The right panel is titled 'New Stream Analytics Job...' and contains fields for 'Job name' (mystreamanalysis), 'Subscription' (Visual Studio Enterprise with MSDN), 'Resource group' (Default-ApplicationInsights-CentralUS), and 'Location'. A checkbox for 'Pin to dashboard' is present, and at the bottom are 'Create' and 'Automation options' buttons.

3. Click **Create**.

Add an input to the Stream Analytics job

1. Open the Stream Analytics job.
2. Under **Job Topology**, click **Inputs**.
3. In the **Inputs** pane, click **Add**, and then enter the following information:

Input alias: The unique alias for the input.

Source: Select **IoT hub**.

Consumer group: Select the consumer group you created.

The screenshot shows the Azure portal interface for adding a new input to a Stream Analytics job. The left sidebar shows the job name 'rpi-c'. The central area shows the 'Inputs' pane with a table listing one entry: 'iothub' (Stream type, IoT hub source). The right panel is titled 'New input' and contains fields for 'Source' (IoT hub), 'Subscription' (Use IoT hub from current subscription), 'IoT hub' (mygateway), 'Endpoint' (Messaging), 'Shared access policy name' (iothubowner), 'Shared access policy key' (redacted), and 'Consumer group' (stream). At the bottom are 'Create' and 'Automation options' buttons.

4. Click **Create**.

Add an output to the Stream Analytics job

1. Under **Job Topology**, click **Outputs**.

2. In the **Outputs** pane, click **Add**, and then enter the following information:

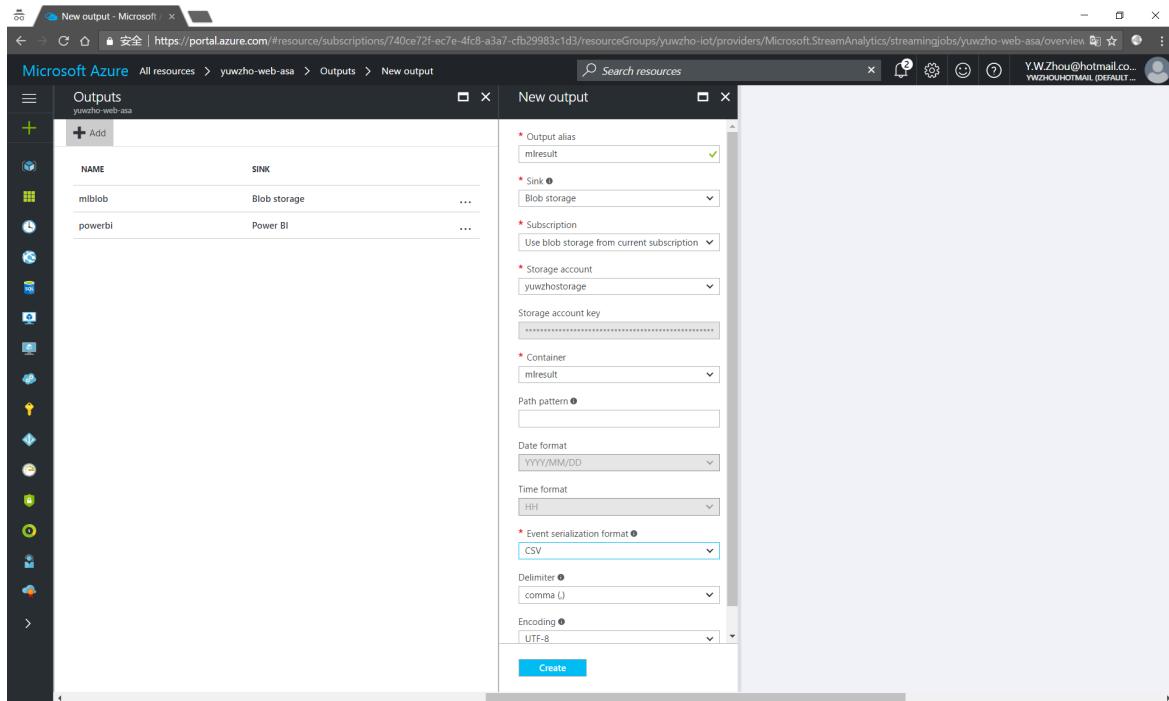
Output alias: The unique alias for the output.

Sink: Select **Blob Storage**.

Storage account: The storage account for your blob storage. You can create a storage account or use an existing one.

Container: The container where the blob is saved. You can create a container or use an existing one.

Event serialization format: Select **CSV**.



3. Click **Create**.

Add a function to the Stream Analytics job to call the web service you deployed

1. Under **Job Topology**, click **Functions** > **Add**.

2. Enter the following information:

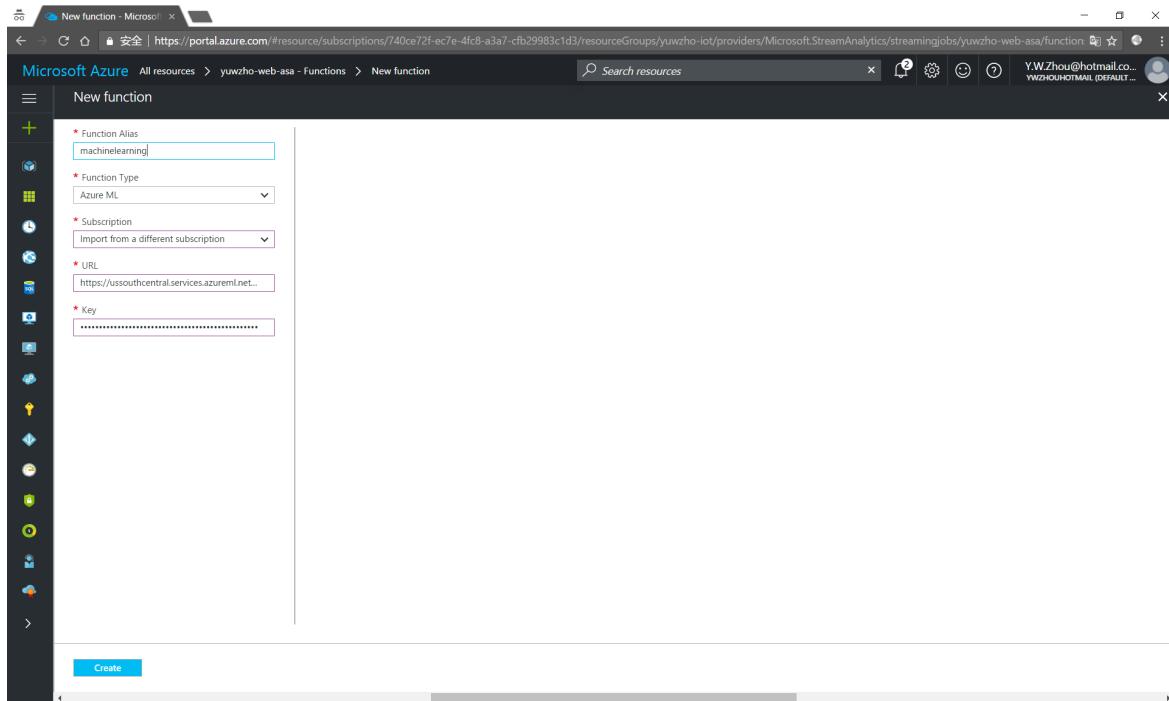
Function Alias: Enter `machinelearning`.

Function Type: Select **Azure ML**.

Import option: Select **Import from a different subscription**.

URL: Enter the WEB SERVICE URL that you noted down from the Excel workbook.

Key: Enter the ACCESS KEY that you noted down from the Excel workbook.



3. Click **Create**.

Configure the query of the Stream Analytics job

1. Under **Job Topology**, click **Query**.
2. Replace the existing code with the following code:

```
WITH machinelearning AS (
    SELECT EventEnqueuedUtcTime, temperature, humidity, machinelearning(temperature, humidity) as
    result from [YourInputAlias]
)
Select System.Timestamp time, CAST (result.[temperature] AS FLOAT) AS temperature, CAST (result.
[humidity] AS FLOAT) AS humidity, CAST (result.[Scored Probabilities] AS FLOAT ) AS 'probabalities of
rain'
Into [YourOutputAlias]
From machinelearning
```

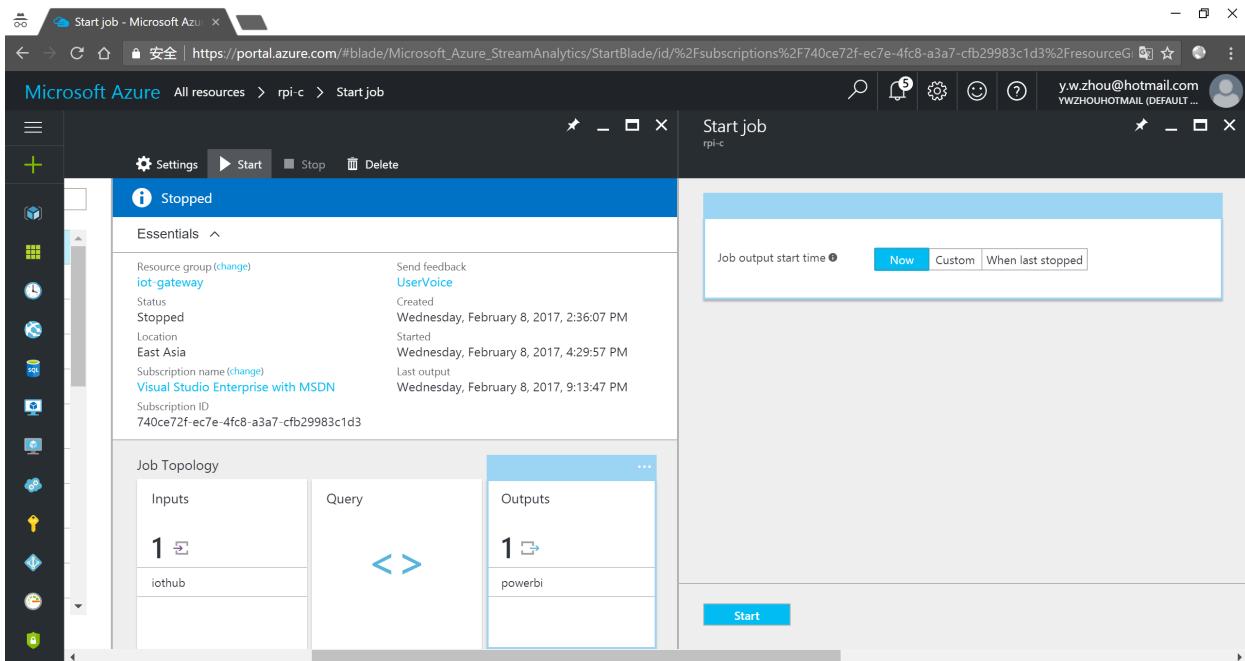
Replace `[YourInputAlias]` with the input alias of the job.

Replace `[YourOutputAlias]` with the output alias of the job.

3. Click **Save**.

Run the Stream Analytics job

In the Stream Analytics job, click **Start > Now > Start**. Once the job successfully starts, the job status changes from **Stopped** to **Running**.



Use Microsoft Azure Storage Explorer to view the weather forecast

Run the client application to start collecting and sending temperature and humidity data to your IoT hub. For each message that your IoT hub receives, the Stream Analytics job calls the weather forecast web service to produce the chance of rain. The result is then saved to your Azure blob storage. Azure Storage Explorer is a tool that you can use to view the result.

1. [Download and install Microsoft Azure Storage Explorer](#).
2. Open Azure Storage Explorer.
3. Sign in to your Azure account.
4. Select your subscription.
5. Click your subscription > **Storage Accounts** > your storage account > **Blob Containers** > your container.
6. Open a .csv file to see the result. The last column records the chance of rain.

```
time,temperature,humidity,probabalities of rain
2017-03-07T02:16:26.353000Z,24.6,20.2,0.506996631622314
2017-03-07T02:16:56.874000Z,25.3,13.3,0.506996631622314
2017-03-07T02:17:04.842000Z,25.2,13.2,0.506996631622314
2017-03-07T02:17:13.154000Z,25.2,13.4,0.506996631622314
2017-03-07T02:17:22.630000Z,25.1,13.3,0.506996631622314
2017-03-07T02:17:30.828000Z,25.1,13.6,0.506996631622314
2017-03-07T02:17:38.889000Z,25,13.9,0.506996631622314
2017-03-07T02:17:54.147000Z,25,13.7,0.506996631622314
2017-03-07T02:18:03.224000Z,24.9,13.3,0.506996631622314
2017-03-07T02:18:10.757000Z,24.9,15.4,0.506996631622314
2017-03-07T02:18:20.388000Z,24.9,14.2,0.506996631622314
2017-03-07T02:18:29.260000Z,24.9,14.6,0.506996631622314
2017-03-07T02:18:38.971000Z,24.8,14.9,0.506996631622314
2017-03-07T02:18:48.002000Z,24.8,14.2,0.506996631622314
2017-03-07T02:18:56.252000Z,24.8,14.7,0.506996631622314
2017-03-07T02:19:04.180000Z,24.8,15.3,0.506996631622314
2017-03-07T02:19:12.965000Z,24.8,14.3,0.506996631622314
2017-03-07T02:19:21.824000Z,24.8,13.8,0.506996631622314
2017-03-07T02:19:31.229000Z,24.8,13.6,0.506996631622314
2017-03-07T02:19:39.330000Z,24.8,14.1,0.506996631622314
2017-03-07T02:19:47.286000Z,24.8,13.6,0.506996631622314
2017-03-07T02:20:13.759000Z,24.8,12.7,0.506996631622314
2017-03-07T02:20:21.356000Z,24.7,13.4,0.517819762229919
2017-03-07T02:20:29.168000Z,24.7,13.4,0.517819762229919
```

Summary

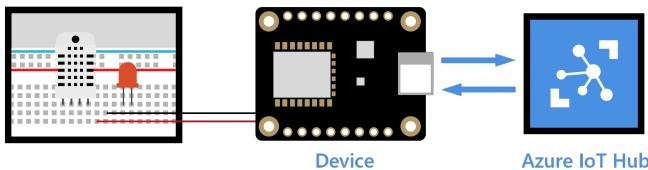
You've successfully used Azure Machine Learning to produce the chance of rain based on the temperature and humidity data that your IoT hub receives.

To continue to get started with Azure IoT Hub and to explore all extended IoT scenarios, see the following:

- [Manage cloud device messaging with Azure IoT Hub Toolkit extension for Visual Studio Code](#)
- [Manage devices with Azure IoT Hub Toolkit extension for Visual Studio Code](#)
- [Set up message routing](#)
- [Use Power BI to visualize real-time sensor data from your IoT hub](#)
- [Use the Web Apps feature of Azure App Service to visualize real-time sensor data from your IoT hub](#)
- [Forecast weather by using the sensor data from your IoT hub in Azure Machine Learning](#)
- [Use Logic Apps for remote monitoring and notifications](#)

Use Azure IoT Tools for Visual Studio Code for Azure IoT Hub device management

2/22/2019 • 3 minutes to read



[Azure IoT Tools](#) is a useful Visual Studio Code extension that makes IoT Hub management and IoT application development easier. It comes with management options that you can use to perform various tasks.

NOTE

The features described in this article are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

MANAGEMENT OPTION	TASK
Direct methods	Make a device act such as starting or stopping sending messages or rebooting the device.
Read device twin	Get the reported state of a device. For example, the device reports the LED is blinking now.
Update device twin	Put a device into certain states, such as setting an LED to green or setting the telemetry send interval to 30 minutes.
Cloud-to-device messages	Send notifications to a device. For example, "It is very likely to rain today. Don't forget to bring an umbrella."

For more detailed explanation on the differences and guidance on using these options, see [Device-to-cloud communication guidance](#) and [Cloud-to-device communication guidance](#).

Device twins are JSON documents that store device state information (metadata, configurations, and conditions). IoT Hub persists a device twin for each device that connects to it. For more information about device twins, see [Get started with device twins](#).

NOTE

This article has been updated to use the new Azure PowerShell Az module. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For installation instructions, see [Install Azure PowerShell](#).

What you learn

You learn using Azure IoT Tools for Visual Studio Code with various management options on your development machine.

What you do

Run Azure IoT Tools for Visual Studio Code with various management options.

What you need

- An active Azure subscription.
- An Azure IoT hub under your subscription.
- [Visual Studio Code](#)
- [Azure IoT Tools for VS Code](#)

Sign in to access your IoT hub

1. In **Explorer** view of VS Code, expand **Azure IoT Hub Devices** section in the bottom left corner.
2. Click **Select IoT Hub** in context menu.
3. A pop-up will show in the bottom right corner to let you sign in to Azure for the first time.
4. After you sign in, your Azure Subscription list will be shown, then select Azure Subscription and IoT Hub.
5. The device list will be shown in **Azure IoT Hub Devices** tab in a few seconds.

NOTE

You can also complete the set up by choosing **Set IoT Hub Connection String**. Enter the connection string for the IoT hub that your IoT device connects to in the pop-up window.

Direct methods

1. Right-click your device and select **Invoke Direct Method**.
2. Enter the method name and payload in input box.
3. Results will be shown in **OUTPUT > Azure IoT Hub Toolkit** view.

Read device twin

1. Right-click your device and select **Edit Device Twin**.
2. An **azure-iot-device-twin.json** file will be opened with the content of device twin.

Update device twin

1. Make some edits of **tags** or **properties.desired** field.
2. Right-click on the **azure-iot-device-twin.json** file.
3. Select **Update Device Twin** to update the device twin.

Send cloud-to-device messages

To send a message from your IoT hub to your device, follow these steps:

1. Right-click your device and select **Send C2D Message to Device**.
2. Enter the message in input box.
3. Results will be shown in **OUTPUT > Azure IoT Hub Toolkit** view.

Next steps

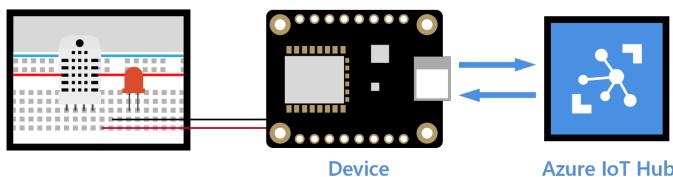
You've learned how to use Azure IoT Tools extension for Visual Studio Code with various management options.

To continue to get started with Azure IoT Hub and to explore all extended IoT scenarios, see the following:

- [Manage cloud device messaging with Azure IoT Hub Toolkit extension for Visual Studio Code](#)
- [Manage devices with Azure IoT Hub Toolkit extension for Visual Studio Code](#)
- [Set up message routing](#)
- [Use Power BI to visualize real-time sensor data from your IoT hub](#)
- [Use the Web Apps feature of Azure App Service to visualize real-time sensor data from your IoT hub](#)
- [Forecast weather by using the sensor data from your IoT hub in Azure Machine Learning](#)
- [Use Logic Apps for remote monitoring and notifications](#)

Use Cloud Explorer for Visual Studio for Azure IoT Hub device management

1/8/2019 • 3 minutes to read



[Cloud Explorer](#) is a useful Visual Studio extension that enables you to view your Azure resources, inspect their properties and perform key developer actions from within Visual Studio. It comes with management options that you can use to perform various tasks.

NOTE

The features described in this article are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

MANAGEMENT OPTION	TASK
Direct methods	Make a device act such as starting or stopping sending messages or rebooting the device.
Read device twin	Get the reported state of a device. For example, the device reports the LED is blinking now.
Update device twin	Put a device into certain states, such as setting an LED to green or setting the telemetry send interval to 30 minutes.
Cloud-to-device messages	Send notifications to a device. For example, "It is very likely to rain today. Don't forget to bring an umbrella."

For more detailed explanation on the differences and guidance on using these options, see [Device-to-cloud communication guidance](#) and [Cloud-to-device communication guidance](#).

Device twins are JSON documents that store device state information (metadata, configurations, and conditions). IoT Hub persists a device twin for each device that connects to it. For more information about device twins, see [Get started with device twins](#).

What you learn

You learn using Cloud Explorer for Visual Studio with various management options on your development machine.

What you do

Run Cloud Explorer for Visual Studio with various management options.

What you need

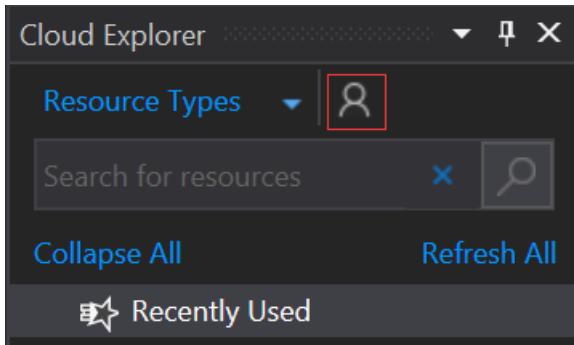
- An active Azure subscription.
- An Azure IoT Hub under your subscription.
- Microsoft Visual Studio 2017 Update 8 or later
- Cloud Explorer component from Visual Studio Installer (selected by default with Azure Workload)

Update Cloud Explorer to latest version

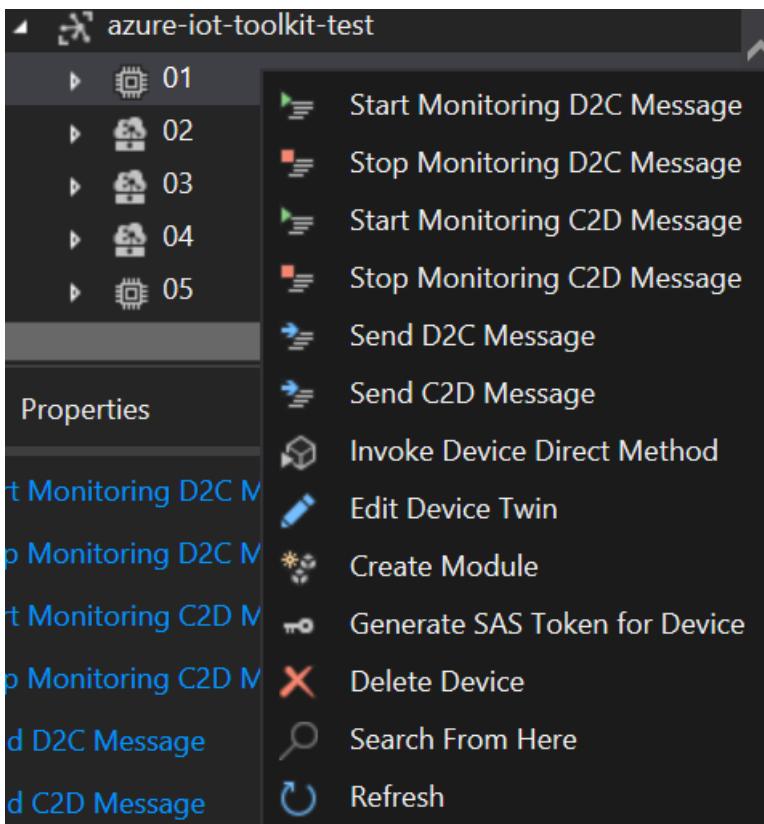
The Cloud Explorer component from Visual Studio Installer only supports monitoring device-to-cloud and cloud-to-device messages. You need to download and install the latest [Cloud Explorer](#) to access the management options.

Sign in to access your IoT Hub

1. In Visual Studio **Cloud Explorer** window, click the Account Management icon. You can open the Cloud Explorer window from **View > Cloud Explorer** menu.



2. Click **Manage Accounts** in Cloud Explorer.
3. Click **Add an account...** in the new window to sign in to Azure for the first time.
4. After you sign in, your Azure subscription list will be shown. Select the Azure subscriptions you want to view and click **Apply**.
5. Expand **Your subscription > IoT Hubs > Your IoT Hub**, the device list will be shown under your IoT Hub node. Right-click one device to access the management options.



Direct methods

1. Right-click your device and select **Invoke Device Direct Method**.
2. Enter the method name and payload in input box.
3. Results will be shown in the **IoT Hub** output pane.

Read device twin

1. Right-click your device and select **Edit Device Twin**.
2. An **azure-iot-device-twin.json** file will be opened with the content of device twin.

Update device twin

1. Make some edits of **tags** or **properties.desired** field to the **azure-iot-device-twin.json** file.
2. Press **Ctrl+S** to update the device twin.
3. Results will be shown in the **IoT Hub** output pane.

Send cloud-to-device messages

To send a message from your IoT Hub to your device, follow these steps:

1. Right-click your device and select **Send C2D Message**.
2. Enter the message in input box.
3. Results will be shown in the **IoT Hub** output pane.

Next steps

You've learned how to use Cloud Explorer for Visual Studio with various management options.

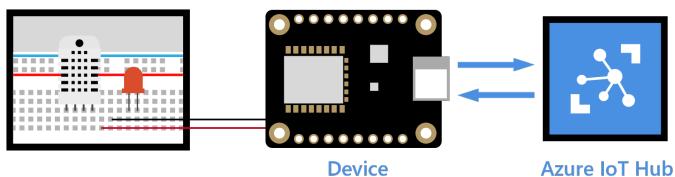
To continue to get started with Azure IoT Hub and to explore all extended IoT scenarios, see the following:

- [Manage cloud device messaging with Azure IoT Hub Toolkit extension for Visual Studio Code](#)

- [Manage devices with Azure IoT Hub Toolkit extension for Visual Studio Code](#)
- [Set up message routing](#)
- [Use Power BI to visualize real-time sensor data from your IoT hub](#)
- [Use the Web Apps feature of Azure App Service to visualize real-time sensor data from your IoT hub](#)
- [Forecast weather by using the sensor data from your IoT hub in Azure Machine Learning](#)
- [Use Logic Apps for remote monitoring and notifications](#)

Use the IoT extension for Azure CLI for Azure IoT Hub device management

10/2/2018 • 4 minutes to read



NOTE

Before you start this tutorial, [set up your device](#). In the article, you set up your Azure IoT device and IoT hub, and you deploy a sample application to run on your device. The application sends collected sensor data to your IoT hub.

The [IoT extension for Azure CLI](#) is a new open source IoT extension that adds to the capabilities of the [Azure CLI](#). The Azure CLI includes commands for interacting with Azure resource manager and management endpoints. For example, you can use Azure CLI to create an Azure VM or an IoT hub. A CLI extension enables an Azure service to augment the Azure CLI giving you access to additional service-specific capabilities. The IoT extension gives IoT developers command line access to all IoT Hub, IoT Edge, and IoT Hub Device Provisioning Service capabilities.

NOTE

The features described in this article are only available in the standard tier of IoT hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

MANAGEMENT OPTION	TASK
Direct methods	Make a device act such as starting or stopping sending messages or rebooting the device.
Twin desired properties	Put a device into certain states, such as setting an LED to green or setting the telemetry send interval to 30 minutes.
Twin reported properties	Get the reported state of a device. For example, the device reports the LED is blinking now.
Twin tags	Store device-specific metadata in the cloud. For example, the deployment location of a vending machine.
Device twin queries	Query all device twins to retrieve those with arbitrary conditions, such as identifying the devices that are available for use.

For more detailed explanation on the differences and guidance on using these options, see [Device-to-cloud](#)

[communication guidance](#) and [Cloud-to-device communication guidance](#).

Device twins are JSON documents that store device state information (metadata, configurations, and conditions). IoT Hub persists a device twin for each device that connects to it. For more information about device twins, see [Get started with device twins](#).

What you learn

You learn using the IoT extension for Azure CLI with various management options on your development machine.

What you do

Run Azure CLI and the IoT extension for Azure CLI with various management options.

What you need

- Complete the tutorial [Setup your device](#) which covers the following requirements:
 - An active Azure subscription.
 - An Azure IoT hub under your subscription.
 - A client application that sends messages to your Azure IoT hub.
- Make sure your device is running with the client application during this tutorial.
- [Python 2.7x or Python 3.x](#)
- The Azure CLI. If you need to install it, see [Install the Azure CLI](#). At a minimum, your Azure CLI version must be 2.0.24 or above. Use `az --version` to validate.
- Install the IoT extension. The simplest way is to run `az extension add --name azure-cli-iot-ext`. [The IoT extension readme](#) describes several ways to install the extension.

Log in to your Azure account

Log in to your Azure account by running the following command:

```
az login
```

Direct methods

```
az iot hub invoke-device-method --device-id <your device id> \
--hub-name <your hub name> \
--method-name <the method name> \
--method-payload <the method payload>
```

Device twin desired properties

Set a desired property interval = 3000 by running the following command:

```
az iot hub device-twin update -n <your hub name> \
-d <your device id> --set properties.desired.interval = 3000
```

This property can be read from your device.

Device twin reported properties

Get the reported properties of the device by running the following command:

```
az iot hub device-twin show -n <your hub name> -d <your device id>
```

One of the twin reported properties is \$metadata.\$lastUpdated which shows the last time the device app updated its reported property set.

Device twin tags

Display the tags and properties of the device by running the following command:

```
az iot hub device-twin show --hub-name <your hub name> --device-id <your device id>
```

Add a field role = temperature&humidity to the device by running the following command:

```
az iot hub device-twin update \  
  --hub-name <your hub name> \  
  --device-id <your device id> \  
  --set tags = '{"role":"temperature&humidity"}'
```

Device twin queries

Query devices with a tag of role = 'temperature&humidity' by running the following command:

```
az iot hub query --hub-name <your hub name> \  
  --query-command "SELECT * FROM devices WHERE tags.role = 'temperature&humidity'"
```

Query all devices except those with a tag of role = 'temperature&humidity' by running the following command:

```
az iot hub query --hub-name <your hub name> \  
  --query-command "SELECT * FROM devices WHERE tags.role != 'temperature&humidity'"
```

Next steps

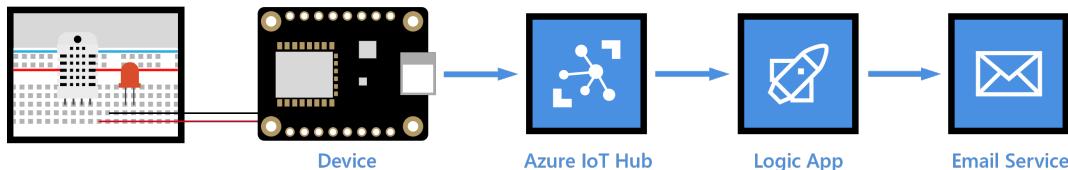
You've learned how to monitor device-to-cloud messages and send cloud-to-device messages between your IoT device and Azure IoT Hub.

To continue to get started with Azure IoT Hub and to explore all extended IoT scenarios, see the following:

- [Manage cloud device messaging with Azure IoT Hub Toolkit extension for Visual Studio Code](#)
- [Manage devices with Azure IoT Hub Toolkit extension for Visual Studio Code](#)
- [Set up message routing](#)
- [Use Power BI to visualize real-time sensor data from your IoT hub](#)
- [Use the Web Apps feature of Azure App Service to visualize real-time sensor data from your IoT hub](#)
- [Forecast weather by using the sensor data from your IoT hub in Azure Machine Learning](#)
- [Use Logic Apps for remote monitoring and notifications](#)

IoT remote monitoring and notifications with Azure Logic Apps connecting your IoT hub and mailbox

10/18/2018 • 4 minutes to read



NOTE

Before you start this tutorial, [set up your device](#). In the article, you set up your Azure IoT device and IoT hub, and you deploy a sample application to run on your device. The application sends collected sensor data to your IoT hub.

Azure Logic Apps provides a way to automate processes as a series of steps. A logic app can connect across various services and protocols. It begins with a trigger such as 'When an account is added', and followed by a combination of actions, one like 'sending a push notification'. This feature makes Logic Apps a perfect IoT solution for IoT monitoring, such as staying alert for anomalies, among other usage scenarios.

What you learn

You learn how to create a logic app that connects your IoT hub and your mailbox for temperature monitoring and notifications. When the temperature is above 30 C, the client application marks `temperatureAlert = "true"` in the message it sends to your IoT hub. The message triggers the logic app to send you an email notification.

What you do

- Create a service bus namespace and add a queue to it.
- Add an endpoint and a routing rule to your IoT hub.
- Create, configure, and test a logic app.

What you need

- Tutorial [Setup your device](#) completed which covers the following requirements:
 - An active Azure subscription.
 - An Azure IoT hub under your subscription.
 - A client application that sends messages to your Azure IoT hub.

Create service bus namespace and add a queue to it

Create a service bus namespace

1. On the [Azure portal](#), click **Create a resource > Enterprise Integration > Service Bus**.
2. Provide the following information:

Name: The name of the service bus.

Pricing tier: Click **Basic** > **Select**. The Basic tier is sufficient for this tutorial.

Resource group: Use the same resource group that your IoT hub uses.

Location: Use the same location that your IoT hub uses.

3. Click **Create**.

The screenshot shows the 'Create queue' blade in the Microsoft Azure portal. On the left, there's a sidebar with icons for Queue, Topic, Delete, and Convert. The main area has a heading 'Create queue' with a 'PREVIEW' link. It shows the following configuration:

- Name:** queue
- Pricing tier:** Basic
- Connection Strings:** Connection Strings
- Max size:** 1 GB
- Message time to live (default):** 14 days
- Lock duration:** 30 seconds
- Checkboxes (checked):** Move expired messages to the dead-letter subqueue, Enable partitioning.
- Checkboxes (unchecked):** Enable duplicate detection, Enable sessions.

A large blue 'Create' button is located at the bottom right of the form.

Add a service bus queue

1. Open the service bus namespace, and then click **+ Queue**.
2. Enter a name for the queue and then click **Create**.
3. Open the service bus queue, and then click **Shared access policies** > **+ Add**.
4. Enter a name for the policy, check **Manage**, and then click **Create**.

The screenshot shows the 'Add new shared access policy' blade in the Microsoft Azure portal. On the left, there's a sidebar with icons for Queue, Topic, Delete, and Convert. The main area has a heading 'Add new shared access policy...' with a 'PREVIEW' link. It shows the following configuration:

- Policy name:** (empty)
- CLAIMS:** Send (checked)
- Manage:** Manage (checked)
- Send:** Send (checked)
- Listen:** Listen (checked)

A large blue 'Create' button is located at the bottom right of the form.

Add an endpoint and a routing rule to your IoT hub

Add an endpoint

1. Open your IoT hub, click Endpoints > + Add.
2. Enter the following information:

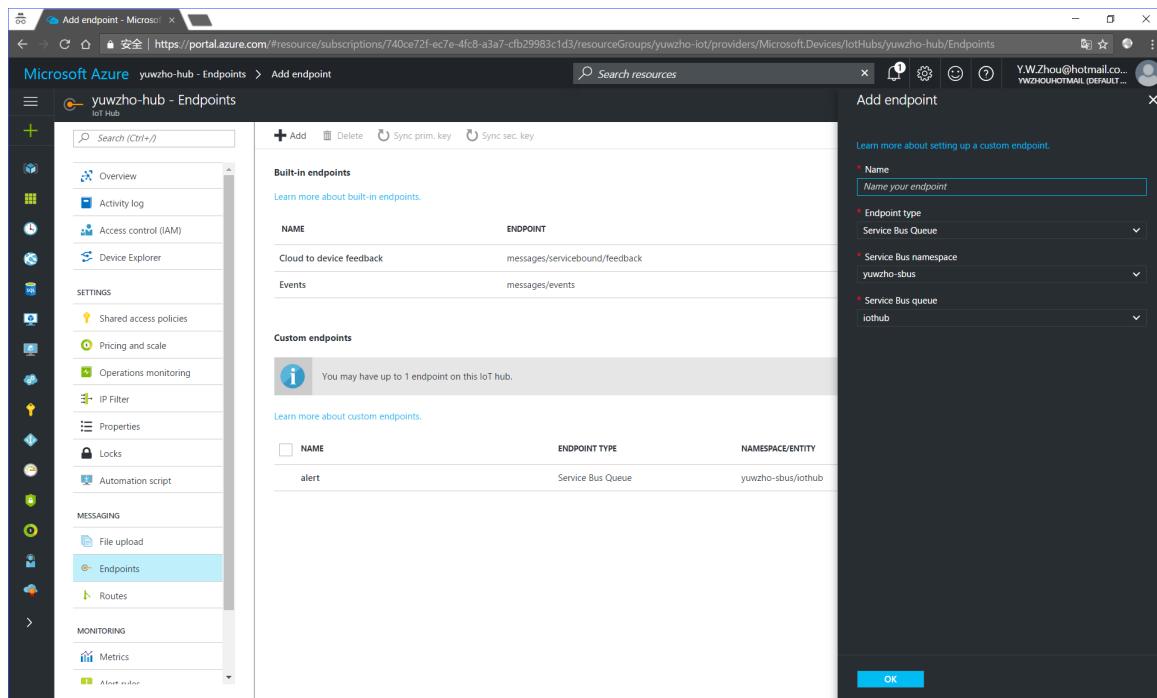
Name: The name of the endpoint.

Endpoint type: Select **Service Bus Queue**.

Service Bus namespace: Select the namespace you created.

Service Bus queue: Select the queue you created.

3. Click **OK**.



Add a routing rule

1. In your IoT hub, click **Routes** > **+ Add**.
2. Enter the following information:

Name: The name of the routing rule.

Data source: Select **DeviceMessages**.

Endpoint: Select the endpoint you created.

Query string: Enter `temperatureAlert = "true"`.

3. Click **Save**.

The screenshot shows the Azure portal's 'Edit a route' interface for an IoT Hub. The left sidebar lists various IoT Hub settings like Overview, Activity log, Access control (IAM), Device Explorer, Shared access policies, Pricing and scale, Operations monitoring, IP Filter, Properties, Locks, Automation script, File upload, Endpoints, and Routes. The 'Routes' option is selected. The main area shows a table of routes. One route is listed: 'myalert' from 'DeviceMessages' with the condition 'temperatureAlert = "true"'. The right side of the screen has a configuration pane for the route, including fields for Name (set to 'myalert'), Data source (set to 'DeviceMessages'), Endpoint (set to 'alert'), and a 'Query string' field containing the query 'temperatureAlert = "true"'. A 'Save' button is located at the bottom right of the configuration pane.

Create and configure a logic app

Create a logic app

1. In the [Azure portal](#), click **Create a resource > Enterprise Integration > Logic App**.
2. Enter the following information:

Name: The name of the logic app.

Resource group: Use the same resource group that your IoT hub uses.

Location: Use the same location that your IoT hub uses.

3. Click **Create**.

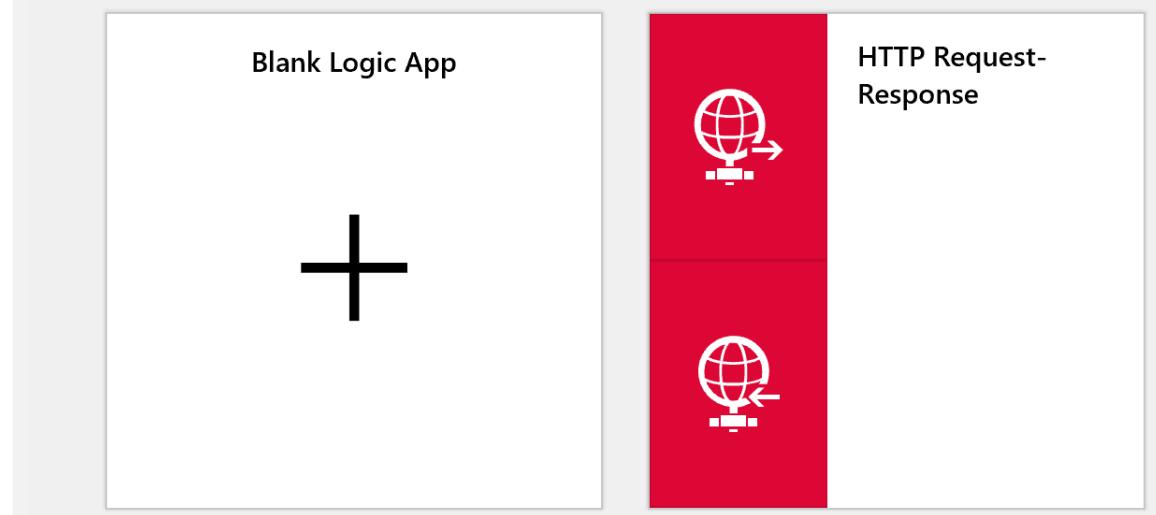
Configure the logic app

1. Open the logic app that opens into the Logic Apps Designer.
2. In the Logic Apps Designer, click **Blank Logic App**.

Logic Apps Designer

Templates

Choose a template below to create your Logic App.



3. Click **Service Bus**.

The screenshot shows the 'Services' section of the Logic Apps Designer. At the top is a search bar with the placeholder 'Search all services and triggers'. Below the search bar are two rows of service icons. The top row includes 'Request / Response' (red), 'Schedule' (orange), 'Service Bus' (green), 'Twitter' (blue), and 'OneDrive for Business' (dark blue). The bottom row includes 'Dynamics 365' (dark blue), 'SharePoint' (blue), 'FTP' (orange), 'SFTP' (yellow), and 'Office 365 Outlook' (blue). In the top right corner of the service grid, there is a 'SEE MORE' link.

4. Click **Service Bus – When one or more messages arrive in a queue (auto-complete)**.

5. Create a service bus connection.

- Enter a connection name.
- Click the service bus namespace > the service bus policy > **Create**.

 When one or more messages arrive in a queue (auto-com... ...

* CONNECTION NAME
ChenServiceBusConnection

* SERVICE BUS POLICY

Name	Rights
RootManageSharedAccessKey	Listen, Manage, Send





Manually enter connection information

- c. Click **Continue** after the service bus connection is created.
- d. Select the queue that you created and enter for **Maximum message count**

 When one or more messages arrive in a queue (auto-com... ...

* Queue name

Maximum message count

[Show advanced options](#) 

How often do you want to check for items?

* Frequency	* Interval
<input type="text" value="Minute"/>	<input type="text" value="3"/>

Connected to ChenServiceBusConnection. [Change connection](#).

- e. Click "Save" button to save the changes.
- 6. Create an SMTP service connection.
 - a. Click **New step > Add an action**.
 - b. Type SMTP service in the search result, and then click **SMTP - Send Email**.

 When one or more messages arrive in a queue (auto-com... ...



 Choose an action



SERVICES SEE MORE


SMTP

TRIGGERS (0) ACTIONS (2) SEE MORE

 SMTP - Send Email (i)

 SMTP - Send Email (V2) Preview (i)

TELL US WHAT YOU NEED

 Help us decide which services and triggers to add next
with [UserVoice](#)

Cancel

- c. Enter the SMTP information of your mailbox, and then click **Create**.

The screenshot shows a logic app with two steps. The first step is 'When one or more messages arrive in a queue (auto-generated)', indicated by a green icon. An arrow points down to the second step, 'SMTP - Send Email', indicated by an orange icon. The 'SMTP - Send Email' step has several configuration fields:

- * CONNECTION NAME: Enter name for connection
- * SMTP SERVER ADDRESS: SMTP Server Address
- * USER NAME: User Name
- * PASSWORD: Password
- SMTP SERVER PORT: SMTP Port Number (example: 587)
- ENABLE SSL?: Enable SSL? (True/False) (i)

A large 'Create' button is located at the bottom of the configuration area.

Get the SMTP information for [Hotmail/Outlook.com](#), [Gmail](#), and [Yahoo Mail](#).

- d. Enter your email address for **From** and **To**, and **High temperature detected** for **Subject** and **Body**.
- e. Click **Save**.

The logic app is in working order when you save it.

Test the logic app

1. Start the client application that you deploy to your device in [Connect ESP8266 to Azure IoT Hub](#).
2. Increase the environment temperature around the SensorTag to be above 30 C. For example, light a candle around your SensorTag.
3. You should receive an email notification sent by the logic app.

NOTE

Your email service provider may need to verify the sender identity to make sure it is you who sends the email.

Next steps

You have successfully created a logic app that connects your IoT hub and your mailbox for temperature monitoring and notifications.

To continue to get started with Azure IoT Hub and to explore all extended IoT scenarios, see the following:

- Manage cloud device messaging with Azure IoT Hub Toolkit extension for Visual Studio Code
- Manage devices with Azure IoT Hub Toolkit extension for Visual Studio Code
- Set up message routing
- Use Power BI to visualize real-time sensor data from your IoT hub
- Use the Web Apps feature of Azure App Service to visualize real-time sensor data from your IoT hub
- Forecast weather by using the sensor data from your IoT hub in Azure Machine Learning
- Use Logic Apps for remote monitoring and notifications

Detect and troubleshoot disconnects with Azure IoT Hub

3/2/2019 • 4 minutes to read

Connectivity issues for IoT devices can be difficult to troubleshoot because there are many possible points of failure. Device-side application logic, physical networks, protocols, hardware, and Azure IoT Hub can all cause problems. This article provides recommendations on how to detect and troubleshoot device connectivity issues from the cloud side (as opposed to device side).

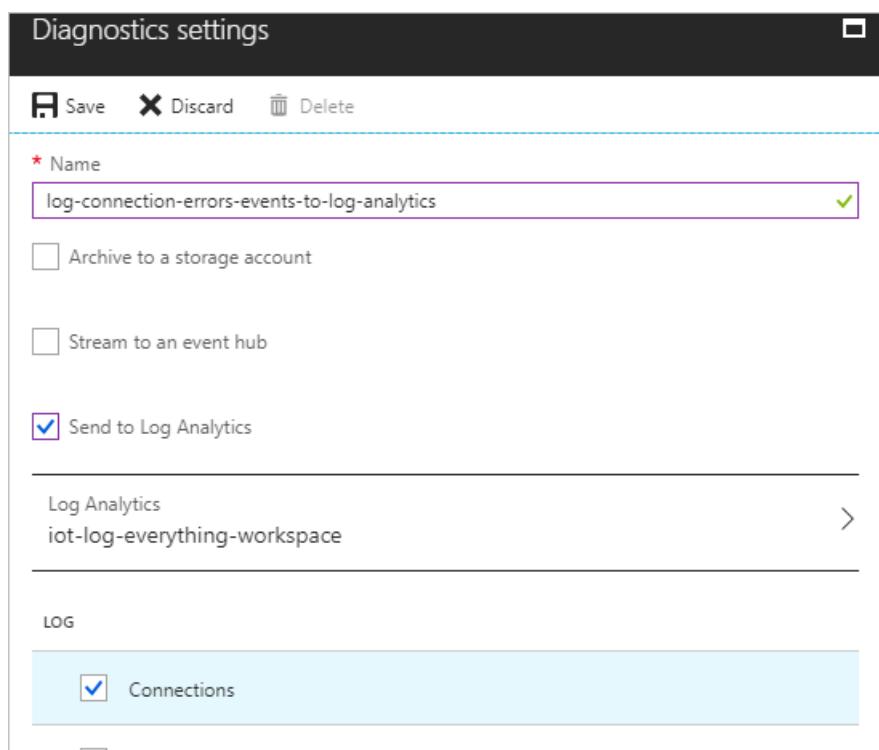
Get alerts and error logs

Use Azure Monitor to get alerts and write logs when device connections drop.

Turn on diagnostic logs

To log device connection events and errors, turn on diagnostics for IoT Hub.

1. Sign in to the [Azure portal](#).
2. Browse to your IoT hub.
3. Select **Diagnostics settings**.
4. Select **Turn on diagnostics**.
5. Enable **Connections** logs to be collected.
6. For easier analysis, turn on **Send to Log Analytics** (see pricing). See the example under [Resolve connectivity errors](#).

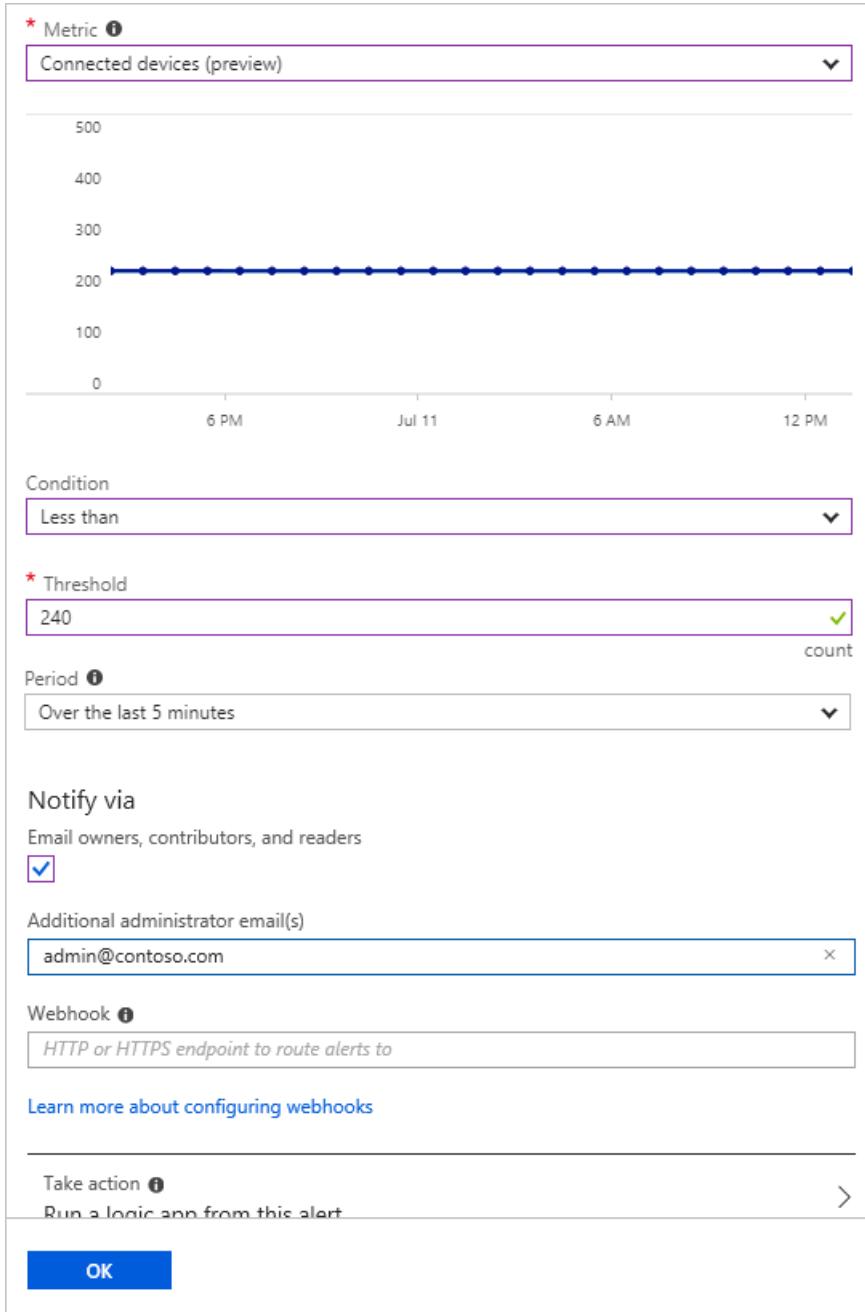


To learn more, see [Monitor the health of Azure IoT Hub and diagnose problems quickly](#).

Set up alerts for the **connected devices** count metric

To get alerts when devices disconnect, configure alerts on the **connected devices** metric.

1. Sign in to the [Azure portal](#).
2. Browse to your IoT hub.
3. Select **Alerts (classic)**.
4. Select **Add metric alert (classic)**.
5. Fill in the form and select **OK**.



To learn more, see [What are classic alerts in Microsoft Azure?](#).

Resolve connectivity errors

When you turn on diagnostic logs and alerts for connected devices, you get alerts when errors occur. This section describes how to resolve common issues when you receive an alert. The steps below assume you've set up Azure Monitor logs for your diagnostic logs.

1. Go to your workspace for **Log Analytics** in the Azure portal.
2. Select **Log Search**.
3. To isolate connectivity error logs for IoT Hub, enter the following query and then select **Run**:

```

search *
| where ( Type == "AzureDiagnostics" and ResourceType == "IOTHUBS")
| where ( Category == "Connections" and Level == "Error")

```

4. If there are results, look for `OperationName`, `ResultType` (error code), and `ResultDescription` (error message) to get more detail on the error.

search *	
where (Type == "AzureDiagnostics" and ResourceType == "IOTHUBS") where (Category == "Connections" and Level == "Error")	
2 Results List Table	
Drag a column header and drop it here to group by that column	
\$table	TimeGenerated
AzureDiagnostics	7/5/2018 1:48:49.000 PM
Type	AzureDiagnostics
ResultType	404104
ResultDescription	DeviceConnectionClosedRemotely
ResourceId	/SUBSCRIPTIONS/838AAEF4-XXXX-XXXX-XXXX-XXXXXXXXXX/RESOURCEGROUPS/US-WEST-SG/PROVIDERS/MICROSOFT.DEVICES/devices/AZ3166
OperationName	deviceDisconnect
Category	Connections
Level	Error
properties_s	{"deviceId": "AZ3166", "protocol": "Mqtt", "authType": null, "maskedIpAddress": "167.220.61.XXX", "statusCode": "404"}
deviceId	AZ3166
protocol	Mqtt
authType	null
maskedIpAddress	167.220.61.XXX
statusCode	404
location_s	westus
SubscriptionId	838aaef4-xxxx-xxxx-xxxx-xxxxxxxxxx
ResourceGroup	US-WEST-SG
ResourceProvider	MICROSOFT.DEVICES

5. Use this table to understand and resolve common errors.

ERROR	ROOT CAUSE	RESOLUTION
404104 DeviceConnectionClosedRemotely	The connection was closed by the device, but IoT Hub doesn't know why. Common causes include MQTT/AMQP timeout and internet connectivity loss.	Make sure the device can connect to IoT Hub by testing the connection . If the connection is fine, but the device disconnects intermittently, make sure to implement proper keep alive device logic for your choice of protocol (MQTT/AMQP).

ERROR	ROOT CAUSE	RESOLUTION
401003 IoTHubUnauthorized	IoT Hub couldn't authenticate the connection.	Make sure that the SAS or other security token you use isn't expired. Azure IoT SDKs automatically generate tokens without requiring special configuration.
409002 LinkCreationConflict	A device has more than one connection. When a new connection request comes for a device, IoT Hub closes the previous one with this error.	In the most common case, a device detects a disconnect and tries to reestablish the connection, but IoT Hub still considers the device connected. IoT Hub closes the previous connection and logs this error. This error usually appears as a side effect of a different, transient issue, so look for other errors in the logs to troubleshoot further. Otherwise, make sure to issue a new connection request only if the connection drops.
500001 ServerError	IoT Hub ran into a server-side issue. Most likely, the issue is transient. While the IoT Hub team works hard to maintain the SLA , small subsets of IoT Hub nodes can occasionally experience transient faults. When your device tries to connect to a node that's having issues, you receive this error.	To mitigate the transient fault, issue a retry from the device. To automatically manage retries , make sure you use the latest version of the Azure IoT SDKs . For best practice on transient fault handling and retries, see Transient fault handling . If the problem persists after retries, check Resource Health and Azure Status to see if IoT Hub has a known problem. If there's no known problems and the issue continues, contact support for further investigation.
500008 GenericTimeout	IoT Hub couldn't complete the connection request before timing out. Like a 500001 ServerError, this error is likely transient.	Follow troubleshooting steps for a 500001 ServerError to the root cause and resolve this error.

Other steps to try

If the previous steps didn't help, you can try:

- If you have access to the problematic devices, either physically or remotely (like SSH), follow the [device-side troubleshooting guide](#) to continue troubleshooting.
- Verify that your devices are **Enabled** in the Azure portal > your IoT hub > IoT devices.
- Get help from [Azure IoT Hub forum](#), [Stack Overflow](#), or [Azure support](#).

To help improve the documentation for everyone, leave a comment in the feedback section below if this guide didn't help you.

Next steps

- To learn more about resolving transient issues, see [Transient fault handling](#).
- To learn more about Azure IoT SDK and managing retries, see [How to manage connectivity and reliable messaging using Azure IoT Hub device SDKs](#).

IoT Hub operations monitoring (deprecated)

3/12/2019 • 6 minutes to read

IoT Hub operations monitoring enables you to monitor the status of operations on your IoT hub in real time. IoT Hub tracks events across several categories of operations. You can opt into sending events from one or more categories to an endpoint of your IoT hub for processing. You can monitor the data for errors or set up more complex processing based on data patterns.

NOTE

IoT Hub **operations monitoring is deprecated and has been removed from IoT Hub on March 10, 2019**. For monitoring the operations and health of IoT Hub, see [Monitor the health of Azure IoT Hub and diagnose problems quickly](#). For more information about the deprecation timeline, see [Monitor your Azure IoT solutions with Azure Monitor and Azure Resource Health](#).

IoT Hub monitors six categories of events:

- Device identity operations
- Device telemetry
- Cloud-to-device messages
- Connections
- File uploads
- Message routing

IMPORTANT

IoT Hub operations monitoring does not guarantee reliable or ordered delivery of events. Depending on IoT Hub underlying infrastructure, some events might be lost or delivered out of order. Use operations monitoring to generate alerts based on error signals such as failed connection attempts, or high-frequency disconnections for specific devices. You should not rely on operations monitoring events to create a consistent store for device state, e.g. a store tracking connected or disconnected state of a device.

How to enable operations monitoring

1. Create an IoT hub. You can find instructions on how to create an IoT hub in the [Get Started](#) guide.
2. Open the blade of your IoT hub. From there, click **Operations monitoring**.

The screenshot shows the Azure IoT Hub Overview page for the resource group 'getStartedWithAnIoT Hub_rg'. The left sidebar lists various settings like Overview, Activity log, Access control (IAM), Device Explorer, Shared access policies, Pricing and scale, IP Filter, Properties, Locks, and Automation script. The 'Pricing and scale' section is expanded, and the 'Operations monitoring' option is highlighted with a red box. The main pane displays the IoT Hub's status, including its hostname ('getStartedWithAnIoT Hub.azure-devices.net'), status ('Active'), location ('West US'), and IoT Hub units ('1'). Below this is a summary card with metrics: 3/24/2017 UTC, GETSTARTEDWITHANIOTHUB, 0% TOTAL, 28 / 8k messages, and 3 devices.

3. Select the monitoring categories you wish to monitor, and then click **Save**. The events are available for reading from the Event Hub-compatible endpoint listed in **Monitoring settings**. The IoT Hub endpoint is called `messages/operationsmonitoringevents`.

getStartedWithAnIoTHub - Operations monitoring

Save Discard

Search (Ctrl+ /)

Overview

Activity log

Access control (IAM)

SETTINGS

Properties

Locks

Automation script

GENERAL

Shared access policies

Pricing and scale

Operations monitoring

IP Filter

Diagnostics

MESSAGING

File upload

Endpoints

Monitoring categories

Device identity operations None

Device-to-cloud communications None

Cloud-to-device communications None

Connections None

File uploads None

Message routing None

Monitoring settings

Partitions 2

Event Hub-compatible name iothub-ehub-getstarted-96435-6074af59

Event Hub-compatible endpoint sb://ihsuprobyres052dednamespace.ser

The screenshot shows the 'Operations monitoring' section of the Azure IoT Hub configuration. On the left, a sidebar lists various monitoring categories: Overview, Activity log, Access control (IAM), Properties, Locks, Automation script, Shared access policies, Pricing and scale, Operations monitoring (which is selected and highlighted in blue), IP Filter, Diagnostics, File upload, and Endpoints. The main pane displays monitoring categories and their settings. Under 'Monitoring categories', there are sections for Device identity operations, Device-to-cloud communications, Cloud-to-device communications, and Connections. Each category has three monitoring levels: None (selected), Error, and Verbose. Below these are 'File uploads' and 'Message routing' sections with similar monitoring level controls. At the bottom, 'Monitoring settings' include 'Partitions' set to 2, and fields for 'Event Hub-compatible name' (iothub-ehub-getstarted-96435-6074af59) and 'Event Hub-compatible endpoint' (sb://ihsuprobyres052dednamespace.ser).

NOTE

Selecting **Verbose** monitoring for the **Connections** category causes IoT Hub to generate additional diagnostics messages. For all other categories, the **Verbose** setting changes the quantity of information IoT Hub includes in each error message.

Event categories and how to use them

Each operations monitoring category tracks a different type of interaction with IoT Hub, and each monitoring category has a schema that defines how events in that category are structured.

Device identity operations

The device identity operations category tracks errors that occur when you attempt to create, update, or delete an entry in your IoT hub's identity registry. Tracking this category is useful for provisioning scenarios.

```
{  
    "time": "UTC timestamp",  
    "operationName": "create",  
    "category": "DeviceIdentityOperations",  
    "level": "Error",  
    "statusCode": 4XX,  
    "statusDescription": "MessageDescription",  
    "deviceId": "device-ID",  
    "durationMs": 1234,  
    "userAgent": "userAgent",  
    "sharedAccessPolicy": "accessPolicy"  
}
```

Device telemetry

The device telemetry category tracks errors that occur at the IoT hub and are related to the telemetry pipeline. This category includes errors that occur when sending telemetry events (such as throttling) and receiving telemetry events (such as unauthorized reader). This category cannot catch errors caused by code running on the device itself.

```
{  
    "messageSizeInBytes": 1234,  
    "batching": 0,  
    "protocol": "Amqp",  
    "authType": "{\"scope\":\"device\",\"type\":\"sas\",\"issuer\":\"iothub\"}",  
    "time": "UTC timestamp",  
    "operationName": "ingress",  
    "category": "DeviceTelemetry",  
    "level": "Error",  
    "statusCode": 4XX,  
    "statusType": 4XX001,  
    "statusDescription": "MessageDescription",  
    "deviceId": "device-ID",  
    "EventProcessedUtcTime": "UTC timestamp",  
    "PartitionId": 1,  
    "EventEnqueuedUtcTime": "UTC timestamp"  
}
```

Cloud-to-device commands

The cloud-to-device commands category tracks errors that occur at the IoT hub and are related to the cloud-to-device message pipeline. This category includes errors that occur when sending cloud-to-device messages (such as unauthorized sender), receiving cloud-to-device messages (such as delivery count exceeded), and receiving cloud-to-device message feedback (such as feedback expired). This category does not catch errors from a device that improperly handles a cloud-to-device message if the cloud-to-device message was delivered successfully.

```
{
  "messageSizeInBytes": 1234,
  "authType": "{\"scope\":\"hub\", \"type\":\"sas\", \"issuer\":\"iothub\"}",
  "deliveryAcknowledgement": 0,
  "protocol": "Amqp",
  "time": " UTC timestamp",
  "operationName": "ingress",
  "category": "C2DCommands",
  "level": "Error",
  "statusCode": 4XX,
  "statusType": 4XX001,
  "statusDescription": "MessageDescription",
  "deviceId": "device-ID",
  "EventProcessedUtcTime": "UTC timestamp",
  "PartitionId": 1,
  "EventEnqueuedUtcTime": "UTC timestamp"
}
```

Connections

The connections category tracks errors that occur when devices connect or disconnect from an IoT hub. Tracking this category is useful for identifying unauthorized connection attempts and for tracking when a connection is lost for devices in areas of poor connectivity.

```
{
  "durationMs": 1234,
  "authType": "{\"scope\":\"hub\", \"type\":\"sas\", \"issuer\":\"iothub\"}",
  "protocol": "Amqp",
  "time": " UTC timestamp",
  "operationName": "deviceConnect",
  "category": "Connections",
  "level": "Error",
  "statusCode": 4XX,
  "statusType": 4XX001,
  "statusDescription": "MessageDescription",
  "deviceId": "device-ID"
}
```

File uploads

The file upload category tracks errors that occur at the IoT hub and are related to file upload functionality. This category includes:

- Errors that occur with the SAS URI, such as when it expires before a device notifies the hub of a completed upload.
- Failed uploads reported by the device.
- Errors that occur when a file is not found in storage during IoT Hub notification message creation.

This category cannot catch errors that directly occur while the device is uploading a file to storage.

```
{  
    "authType": "{\"scope\":\"hub\", \"type\":\"sas\", \"issuer\":\"iothub\"}",  
    "protocol": "HTTP",  
    "time": " UTC timestamp",  
    "operationName": "ingress",  
    "category": "fileUpload",  
    "level": "Error",  
    "statusCode": 4XX,  
    "statusType": 4XX001,  
    "statusDescription": "MessageDescription",  
    "deviceId": "device-ID",  
    "blobUri": "http://bloburi.com",  
    "durationMs": 1234  
}
```

Message routing

The message routing category tracks errors that occur during message route evaluation and endpoint health as perceived by IoT Hub. This category includes events such as when a rule evaluates to "undefined", when IoT Hub marks an endpoint as dead, and any other errors received from an endpoint. This category does not include specific errors about the messages themselves (such as device throttling errors), which are reported under the "device telemetry" category.

```
{  
    "messageSizeInBytes": 1234,  
    "time": "UTC timestamp",  
    "operationName": "ingress",  
    "category": "routes",  
    "level": "Error",  
    "deviceId": "device-ID",  
    "messageId": "ID of message",  
    "routeName": "myroute",  
    "endpointName": "myendpoint",  
    "details": "ExternalEndpointDisabled"  
}
```

Connect to the monitoring endpoint

The monitoring endpoint on your IoT hub is an Event Hub-compatible endpoint. You can use any mechanism that works with Event Hubs to read monitoring messages from this endpoint. The following sample creates a basic reader that is not suitable for a high throughput deployment. For more information about how to process messages from Event Hubs, see the [Get Started with Event Hubs](#) tutorial.

To connect to the monitoring endpoint, you need a connection string and the endpoint name. The following steps show you how to find the necessary values in the portal:

1. In the portal, navigate to your IoT Hub resource blade.
2. Choose **Operations monitoring**, and make a note of the **Event Hub-compatible name** and **Event Hub-compatible endpoint** values:

The screenshot shows the 'Operations monitoring' section of the Azure IoT Hub settings. The 'Operations monitoring' button is highlighted with a red box. On the right, under 'Monitoring settings', the 'Event Hub-compatible name' and 'Event Hub-compatible endpoint' fields are highlighted with red boxes.

3. Choose **Shared access policies**, then choose **service**. Make a note of the **Primary key** value:

The screenshot shows the 'Shared access policies' section of the Azure IoT Hub settings. The 'service' policy is selected and highlighted with a red box. The 'Primary key' value is also highlighted with a red box.

The following C# code sample is taken from a Visual Studio **Windows Classic Desktop** C# console app. The project has the **WindowsAzure.ServiceBus** NuGet package installed.

- Replace the connection string placeholder with a connection string that uses the **Event Hub-compatible**

endpoint and service **Primary key** values you noted previously as shown in the following example:

```
"Endpoint={your Event Hub-compatible endpoint};SharedAccessKeyName=service;SharedAccessKey={your  
service primary key value}"
```

- Replace the monitoring endpoint name placeholder with the **Event Hub-compatible name** value you noted previously.

```
class Program  
{  
    static string connectionString = "{your monitoring endpoint connection string}";  
    static string monitoringEndpointName = "{your monitoring endpoint name}";  
    static EventHubClient eventHubClient;  
  
    static void Main(string[] args)  
    {  
        Console.WriteLine("Monitoring. Press Enter key to exit.\n");  
  
        eventHubClient = EventHubClient.CreateFromConnectionString(connectionString,  
monitoringEndpointName);  
        var d2cPartitions = eventHubClient.GetRuntimeInformation().PartitionIds;  
        CancellationTokenSource cts = new CancellationTokenSource();  
        var tasks = new List<Task>();  
  
        foreach (string partition in d2cPartitions)  
        {  
            tasks.Add(ReceiveMessagesFromDeviceAsync(partition, cts.Token));  
        }  
  
        Console.ReadLine();  
        Console.WriteLine("Exiting...");  
        cts.Cancel();  
        Task.WaitAll(tasks.ToArray());  
    }  
  
    private static async Task ReceiveMessagesFromDeviceAsync(string partition, CancellationToken ct)  
    {  
        var eventHubReceiver = eventHubClient.GetDefaultConsumerGroup().CreateReceiver(partition,  
DateTime.UtcNow);  
        while (true)  
        {  
            if (ct.IsCancellationRequested)  
            {  
                await eventHubReceiver.CloseAsync();  
                break;  
            }  
  
            EventData eventData = await eventHubReceiver.ReceiveAsync(new TimeSpan(0,0,10));  
  
            if (eventData != null)  
            {  
                string data = Encoding.UTF8.GetString(eventData.GetBytes());  
                Console.WriteLine("Message received. Partition: {0} Data: '{1}'", partition, data);  
            }  
        }  
    }  
}
```

Next steps

To further explore the capabilities of IoT Hub, see:

- [IoT Hub developer guide](#)

- Deploying AI to edge devices with Azure IoT Edge

Migrate your IoT Hub from operations monitoring to diagnostics settings

3/12/2019 • 2 minutes to read

Customers using [operations monitoring](#) to track the status of operations in IoT Hub can migrate that workflow to [Azure diagnostics settings](#), a feature of Azure Monitor. Diagnostics settings supply resource-level diagnostic information for many Azure services.

The **operations monitoring functionality of IoT Hub is deprecated**, and has been removed from the portal. This article provides steps to move your workloads from operations monitoring to diagnostics settings. For more information about the deprecation timeline, see [Monitor your Azure IoT solutions with Azure Monitor and Azure Resource Health](#).

Update IoT Hub

To update your IoT Hub in the Azure portal, first turn on diagnostics settings, then turn off operations monitoring.

Enable logging with diagnostics settings

1. Sign in to the [Azure portal](#) and navigate to your IoT hub.
2. Select **Diagnostics settings**.
3. Select **Turn on diagnostics**.

The screenshot shows the 'Diagnostics settings' blade in the Azure portal. At the top, there are dropdown menus for Subscription (Visual Studio Enterprise with MSDN), Resource group (IoT), and Resource type (IoT Hub). Below these, the breadcrumb navigation shows Visual Studio Enterprise with MSDN > IoT > DiagnosticsHub. A red box highlights the 'Turn on diagnostics' section, which contains the text 'Turn on diagnostics to collect the following data.' followed by a bulleted list of data types: Connections, DeviceTelemetry, C2DCommands, DeviceIdentityOperations, FileUploadOperations, Routes, D2CTwinOperations, C2DTwinOperations, Twin Queries, JobsOperations, DirectMethods, and AllMetrics.

4. Give the diagnostic settings a name.
5. Choose where you want to send the logs. You can select any combination of the three options:
 - Archive to a storage account
 - Stream to an event hub
 - Send to Log Analytics
6. Choose which operations you want to monitor, and enable logs for those operations. The operations that diagnostic settings can report on are:
 - Connections

- Device telemetry
- Cloud-to-device messages
- Device identity operations
- File uploads
- Message routing
- Cloud-to-device twin operations
- Device-to-cloud twin operations
- Twin operations
- Job operations
- Direct methods
- Distributed tracing (preview)
- Configurations
- Device streams
- Device metrics

7. Save the new settings.

If you want to turn on diagnostics settings with PowerShell, use the following code:

```
Connect-AzureRmAccount
Select-AzureRmSubscription -SubscriptionName <subscription that includes your IoT Hub>
Set-AzureRmDiagnosticSetting -ResourceId <your resource Id> -ServiceBusRuleId <your service bus rule Id> -Enabled $true
```

New settings take effect in about 10 minutes. After that, logs appear in the configured archival target on the **Diagnostics settings** blade. For more information about configuring diagnostics, see [Collect and consume log data from your azure resources](#).

Turn off operations monitoring

NOTE

As of March 11, 2019, the operations monitoring feature is removed from IoT Hub's Azure portal interface. The steps below no longer apply. To migrate, make sure that the correct categories are turned on in Azure Monitor diagnostic settings above.

Once you test the new diagnostics settings in your workflow, you can turn off the operations monitoring feature.

1. In your IoT Hub menu, select **Operations monitoring**.
2. Under each monitoring category, select **None**.
3. Save the operations monitoring changes.

Update applications that use operations monitoring

The schemas for operations monitoring and diagnostics settings vary slightly. It's important that you update the applications that use operations monitoring today to map to the schema used by diagnostics settings.

Also, diagnostics settings offers five new categories for tracking. After you update applications for the existing schema, add the new categories as well:

- Cloud-to-device twin operations
- Device-to-cloud twin operations
- Twin queries
- Jobs operations

- Direct Methods

For the specific schema structures, see [Understand the schema for diagnostics settings](#).

Monitoring device connect and disconnect events with low latency

To monitor device connect and disconnect events in production, we recommend subscribing to the [device disconnected event](#) on Event Grid to get alerts and monitor the device connection state. Use this [tutorial](#) to learn how to integrate Device Connected and Device Disconnected events from IoT Hub in your IoT solution.

Next steps

[Monitor the health of Azure IoT Hub and diagnose problems quickly](#)

Summary of customer data request features

2/28/2019 • 2 minutes to read

The Azure IoT Hub is a REST API-based cloud service targeted at enterprise customers that enables secure, bi-directional communication between millions of devices and a partitioned Azure service.

NOTE

This article provides steps for how to delete personal data from the device or service and can be used to support your obligations under the GDPR. If you're looking for general info about GDPR, see the [GDPR section of the Service Trust portal](#).

Individual devices are assigned a device identifier (device ID) by a tenant administrator. Device data is based on the assigned device ID. Microsoft maintains no information and has no access to data that would allow device ID to user correlation.

Many of the devices managed in Azure IoT Hub are not personal devices, for example an office thermostat or factory robot. Customers may, however, consider some devices to be personally identifiable and at their discretion may maintain their own asset or inventory tracking methods that tie devices to individuals. Azure IoT Hub manages and stores all data associated with devices as if it were personal data.

Tenant administrators can use either the Azure portal or the service's REST APIs to fulfill information requests by exporting or deleting data associated with a device ID.

If you use the routing feature of the Azure IoT Hub service to forward device messages to other services, then data requests must be performed by the tenant admin for each routing endpoint in order to complete a full request for a given device. For more details, see the reference documentation for each endpoint. For more information about supported endpoints, see [Reference - IoT Hub endpoints](#).

If you use the Azure Event Grid integration feature of the Azure IoT Hub service, then data requests must be performed by the tenant admin for each subscriber of these events. For more information, see [React to IoT Hub events by using Event Grid](#).

If you use the Azure Monitor integration feature of the Azure IoT Hub service to create diagnostic logs, then data requests must be performed by the tenant admin against the stored logs. For more information, see [Monitor the health of Azure IoT Hub](#).

Deleting customer data

Tenant administrators can use the IoT devices blade of the Azure IoT Hub extension in the Azure portal to delete a device, which deletes the data associated with that device.

It is also possible to perform delete operations for devices using REST APIs. For more information, see [Service - Delete Device](#).

Exporting customer data

Tenant administrators can utilize copy and paste within the IoT devices blade of the Azure IoT Hub extension in the Azure portal to export data associated with a device.

It is also possible to perform export operations for devices using REST APIs. For more information, see [Service - Get Device](#).

NOTE

When you use Microsoft's enterprise services, Microsoft generates some information, known as system-generated logs. Some Azure IoT Hub system-generated logs are not accessible or exportable by tenant administrators. These logs constitute factual actions conducted within the service and diagnostic data related to individual devices.

Links to additional documentation

Full documentation for Azure IoT Hub Service APIs is located at [IoT Hub Service APIs](#).