

Contents

[IoT Hub Documentation](#)

[Overview](#)

[What is Azure IoT Hub?](#)

[Quickstarts](#)

[Send telemetry](#)

[Get started](#)

[C](#)

[Node.js](#)

[.NET](#)

[Java](#)

[Python](#)

[Android](#)

[iOS](#)

[Xamarin.Forms](#)

[Control a device](#)

[Node.js](#)

[.NET](#)

[Java](#)

[Python](#)

[Android](#)

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

[.NET](#)

[Node.js](#)

[C](#)

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

[.NET](#)

[Node.js](#)

[C](#)

[Tutorials](#)

Routing messages

Part 1 - Configure message routing

Portal

Azure RM template

Azure CLI

PowerShell

Part 2 - View message routing results

Message enrichments

Use metrics and diagnostic logs

Perform manual failover

Configure your devices

Manage firmware updates

Test device connectivity

Samples

Azure Policy built-ins

Concepts

TLS support

Virtual networks support

Message enrichments overview

Overview of device management

Compare IoT Hub and Event Hubs

Choose the right tier

High availability and disaster recovery

How to clone an IoT Hub to another region

Understanding IoT hub IP address

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

Cloud-to-device feature guide

Send and receive messages

[Use message routing to send device-to-cloud messages](#)

[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](#)

[AMQP support](#)

[Glossary](#)

Security

[Recommendations](#)

[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](#)

[Azure IoT C SDK resource information](#)

[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.js](#)

[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](#)
- [Manage public network access](#)
- [Automatic device management at scale](#)
 - [Use Azure portal](#)
 - [Use Azure CLI](#)
- [Bulk import and export 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](#)
 - [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 in a web app](#)
 - [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 and problem solution](#)
 - [Device disconnects](#)
 - [Troubleshoot message routing](#)
 - [Problem resolution](#)

400027 ConnectionForcefullyClosedOnNewConnection
401003 IoTHubUnauthorized
403002 IoTHubQuotaExceeded
403004 DeviceMaximumQueueDepthExceeded
403006 DeviceMaximumActiveFileUploadLimitExceeded
404001 DeviceNotFound
404103 DeviceNotOnline
404104 DeviceConnectionClosedRemotely
409001 DeviceAlreadyExists
409002 LinkCreationConflict
412002 DeviceMessageLockLost
429001 ThrottlingException
500xxx Internal errors
503003 PartitionNotFound
504101 GatewayTimeout

Reference

[Azure CLI](#)
[Azure PowerShell](#)
[.NET \(Device\)](#)
[.NET \(Service\)](#)
[.NET \(Management\)](#)
[Java \(Device\)](#)
[Java \(Service\)](#)
[Node.js \(Device\)](#)
[Node.js \(Service\)](#)
[Node.js \(Management\)](#)
[Python \(Device\)](#)
[Python \(Service\)](#)
[Python \(Management\)](#)
[C device SDK](#)
[REST \(Device\)](#)
[REST \(Service\)](#)

- [REST \(IoT Hub Resource\)](#)
- [REST \(Certificates\)](#)
- [Resource Manager template](#)
- [Feature and API retirement](#)
- [Operations monitoring](#)
- [Migrate to diagnostics settings](#)
- [TLS 1.0 and 1.1 deprecation](#)
- [Resources](#)
 - [Support and help options](#)
 - [Azure IoT services](#)
 - [IoT Hub](#)
 - [IoT Hub Device Provisioning Service](#)
 - [IoT Central](#)
 - [IoT Edge](#)
 - [IoT solution accelerators](#)
 - [IoT Plug and Play](#)
 - [Azure Maps](#)
 - [Time Series Insights](#)
 - [Azure IoT SDKs](#)
 - [IoT Service SDKs](#)
 - [IoT Device SDKs](#)
 - [Azure IoT samples](#)
 - [C# \(.NET\)](#)
 - [Node.js](#)
 - [Java](#)
 - [Python](#)
 - [iOS Platform](#)
 - [Azure Certified for IoT device catalog](#)
 - [Azure IoT Developer Center](#)
 - [Customer data requests](#)
 - [Azure Roadmap](#)
 - [Azure IoT Tools](#)

[Azure IoT Explorer tool](#)

[iothub-diagnostics tool](#)

[Pricing](#)

[Pricing calculator](#)

[Service updates](#)

[Technical case studies](#)

[Videos](#)

What is Azure IoT Hub?

7/29/2020 • 3 minutes to read • [Edit Online](#)

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. For more information about scaling your IoT Hub, see [IoT Hub Scaling](#). To learn more about the multiple tiers of service offered by IoT Hub and how to best fit your scalability needs, check 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 information, see [IoT Hub Quotas and Throttling](#). For more details on quota limits, see one of the following articles:

- [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](#)

To learn more about the ways you can build and deploy IoT solutions with Azure IoT, visit:

- [Fundamentals: Azure IoT technologies and solutions.](#)

Quickstart: Send telemetry from a device to an IoT hub and monitor it with the Azure CLI

7/29/2020 • 8 minutes to read • [Edit Online](#)

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 use the Azure CLI to create an IoT Hub and a simulated device, send device telemetry to the hub, and send a cloud-to-device message. You also use the Azure portal to visualize device metrics. This is a basic workflow for developers who use the CLI to interact with an IoT Hub application.

Prerequisites

- If you don't have an Azure subscription, [create one for free](#) before you begin.
- Azure CLI. You can run all commands in this quickstart using the Azure Cloud Shell, an interactive CLI shell that runs in your browser. If you use the Cloud Shell, you don't need to install anything. If you prefer to use the CLI locally, this quickstart requires Azure CLI version 2.0.76 or later. Run `az --version` to find the version. To install or upgrade, see [Install Azure CLI](#).

Sign in to the Azure portal

Sign in to the Azure portal at <https://portal.azure.com>.

Regardless whether you run the CLI locally or in the Cloud Shell, keep the portal open in your browser. You use it later in this quickstart.

Launch the Cloud Shell

In this section, you launch an instance of the Azure Cloud Shell. If you use the CLI locally, skip to the section [Prepare two CLI sessions](#).

To launch the Cloud Shell:

1. Select the **Cloud Shell** button on the top-right menu bar in the Azure portal.



NOTE

If this is the first time you've used the Cloud Shell, it prompts you to create storage, which is required to use the Cloud Shell. Select a subscription to create a storage account and Microsoft Azure Files share.

2. Select your preferred CLI environment in the **Select environment** dropdown. This quickstart uses the **Bash** environment. All the following CLI commands work in the Powershell environment too.

A screenshot of the Azure Cloud Shell interface. The title bar says "PowerShell" and "Bash". The status bar at the top right says "Cloud Shell.Succeeded." and "terminal...". A red box highlights the "Bash" tab. Below the tabs, a message says "MOTD: Connect to a remote Azure VM: Enter-AzVM". The main area shows command-line output: "VERBOSE: Authenticating to Azure ...", "VERBOSE: Building your Azure drive ...", "Azure:/", and "PS Azure:\> []".

Prepare two CLI sessions

In this section, you prepare two Azure CLI sessions. If you're using the Cloud Shell, you will run the two sessions in separate browser tabs. If using a local CLI client, you will run two separate CLI instances. You'll use the first session as a simulated device, and the second session to monitor and send messages. To run a command, select **Copy** to copy a block of code in this quickstart, paste it into your shell session, and run it.

Azure CLI requires you to be logged into your Azure account. All communication between your Azure CLI shell session and your IoT hub is authenticated and encrypted. As a result, this quickstart does not need additional authentication that you'd use with a real device, such as a connection string.

- Run the `az extension add` command to add the Microsoft Azure IoT Extension for Azure CLI to your CLI shell. The IOT Extension adds IoT Hub, IoT Edge, and IoT Device Provisioning Service (DPS) specific commands to Azure CLI.

```
az extension add --name azure-iot
```

After you install the Azure IOT extension, you don't need to install it again in any Cloud Shell session.

NOTE

This article uses the newest version of the Azure IoT extension, called `azure-iot`. The legacy version is called `azure-cli-iot-ext`. You should only have one version installed at a time. You can use the command `az extension list` to validate the currently installed extensions.

Use `az extension remove --name azure-cli-iot-ext` to remove the legacy version of the extension.

Use `az extension add --name azure-iot` to add the new version of the extension.

To see what extensions you have installed, use `az extension list`.

- Open a second CLI session. If you're using the Cloud Shell, select **Open new session**. If you're using the CLI locally, open a second instance.

A screenshot of the Azure Cloud Shell interface. The title bar says "Bash" and has a "New Session" button highlighted with a red box. The status bar at the top right says "Requesting a Cloud Shell.Succeeded." and "Connecting terminal...". A tooltip for the "New Session" button says "Open new session". Below the title bar, it says "Welcome to Azure Cloud Shell". The main area shows command-line output: "Type "az" to use Azure CLI", "Type "help" to learn about Cloud Shell", "tim@Azure:~\$ az extension add --name azure-iot", and "tim@Azure:~\$..".

Create an IoT Hub

In this section, you use the Azure CLI to create a resource group and an IoT Hub. An Azure resource group is a logical container into which Azure resources are deployed and managed. An IoT Hub acts as a central message hub for bi-directional communication between your IoT application and the devices.

TIP

Optionally, you can create an Azure resource group, an IoT Hub, and other resources by using the [Azure portal](#), [Visual Studio Code](#), or other programmatic methods.

- Run the [az group create](#) command to create a resource group. The following command creates a resource group named *MyResourceGroup* in the *eastus* location.

```
az group create --name MyResourceGroup --location eastus
```

- Run the [az iot hub create](#) command to create an IoT hub. It might take a few minutes to create an IoT hub.

YourIoTHubName. Replace this placeholder below with the name you chose for your IoT hub. An IoT hub name must be globally unique in Azure. This placeholder is used in the rest of this quickstart to represent your IoT hub name.

```
az iot hub create --resource-group MyResourceGroup --name {YourIoTHubName}
```

Create and monitor a device

In this section, you create a simulated device in the first CLI session. The simulated device sends device telemetry to your IoT hub. In the second CLI session, you monitor events and telemetry, and send a cloud-to-device message to the simulated device.

To create and start a simulated device:

- Run the [az iot hub device-identity create](#) command in the first CLI session. This creates the simulated device identity.

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

simDevice. You can use this name directly for the simulated device in the rest of this quickstart. Optionally, use a different name.

```
az iot hub device-identity create --device-id simDevice --hub-name {YourIoTHubName}
```

- Run the [az iot device simulate](#) command in the first CLI session. This starts the simulated device. The device sends telemetry to your IoT hub and receives messages from it.

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

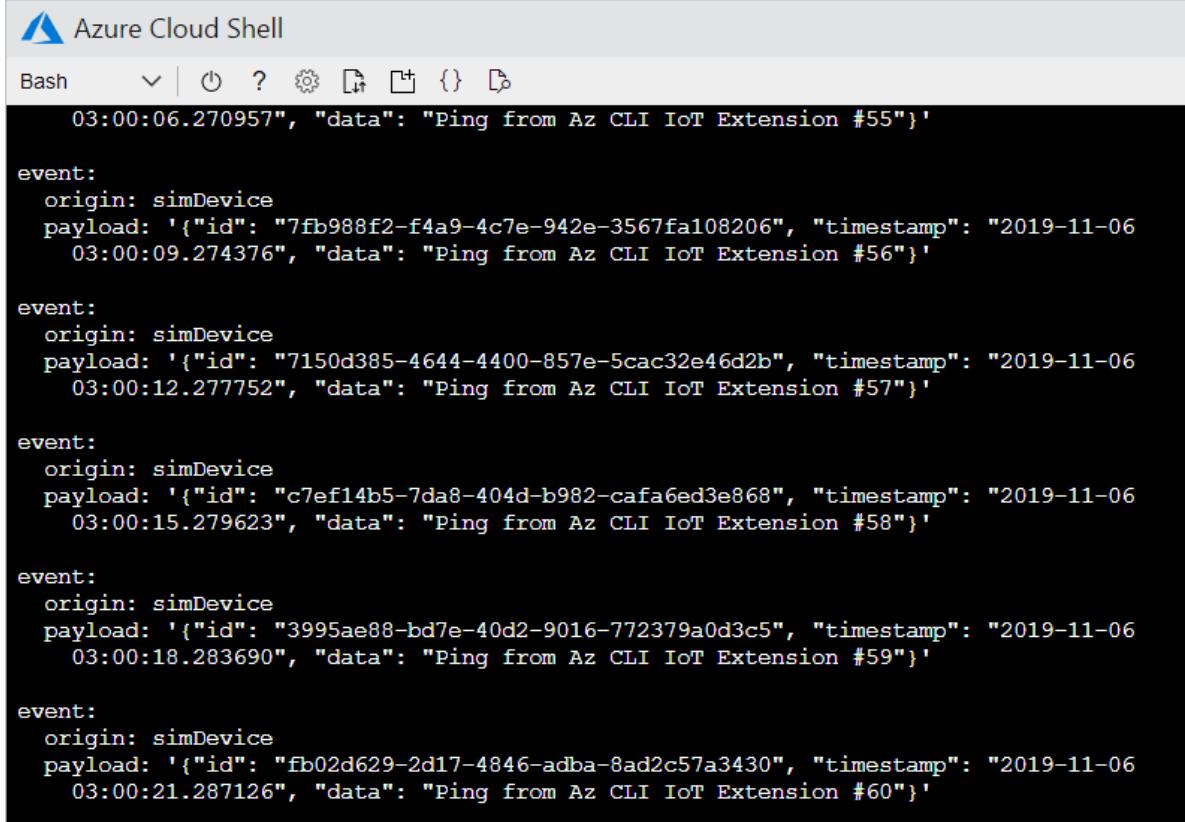
```
az iot device simulate -d simDevice -n {YourIoTHubName}
```

To monitor a device:

- In the second CLI session, run the [az iot hub monitor-events](#) command. This starts monitoring the simulated device. The output shows telemetry that the simulated device sends to the IoT hub.

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

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



The screenshot shows the Azure Cloud Shell interface with a Bash session. The command `az iot hub monitor-events --output table --hub-name {YourIoTHubName}` is run, and the output displays six simulated device ping messages. Each message includes a timestamp, a data payload indicating a ping from the Az CLI IoT Extension, and a unique device ID and timestamp.

```
03:00:06.270957", "data": "Ping from Az CLI IoT Extension #55"}'

event:
origin: simDevice
payload: '{"id": "7fb988f2-f4a9-4c7e-942e-3567fa108206", "timestamp": "2019-11-06
03:00:09.274376", "data": "Ping from Az CLI IoT Extension #56"}'

event:
origin: simDevice
payload: '{"id": "7150d385-4644-4400-857e-5cac32e46d2b", "timestamp": "2019-11-06
03:00:12.277752", "data": "Ping from Az CLI IoT Extension #57"}'

event:
origin: simDevice
payload: '{"id": "c7ef14b5-7da8-404d-b982-cafa6ed3e868", "timestamp": "2019-11-06
03:00:15.279623", "data": "Ping from Az CLI IoT Extension #58"}'

event:
origin: simDevice
payload: '{"id": "3995ae88-bd7e-40d2-9016-772379a0d3c5", "timestamp": "2019-11-06
03:00:18.283690", "data": "Ping from Az CLI IoT Extension #59"}'

event:
origin: simDevice
payload: '{"id": "fb02d629-2d17-4846-adba-8ad2c57a3430", "timestamp": "2019-11-06
03:00:21.287126", "data": "Ping from Az CLI IoT Extension #60"}'
```

- After you monitor the simulated device in the second CLI session, press **Ctrl+C** to stop monitoring.

Use the CLI to send a message

In this section, you use the second CLI session to send a message to the simulated device.

- In the first CLI session, confirm that the simulated device is running. If the device has stopped, run the following command to start it:

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

```
az iot device simulate -d simDevice -n {YourIoTHubName}
```

- In the second CLI session, run the `az iot device c2d-message send` command. This sends a cloud-to-device message from your IoT hub to the simulated device. The message includes a string and two key-value pairs.

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

```
az iot device c2d-message send -d simDevice --data "Hello World" --props "key0=value0;key1=value1" -n {YourIoTHubName}
```

Optionally, you can send cloud-to-device messages by using the Azure portal. To do this, browse to the overview page for your IoT Hub, select **IoT Devices**, select the simulated device, and select **Message to Device**.

- In the first CLI session, confirm that the simulated device received the message.

```
Bash    ▾ | ⌁ ? ⌘ ⌄ ⌅ ⌆ ⌇ ⌈ ⌉ ⌊ ⌋
tim@Azure:~$ az iot device simulate -d simDevice -r {YourIoTHub}
Connected to target IoT Hub MQTT broker with result: success
Subscribed to device bound message queue
.....
_Received C2D message with topic_: devices/simDevice/messages/devicebound/%24.mid=c945124b-9304-46f
.Payload_: b'Hello World'
..|
```

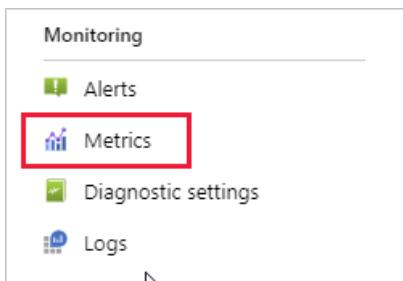
4. After you view the message, close the second CLI session. Keep the first CLI session open. You use it to clean up resources in a later step.

View messaging metrics in the portal

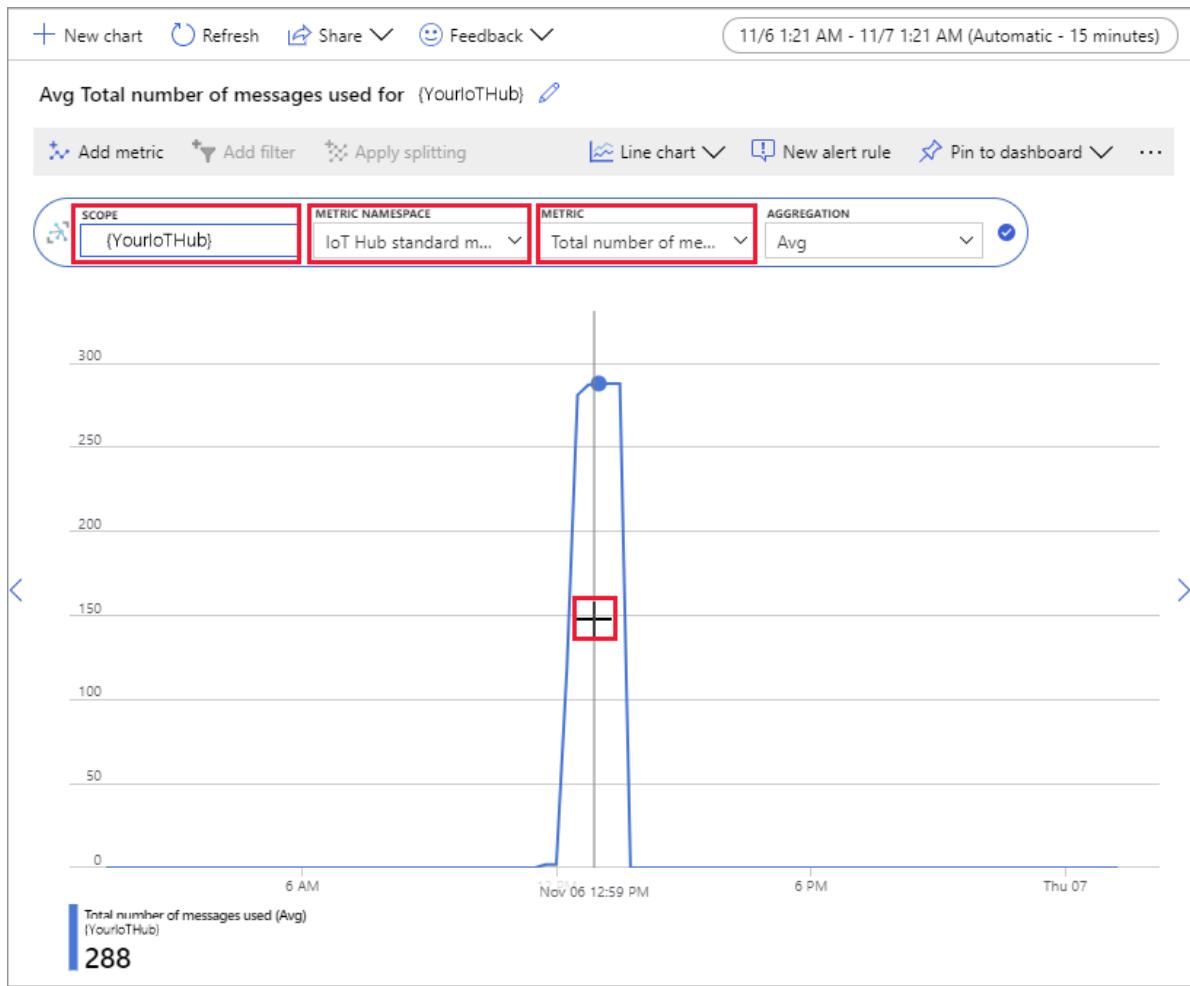
The Azure portal enables you to manage all aspects of your IoT Hub and devices. In a typical IoT Hub application that ingests telemetry from devices, you might want to monitor devices or view metrics on device telemetry.

To visualize messaging metrics in the Azure portal:

1. In the left navigation menu on the portal, select **All Resources**. This lists all resources in your subscription, including the IoT hub you created.
2. Select the link on the IoT hub you created. The portal displays the overview page for the hub.
3. Select **Metrics** in the left pane of your IoT Hub.



4. Enter your IoT hub name in **Scope**.
5. Select *IoT Hub Standard Metrics* in **Metric Namespace**.
6. Select *Total number of messages used* in **Metric**.
7. Hover your mouse pointer over the area of the timeline in which your device sent messages. The total number of messages at a point in time appears in the lower left corner of the timeline.



8. Optionally, use the **Metric** dropdown to display other metrics on your simulated device. For example, *C2d message deliveries completed* or *Total devices (preview)*.

Clean up resources

If you no longer need the Azure resources created in this quickstart, you can use the Azure CLI to delete them.

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

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.

To delete a resource group by name:

1. Run the [az group delete](#) command. This removes the resource group, the IoT Hub, and the device registration you created.

```
az group delete --name MyResourceGroup
```

2. Run the [az group list](#) command to confirm the resource group is deleted.

```
az group list
```

Next steps

In this quickstart, you used the Azure CLI to create an IoT hub, create a simulated device, send telemetry, monitor telemetry, send a cloud-to-device message, and clean up resources. You used the Azure portal to visualize messaging metrics on your device.

If you are a device developer, the suggested next step is to see the telemetry quickstart that uses the Azure IoT Device SDK for C. Optionally, see one of the available Azure IoT Hub telemetry quickstart articles in your preferred language or SDK.

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

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

7/29/2020 • 11 minutes to read • [Edit Online](#)

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 is written for Windows, but you can complete this quickstart on Linux as well.

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|---|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal . |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

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

Prerequisites

- Install [Visual Studio 2019](#) with the '**Desktop development with C++**' workload enabled.

- Install the latest version of [Git](#).
- Make sure that port 8883 is open in your firewall. The device sample in this quickstart uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).
- Run the following command to add the Microsoft Azure IoT Extension for Azure CLI to your Cloud Shell instance. The IoT Extension adds IoT Hub, IoT Edge, and IoT Device Provisioning Service (DPS) specific commands to Azure CLI.

```
az extension add --name azure-iot
```

NOTE

This article uses the newest version of the Azure IoT extension, called `azure-iot`. The legacy version is called `azure-cli-iot-ext`. You should only have one version installed at a time. You can use the command `az extension list` to validate the currently installed extensions.

Use `az extension remove --name azure-cli-iot-ext` to remove the legacy version of the extension.

Use `az extension add --name azure-iot` to add the new version of the extension.

To see what extensions you have installed, use `az extension list`.

Prepare the development environment

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

For the following environments, you can use the SDK by installing these packages and libraries:

- **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've 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're 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'll 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 [CMake build system](#).

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. Find the tag name for the [latest release](#) of the SDK.

3. Open a command prompt or Git Bash shell. Run the following commands to clone the latest release of the

Azure IoT C SDK GitHub repository. Use the tag you found in the previous step as the value for the `-b` parameter:

```
git clone -b <release-tag> https://github.com/Azure/azure-iot-sdk-c.git
cd azure-iot-sdk-c
git submodule update --init
```

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. Run the following commands from the `azure-iot-sdk-c` directory:

```
mkdir cmake
cd cmake
```

5. Run the following command to build 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` doesn'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
```

Create an IoT hub

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

1. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to

you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.

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

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

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 that, the title 'IoT hub' is displayed with the Microsoft logo. A red box highlights the 'Project details' section, which includes fields for Subscription, Resource group, Region, and IoT hub name. The 'Subscription' field is set to 'Personal IoT items'. The 'Resource group' field has a dropdown menu with 'Create new' option. The 'Region' field is set to 'East Asia'. The 'IoT hub name' field contains the placeholder 'Once your hub is created, this name can't be changed'. At the bottom of the form, there are navigation buttons: 'Review + create' (in blue), '< Previous', 'Next: Size and scale >' (which is highlighted with a red box), and 'Automation options'.

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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
Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center On
Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.
If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure](#)

resources.

The screenshot shows the 'Tags' tab selected in the top navigation bar of the 'IoT hub' creation wizard. Below the tabs, a note explains that tags are name/value pairs used for categorization and billing. A table lists two tags: 'department' with value 'accounting' and another row with empty fields. At the bottom are buttons for 'Review + create', 'Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' page with a red box around the 'Review + create' button in the top navigation. The page displays configuration details under three sections: Basics, Size and scale, and Tags. The 'Create' button at the bottom left is also highlighted with a red box.

| Section | Setting | Value |
|----------------|----------------------------|--------------------------------|
| Basics | Subscription | Personal testing |
| | Resource group | iot-hubs |
| | Region | West US 2 |
| | IoT hub name | you-hub-name |
| 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 |
| | Azure Security Center | 0.001 USD per device per month |
| | Tags | |
| Tags | department: accounting | |

8. Select **Create** to create your new 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'll use the Azure Cloud

Shell with the [IoT extension](#) to register a simulated device.

1. Run the following command in Azure Cloud Shell to create the device identity.

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

MyCDevice: This is the name of the device you're registering. It's recommended to use **MyCDevice** as shown. If you choose a different name for your device, you'll also need to use that name throughout this article, and update the device name in the sample applications before you run them.

```
az iot hub device-identity create --hub-name {YourIoTHubName} --device-id MyCDevice
```

2. Run the following command 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 MyCDevice --output table
```

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

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

You'll 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 under the working directory where you cloned the Azure IoT C SDK:

```
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 earlier. 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. Enter the following command from your working directory:

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

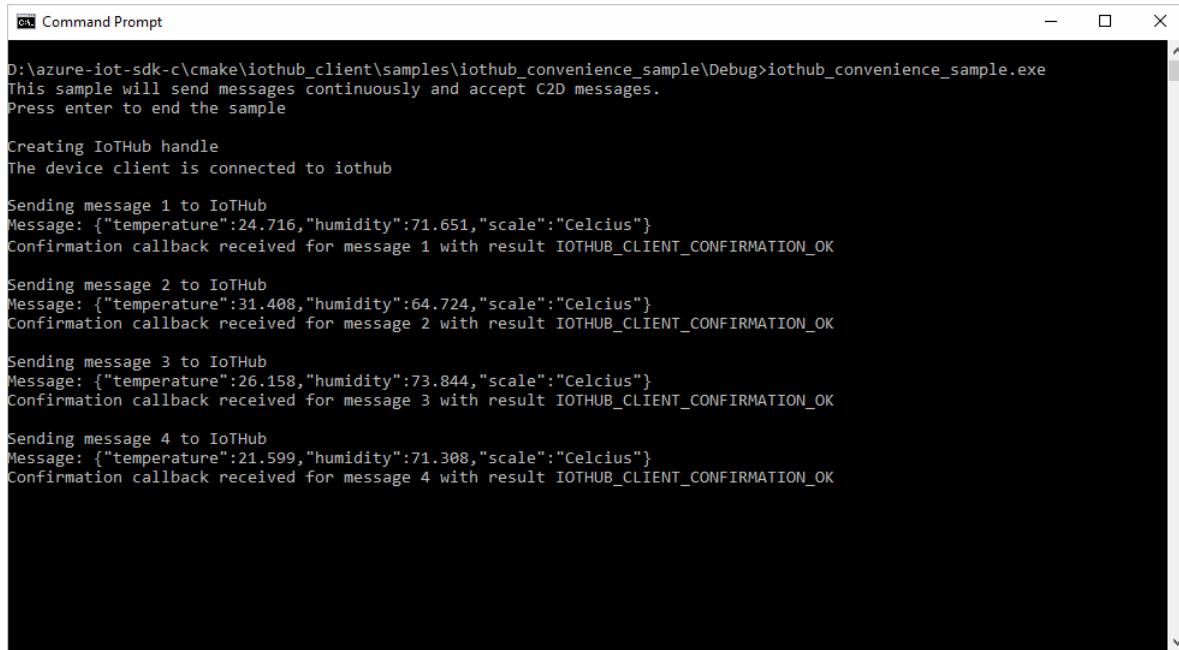
4. Run CMake in your local terminal window to build the sample with your updated `connectionString` value:

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

5. In your 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 displays the output of a C++ application named "iothub_convenience_sample.exe". The application logs messages indicating it is sending four temperature and humidity messages to an IoT Hub, with each message being confirmed successfully.

```
O:\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 IoTHub handle
The device client is connected to iothub

Sending message 1 to IoTHub
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 IoTHub
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 IoTHub
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 IoTHub
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'll 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.

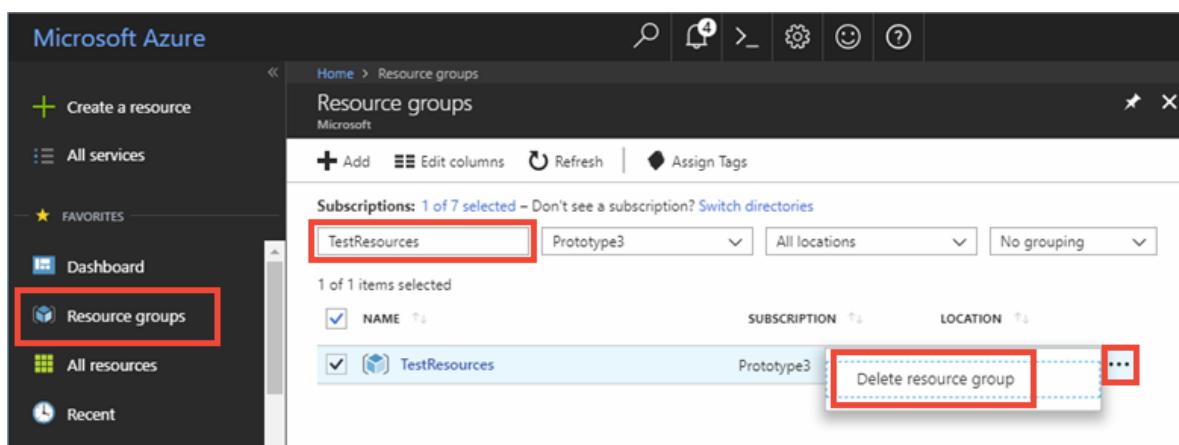
Otherwise, you can delete the Azure resources created in this article to avoid charges.

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 select **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, select ... 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 select **Delete**. After a few moments, the resource group and all of its contained

resources are deleted.

Next steps

In this quickstart, you 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.

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)

7/29/2020 • 9 minutes to read • [Edit Online](#)

In this quickstart, you send telemetry from a simulated device application through Azure IoT Hub to a back-end application for processing. 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. This 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.

Prerequisites

- An Azure account with an active subscription. [Create one for free](#).
- [Node.js 10+](#). If you are using the Azure Cloud Shell, do not update the installed version of Node.js. The Azure Cloud Shell already has the latest Node.js version.
- [A sample Node.js project](#).
- Port 8883 open in your firewall. The device sample in this quickstart uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

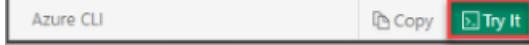
You can verify the current version of Node.js on your development machine using the following command:

```
node --version
```

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|---|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |

| OPTION | EXAMPLE/LINK |
|---|--|
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal . |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

Add Azure IoT Extension

Run the following command to add the Microsoft Azure IoT Extension for Azure CLI to your Cloud Shell instance. The IoT Extension adds IoT Hub, IoT Edge, and IoT Device Provisioning Service (DPS) specific commands to Azure CLI.

```
az extension add --name azure-iot
```

NOTE

This article uses the newest version of the Azure IoT extension, called `azure-iot`. The legacy version is called `azure-cli-iot-ext`. You should only have one version installed at a time. You can use the command `az extension list` to validate the currently installed extensions.

Use `az extension remove --name azure-cli-iot-ext` to remove the legacy version of the extension.

Use `az extension add --name azure-iot` to add the new version of the extension.

To see what extensions you have installed, use `az extension list`.

Create an IoT hub

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

1. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.

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

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

The screenshot shows the 'Basics' step of the IoT hub creation wizard. It includes fields for Subscription, Resource group, Region, and IoT hub name, all enclosed in a red box. Below the form are navigation buttons: 'Review + create' (blue), '< Previous' (disabled), 'Next: Size and scale >' (highlighted with a red box), and 'Automation options'.

Home > New > IoT hub

IoT hub
Microsoft

Basics [Size and scale](#) [Tags](#) [Review + create](#)

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

Project details

Choose the subscription you'll use to manage deployments and costs. Use resource groups like folders to help you organize and manage resources.

Subscription * ⓘ Personal IoT items

Resource group * ⓘ Create new

Region * ⓘ East Asia

IoT hub name * ⓘ Once your hub is created, this name can't be changed

[Review + create](#) [< Previous](#) [Next: Size and scale >](#) [Automation options](#)

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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
Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center [On](#) Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.

If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to

categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure resources](#).

The screenshot shows the 'Tags' tab of the IoT hub configuration page. It displays a table with two rows of tag entries. The first row has 'Name' as 'department' and 'Value' as 'accounting'. The second row is empty. Below the table are buttons for 'Review + create', '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

| Name | Value | Resource | Actions |
|------------|------------|----------|---------|
| department | accounting | IoT Hub | ... |
| | | IoT Hub | |

Review + create < Previous: Size and scale Next: Review + create > Automation options

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' step of the IoT hub creation process. It lists the configuration details with the 'Review + create' tab highlighted by a red box. The configuration includes:

- Basics**:
 - Subscription: Personal testing
 - Resource group: iot-hubs
 - Region: West US 2
 - IoT hub name: you-hub-name
- 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
 - Azure Security Center: 0.001 USD per device per month
- Tags**:
 - department: accounting

At the bottom, the 'Create' button is highlighted with a red box, and other buttons include '< Previous: Tags', 'Next >', and 'Automation options'.

8. Select **Create** to create your new 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 command in Azure Cloud Shell to create the device identity.

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

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

```
az iot hub device-identity create --hub-name {YourIoTHubName} --device-id MyNodeDevice
```

2. Run the following command 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'll use this value later in the quickstart.

3. You also need the *Event Hubs-compatible endpoint*, *Event Hubs-compatible path*, and *service 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 service --query primaryKey --hub-name {YourIoTHubName}
```

Make a note of these three values, which you'll 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.

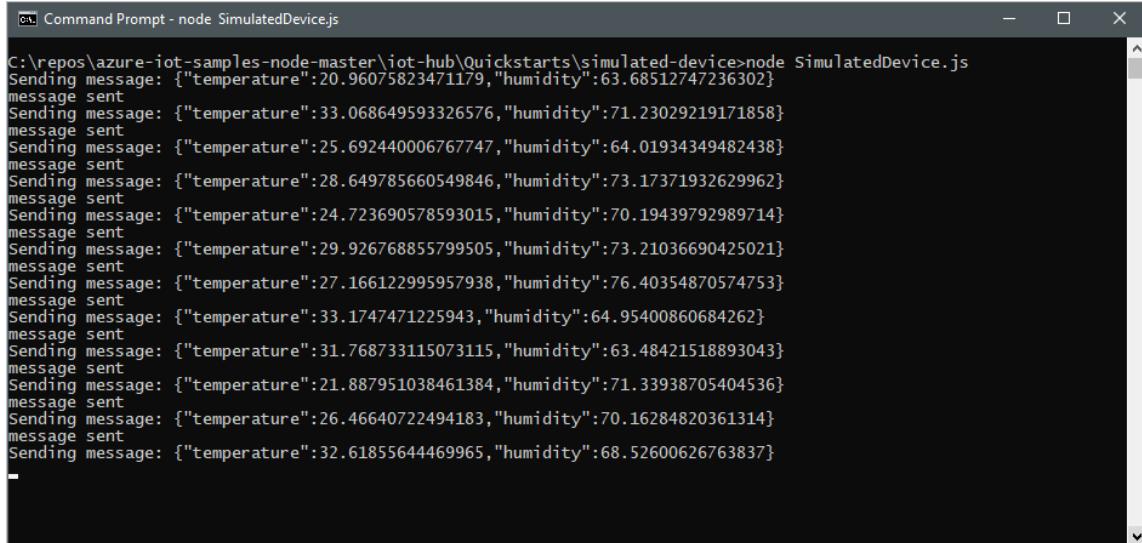
1. Open your local terminal window, navigate to the root folder of the sample Node.js project. Then navigate to the `iot-hub\Quickstarts\simulated-device` folder.
2. Open the `SimulatedDevice.js` file in a text editor of your choice.

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

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.

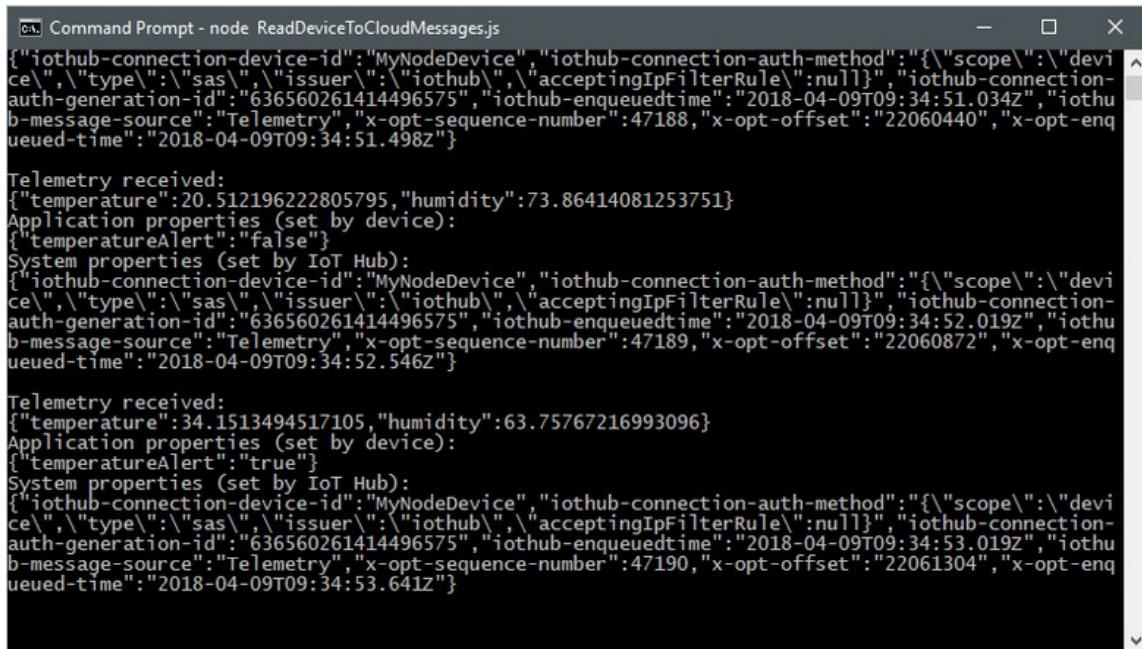
1. Open another local terminal window, navigate to the root folder of the sample Node.js project. Then navigate to the **iot-hub\Quickstarts\read-d2c-messages** folder.
2. Open the **ReadDeviceToCloudMessages.js** file in a text editor of your choice. Update the following variables and save your changes 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 earlier. |
| <code>eventHubsCompatiblePath</code> | Replace the value of the variable with the Event Hubs-compatible path you made a note of earlier. |
| <code>iotHubSasKey</code> | Replace the value of the variable with the service primary key you made a note of earlier. |

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-enqueuedtime": "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-enqueuedtime": "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-enqueuedtime": "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.

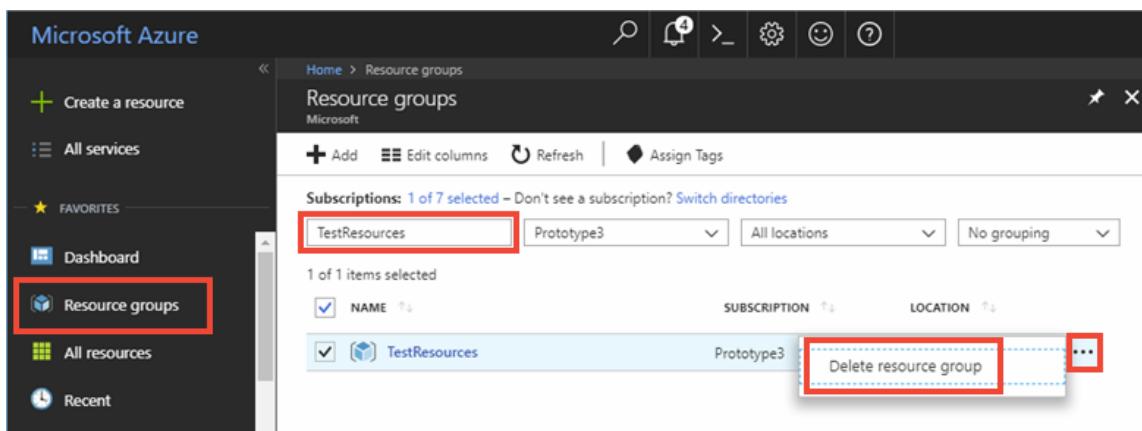
Otherwise, you can delete the Azure resources created in this article to avoid charges.

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 select **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, select ... 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 select **Delete**. After a few moments, the resource group and all of its contained resources are deleted.

Next steps

In this quickstart, you set up 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 (.NET)

7/29/2020 • 10 minutes to read • [Edit Online](#)

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.

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|--|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal . |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the Copy button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

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 3.0 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
```

NOTE

.NET Core SDK 3.0 or greater is recommended to compile the Event Hubs service code used to read telemetry in this quickstart. You can use .NET Core SDK 2.1 if you set the language version for the service code to preview as noted in the [Read the telemetry from your hub](#) section.

Run the following command to add the Microsoft Azure IoT Extension for Azure CLI to your Cloud Shell instance. The IOT Extension adds IoT Hub, IoT Edge, and IoT Device Provisioning Service (DPS) specific commands to Azure CLI.

```
az extension add --name azure-iot
```

NOTE

This article uses the newest version of the Azure IoT extension, called `azure-iot`. The legacy version is called `azure-cli-iot-ext`. You should only have one version installed at a time. You can use the command `az extension list` to validate the currently installed extensions.

Use `az extension remove --name azure-cli-iot-ext` to remove the legacy version of the extension.

Use `az extension add --name azure-iot` to add the new version of the extension.

To see what extensions you have installed, use `az extension list`.

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

Make sure that port 8883 is open in your firewall. The device sample in this quickstart uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

Create an IoT hub

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

1. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to

you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.

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

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

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 that is the title 'IoT hub' and the provider 'Microsoft'. A red box highlights the 'Project details' section, which includes fields for Subscription (Personal IoT items), Resource group (Create new), Region (East Asia), and IoT hub name (Once your hub is created, this name can't be changed). At the bottom of the screen, there are navigation buttons: 'Review + create' (blue), '< Previous' (disabled), 'Next: Size and scale >' (highlighted with a red box), and 'Automation options'.

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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
Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center On
Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.
If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure resources](#).

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags **Review + create**

Tags are name/value pairs. To categorize resources and consolidate billing, apply the same tag to multiple resources and resource groups. Your tags will update automatically if you change your resources. [Learn more](#)

| Name ⓘ | Value ⓘ | Resource | ⋮ |
|------------|--------------|----------|---|
| department | : accounting | IoT Hub | |
| | : | IoT Hub | |

Review + create < Previous: Size and scale Next: Review + create > Automation options

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags **Review + create**

Basics

| | |
|----------------|------------------|
| Subscription | Personal testing |
| Resource group | iot-hubs |
| Region | West US 2 |
| IoT hub name | you-hub-name |

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 |
| Azure Security Center | 0.001 USD per device per month |

Tags

| | |
|------------|------------|
| department | accounting |
|------------|------------|

Create < Previous: Tags Next > Automation options

8. Select **Create** to create your new 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 command in Azure Cloud Shell to create the device identity.

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

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

```
az iot hub device-identity create --hub-name {YourIoTHubName} --device-id MyDotnetDevice
```

- Run the following command 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=MyDotnetDevice;SharedAccessKey={YourSharedAccessKey}
```

You'll use this value later in the quickstart.

- You also need the *Event Hubs-compatible endpoint*, *Event Hubs-compatible path*, and *service 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 service --query primaryKey --hub-name {YourIoTHubName}
```

Make a note of these three values, which you'll 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.

- 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.
- 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 earlier. Then save your changes to `SimulatedDevice.cs`.

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

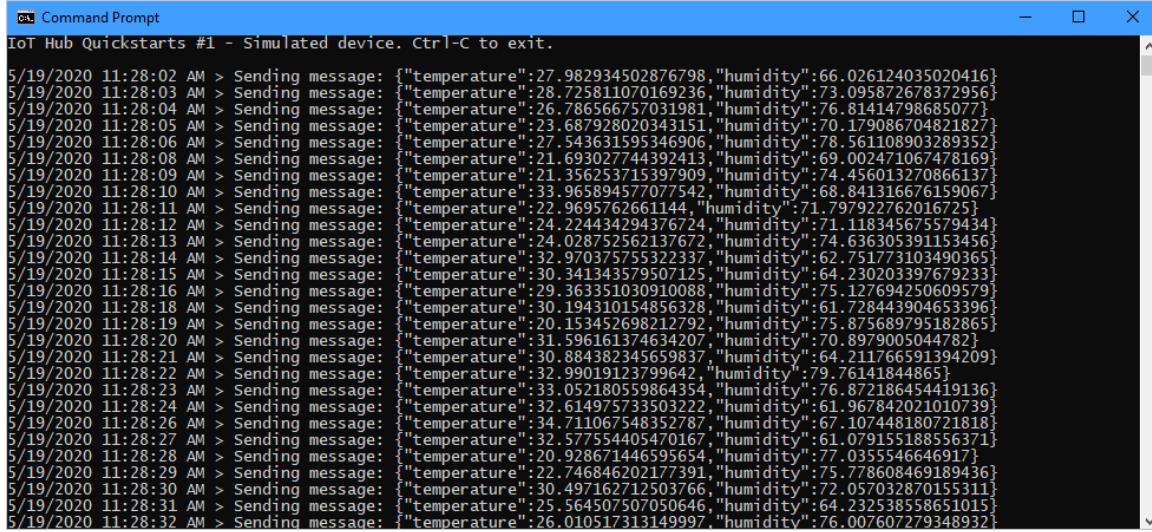
```
dotnet restore
```

- 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:



```
IoT Hub Quickstarts #1 - Simulated device. Ctrl-C to exit.

5/19/2020 11:28:02 AM > Sending message: {"temperature":27.982934502876798,"humidity":66.026124035020416}
5/19/2020 11:28:03 AM > Sending message: {"temperature":28.725811070169236,"humidity":73.095872678372956}
5/19/2020 11:28:04 AM > Sending message: {"temperature":26.786566757031981,"humidity":76.8141479868507}
5/19/2020 11:28:05 AM > Sending message: {"temperature":23.687928020343151,"humidity":70.179086704821827}
5/19/2020 11:28:06 AM > Sending message: {"temperature":27.543631595346906,"humidity":78.561108903289352}
5/19/2020 11:28:08 AM > Sending message: {"temperature":21.693027744392413,"humidity":69.002471067478169}
5/19/2020 11:28:09 AM > Sending message: {"temperature":21.356253715397909,"humidity":74.456013270866137}
5/19/2020 11:28:10 AM > Sending message: {"temperature":33.965894577077542,"humidity":68.841316676159067}
5/19/2020 11:28:11 AM > Sending message: {"temperature":22.9695762661144,"humidity":71.797922762016725}
5/19/2020 11:28:12 AM > Sending message: {"temperature":24.224434294376724,"humidity":71.118345675579434}
5/19/2020 11:28:13 AM > Sending message: {"temperature":24.028752562137672,"humidity":74.636305391153456}
5/19/2020 11:28:14 AM > Sending message: {"temperature":32.970375755322337,"humidity":62.751773103490365}
5/19/2020 11:28:15 AM > Sending message: {"temperature":30.341343759507125,"humidity":64.230203397679233}
5/19/2020 11:28:16 AM > Sending message: {"temperature":29.363351030910088,"humidity":75.127694250609579}
5/19/2020 11:28:18 AM > Sending message: {"temperature":30.194310154856328,"humidity":61.72843904653396}
5/19/2020 11:28:19 AM > Sending message: {"temperature":20.153452698212792,"humidity":75.875689795182865}
5/19/2020 11:28:20 AM > Sending message: {"temperature":31.596161374634207,"humidity":70.8979005044782}
5/19/2020 11:28:21 AM > Sending message: {"temperature":30.884382345659837,"humidity":64.211766591394209}
5/19/2020 11:28:22 AM > Sending message: {"temperature":32.99019123799642,"humidity":79.76141844865}
5/19/2020 11:28:23 AM > Sending message: {"temperature":33.052180559864354,"humidity":76.872186454419136}
5/19/2020 11:28:24 AM > Sending message: {"temperature":32.614975733503222,"humidity":61.967842021010739}
5/19/2020 11:28:26 AM > Sending message: {"temperature":34.711067548352787,"humidity":67.107448180721818}
5/19/2020 11:28:27 AM > Sending message: {"temperature":32.577554405470167,"humidity":61.079155188556371}
5/19/2020 11:28:28 AM > Sending message: {"temperature":20.928671446595654,"humidity":77.035554646917}
5/19/2020 11:28:29 AM > Sending message: {"temperature":22.746846202177391,"humidity":75.778608469189436}
5/19/2020 11:28:30 AM > Sending message: {"temperature":30.497162712503766,"humidity":72.057032870155311}
5/19/2020 11:28:31 AM > Sending message: {"temperature":25.564507507050646,"humidity":64.232538558651015}
5/19/2020 11:28:32 AM > Sending message: {"temperature":26.010517313149997,"humidity":76.007607279348932}
```

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 |
|--|---|
| <code>EventHubsCompatibleEndpoint</code> | Replace the value of the variable with the Event Hubs-compatible endpoint you made a note of earlier. |
| <code>EventHubName</code> | Replace the value of the variable with the Event Hubs-compatible path you made a note of earlier. |
| <code>IotHubSasKey</code> | Replace the value of the variable with the service primary key you made a note of earlier. |

NOTE

If you're using .NET Core SDK 2.1, you must set the language version to preview to compile the code. To do this, open the `read-d2c-messages.csproj` file and set the value of the `<LangVersion>` element to `preview`.

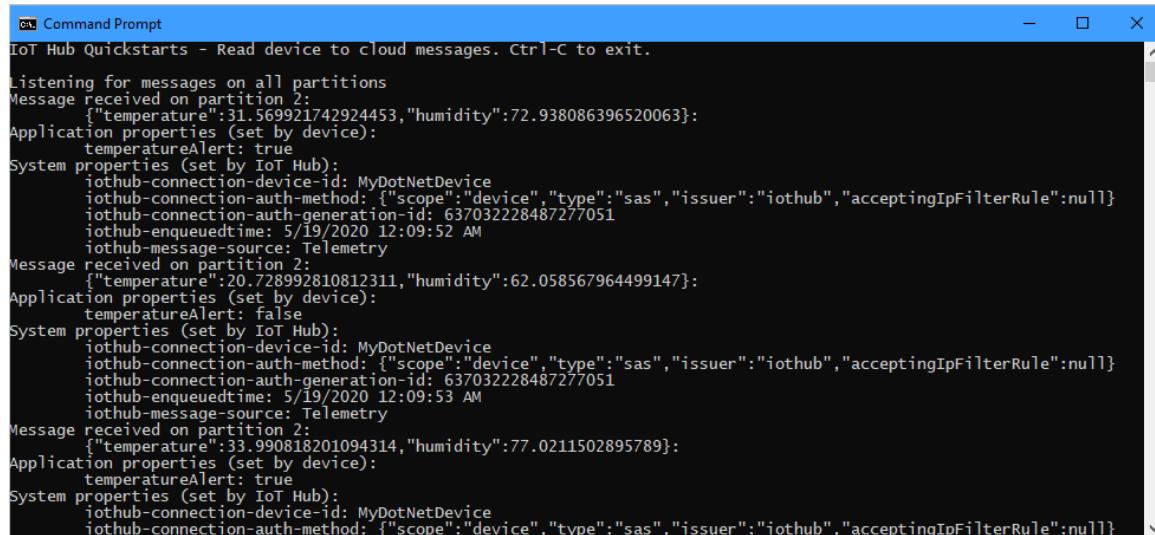
3. In the local terminal window, run the following commands to install the required libraries for the back-end application:

```
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:



```
IoT Hub Quickstarts - Read device to cloud messages. Ctrl-C to exit.

Listening for messages on all partitions
Message received on partition 2:
  {"temperature":31.569921742924453,"humidity":72.938086396520063}:
Application properties (set by device):
  temperatureAlert: true
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: 637032228487277051
  iothub-enqueuedtime: 5/19/2020 12:09:52 AM
  iothub-message-source: Telemetry
Message received on partition 2:
  {"temperature":20.728992810812311,"humidity":62.058567964499147}:
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: 637032228487277051
  iothub-enqueuedtime: 5/19/2020 12:09:53 AM
  iothub-message-source: Telemetry
Message received on partition 2:
  {"temperature":33.990818201094314,"humidity":77.0211502895789}:
Application properties (set by device):
  temperatureAlert: true
System properties (set by IoT Hub):
  iothub-connection-device-id: MyDotNetDevice
  iothub-connection-auth-method: {"scope":"device","type":"sas","issuer":"iothub","acceptingIpFilterRule":null}
```

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 select **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, select ... 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), 'Dashboard', 'All resources', and 'Recent'. The main content area is titled 'Resource groups' under 'Microsoft'. It displays a table with one item selected: 'TestResources' (Subscription: Prototype3, Location: West US). A red box highlights the search bar at the top containing 'TestResources'. Another red box highlights the 'Delete resource group' button in the table's header row. The table has columns for NAME, SUBSCRIPTION, and LOCATION.

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 select **Delete**. After a few moments, the resource group and all of its contained resources are deleted.

Next steps

In this quickstart, you 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 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 to an Azure IoT hub and read it with a Java application

7/29/2020 • 9 minutes to read • [Edit Online](#)

In this quickstart, you send telemetry to Azure IoT Hub and read it with a Java application. 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. This 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.

Prerequisites

- An Azure account with an active subscription. [Create one for free](#).
- Java SE Development Kit 8. In [Java long-term support for Azure and Azure Stack](#), under **Long-term support**, select **Java 8**.
- [Apache Maven 3](#).
- [A sample Java project](#).
- Port 8883 open in your firewall. The device sample in this quickstart uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

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

```
java -version
```

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

```
mvn --version
```

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|---|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |

| OPTION | EXAMPLE/LINK |
|---|---|
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  Launch Cloud Shell |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal . |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

Add Azure IoT Extension

Run the following command to add the Microsoft Azure IoT Extension for Azure CLI to your Cloud Shell instance. The IoT Extension adds IoT Hub, IoT Edge, and IoT Device Provisioning Service (DPS) specific commands to Azure CLI.

```
az extension add --name azure-iot
```

NOTE

This article uses the newest version of the Azure IoT extension, called `azure-iot`. The legacy version is called `azure-cli-iot-ext`. You should only have one version installed at a time. You can use the command `az extension list` to validate the currently installed extensions.

Use `az extension remove --name azure-cli-iot-ext` to remove the legacy version of the extension.

Use `az extension add --name azure-iot` to add the new version of the extension.

To see what extensions you have installed, use `az extension list`.

Create an IoT hub

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

1. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).

- **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.
- **IoT Hub Name:** Enter a name for your hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

Home > New > IoT hub

IoT hub

Microsoft X

[Basics](#) [Size and scale](#) [Tags](#) [Review + create](#)

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

Project details

Choose the subscription you'll use to manage deployments and costs. Use resource groups like folders to help you organize and manage resources.

Subscription * ⓘ ▼

Resource group * ⓘ ▼
[Create new](#)

Region * ⓘ ▼

IoT hub name * ⓘ

Review + create < Previous Next: Size and scale > [Automation options](#)

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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
Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center On
Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.
If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure resources](#).

Home > New > IoT hub

IoT hub

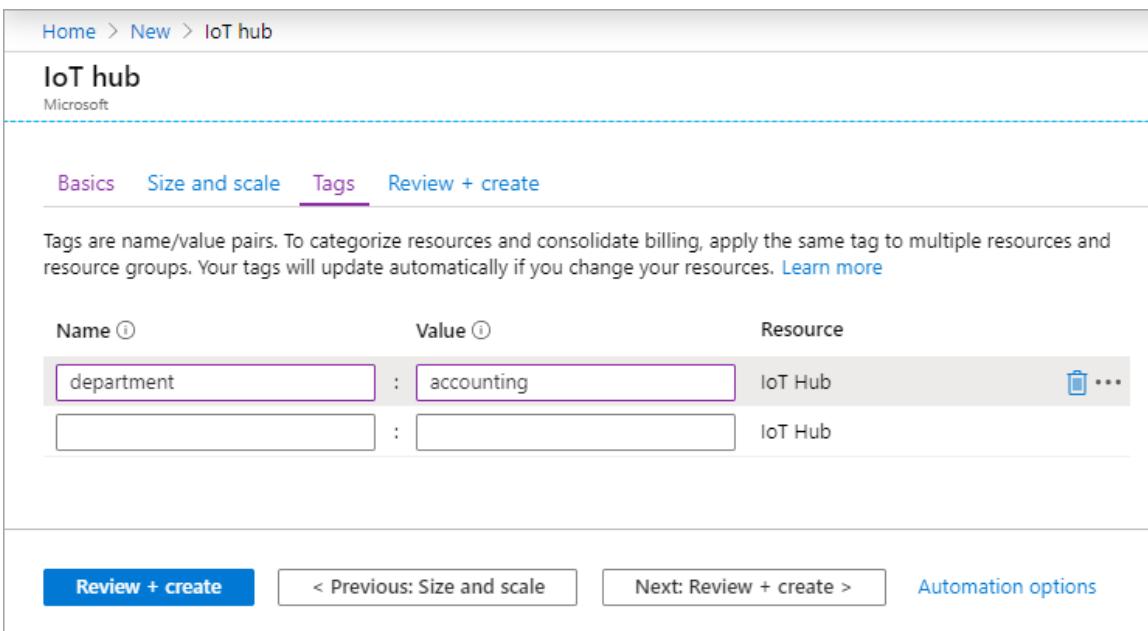
Microsoft

Basics Size and scale Tags **Review + create**

Tags are name/value pairs. To categorize resources and consolidate billing, apply the same tag to multiple resources and resource groups. Your tags will update automatically if you change your resources. [Learn more](#)

| Name ⓘ | Value ⓘ | Resource | ⋮ |
|------------|--------------|----------|-----|
| department | : accounting | IoT Hub | ... |
| | : | IoT Hub | |

Review + create < Previous: Size and scale Next: Review + create > Automation options



7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics **Size and scale** Tags **Review + create**

Basics

| | |
|----------------|------------------|
| Subscription | Personal testing |
| Resource group | iot-hubs |
| Region | West US 2 |
| IoT hub name | you-hub-name |

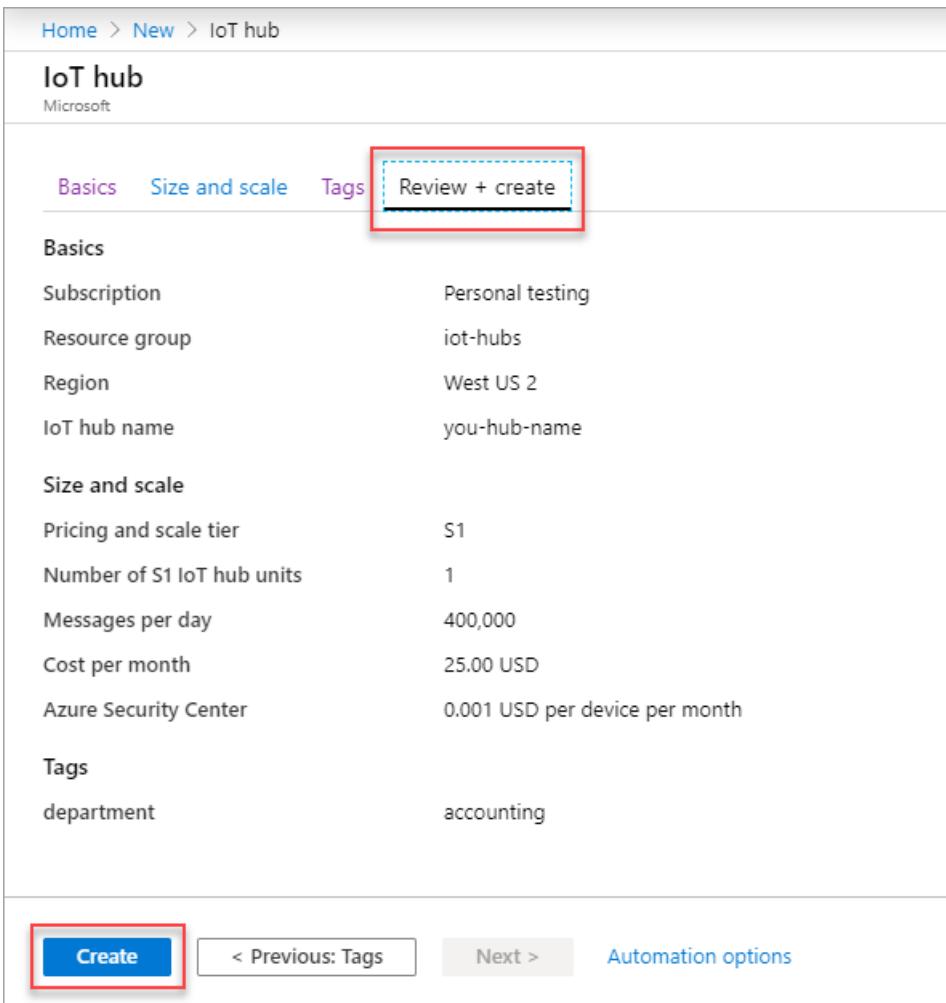
Size and scale

| | |
|----------------------------|--------------------------------|
| Pricing and scale tier | S1 |
| Number of S1 IoT hub units | 1 |
| Messages per day | 400,000 |
| Cost per month | 25.00 USD |
| Azure Security Center | 0.001 USD per device per month |

Tags

| | |
|------------|------------|
| department | accounting |
|------------|------------|

Create < Previous: Tags Next > Automation options



8. Select **Create** to create your new 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 command in Azure Cloud Shell to create the device identity.

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

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

```
az iot hub device-identity create --hub-name {YourIoTHubName} --device-id MyJavaDevice
```

- Run the following command 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 MyJavaDevice --output table
```

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

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

You'll use this value later in the quickstart.

- You also need the *Event Hubs-compatible endpoint*, *Event Hubs-compatible path*, and *service 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 chose 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 service --query primaryKey --hub-name {YourIoTHubName}
```

Make a note of these three values, which you'll 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.

- 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.
- 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 earlier. Then save your changes to `SimulatedDevice.java`.

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

```
mvn clean package
```

- 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:

```
C:\code2\azure-iot-samples-java\iot-hub\Quickstarts\simulated-device>java -jar target/simulated-device-1.0.0-with-deps.jar
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Press ENTER to exit.
Sending message: {"temperature":32.047153757856954,"humidity":73.80569612675366}
IoT Hub responded to message with status: OK_EMPTY
Sending message: {"temperature":26.887717061936726,"humidity":63.4490677572944}
IoT Hub responded to message with status: OK_EMPTY
Sending message: {"temperature":29.639075249678264,"humidity":65.84568508047677}
IoT Hub responded to message with status: OK_EMPTY
Sending message: {"temperature":24.35352986306092,"humidity":67.93220445265716}
IoT Hub responded to message with status: OK_EMPTY
Sending message: {"temperature":31.212861653069837,"humidity":66.14790168044107}
IoT Hub responded to message with status: OK_EMPTY
Sending message: {"temperature":34.5791227222709,"humidity":61.085442658193806}
IoT Hub responded to message with status: OK_EMPTY
Sending message: {"temperature":32.90613349520669,"humidity":63.919959262821564}
IoT Hub responded to message with status: OK_EMPTY
Sending message: {"temperature":23.012251730986236,"humidity":77.92823808073862}
IoT Hub responded to message with status: OK_EMPTY
Sending message: {"temperature":26.953280152633035,"humidity":78.38000666046342}
IoT Hub responded to message with status: OK_EMPTY
Sending message: {"temperature":32.59451394934783,"humidity":68.08788559533166}
IoT Hub responded to message with status: OK_EMPTY
Sending message: {"temperature":30.673219351100208,"humidity":75.67987886907717}
IoT Hub responded to message with status: OK_EMPTY
Sending message: {"temperature":23.511026901106764,"humidity":73.7590567425582}
```

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.

- 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.
- 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>EVENT_HUBS_COMPATIBLE_ENDPOINT</code> | Replace the value of the variable with the Event Hubs-compatible endpoint you made a note of earlier. |
| <code>EVENT_HUBS_COMPATIBLE_PATH</code> | Replace the value of the variable with the Event Hubs-compatible path you made a note of earlier. |
| <code>IOT_HUB_SAS_KEY</code> | Replace the value of the variable with the service primary key you made a note of earlier. |

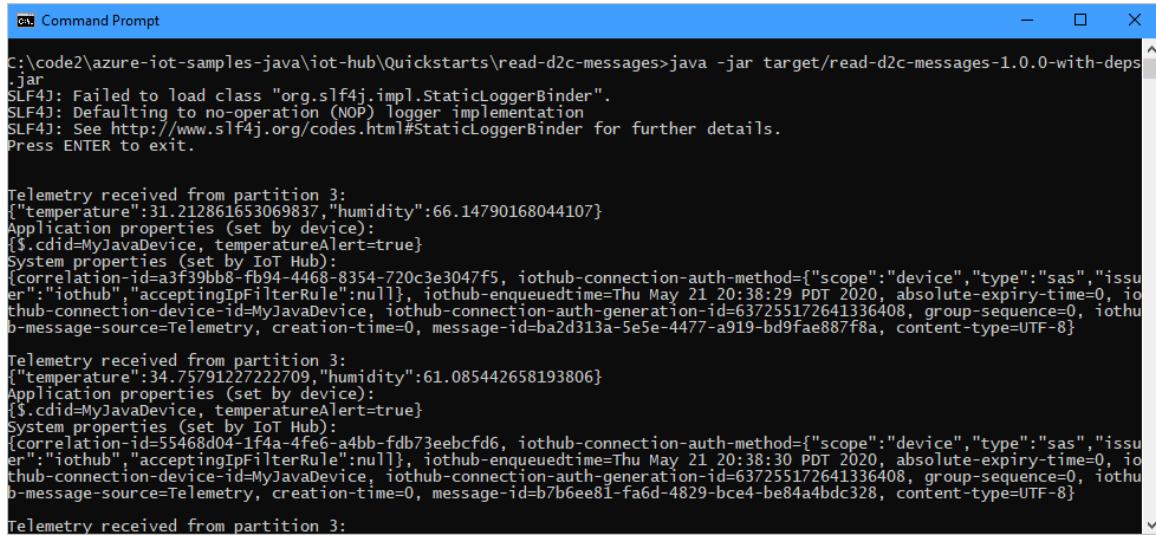
- In the local terminal window, run the following commands to install the required libraries and build the back-end application:

```
mvn clean package
```

- 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:



```
C:\code2\azure-iot-samples-java\iot-hub\Quickstarts\read-d2c-messages>java -jar target/read-d2c-messages-1.0.0-with-deps.jar
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) Logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Press ENTER to exit.

Telemetry received from partition 3:
{"temperature":31.212861653069837,"humidity":66.14790168044107}
Application properties (set by device):
{$.cdid=MyJavaDevice, temperatureAlert=true}
System properties (set by IoT Hub):
{correlation-id=a3f39bb8-fb94-4468-8354-720c3e3047f5, iothub-connection-auth-method={"scope":"device","type":"sas","issuer":"iothub", "acceptingIpFilterRule":null}, iothub-enqueueetime=Thu May 21 20:38:29 PDT 2020, absolute-expiry-time=0, iothub-connection-device-id=MyJavaDevice, iothub-connection-auth-generation-id=637255172641336408, group-sequence=0, iothub-message-source=Telemetry, creation-time=0, message-id=ba2d313a-5e5e-4477-a919-bd9fae887f8a, content-type=UTF-8}

Telemetry received from partition 3:
{"temperature":34.7579122722709,"humidity":61.085442658193806}
Application properties (set by device):
{$.cdid=MyJavaDevice, temperatureAlert=true}
System properties (set by IoT Hub):
{correlation-id=55468d04-1f4a-4fe6-a4bb-fdb73eebcfd6, iothub-connection-auth-method={"scope":"device","type":"sas","issuer":"iothub", "acceptingIpFilterRule":null}, iothub-enqueueetime=Thu May 21 20:38:30 PDT 2020, absolute-expiry-time=0, iothub-connection-device-id=MyJavaDevice, iothub-connection-auth-generation-id=637255172641336408, group-sequence=0, iothub-message-source=Telemetry, creation-time=0, message-id=b7b6ee81-fa6d-4829-bce4-be84a4bdc328, content-type=UTF-8}

Telemetry received from partition 3:
```

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.

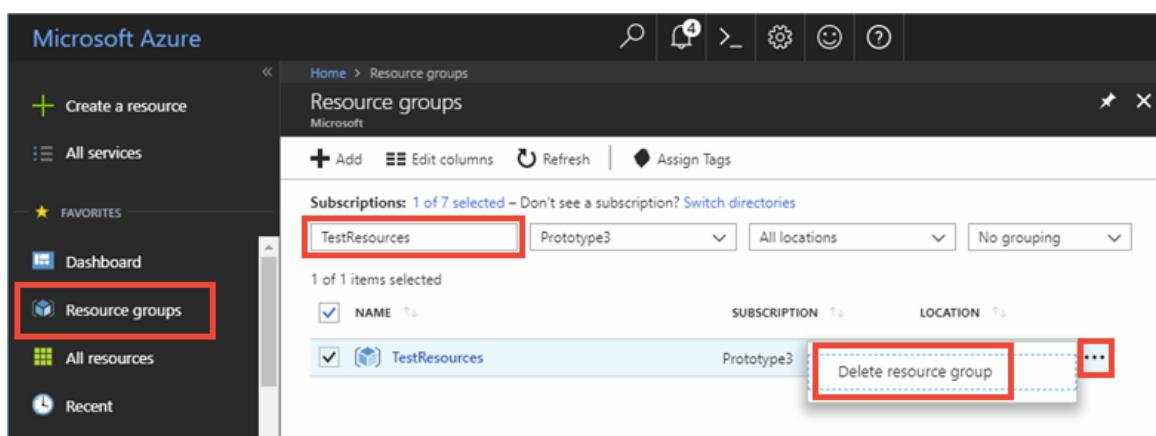
Otherwise, you can delete the Azure resources created in this article to avoid charges.

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 select **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, select ... then **Delete resource group**.



Microsoft Azure

Home > Resource groups

Resource groups

Subscriptions: 1 of 7 selected - Don't see a subscription? Switch directories

| NAME | SUBSCRIPTION | LOCATION |
|---------------|--------------|----------|
| TestResources | Prototype3 | |

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 select **Delete**. After a few moments, the resource group and all of its contained resources are deleted.

Next steps

In this quickstart, you set up 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)

7/29/2020 • 9 minutes to read • [Edit Online](#)

In this quickstart, you send telemetry from a simulated device application through Azure IoT Hub to a back-end application for processing. 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. This quickstart uses two pre-written Python 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.

Prerequisites

- An Azure account with an active subscription. [Create one for free](#).
- [Python 3.7+](#). For other versions of Python supported, see [Azure IoT Device Features](#).
- [A sample Python project](#).
- Port 8883 open in your firewall. The device sample in this quickstart uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|--|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal. |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the Copy button on a code block to copy the code.

3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.

4. Select **Enter** to run the code.

Add Azure IoT Extension

Run the following command to add the Microsoft Azure IoT Extension for Azure CLI to your Cloud Shell instance. The IoT Extension adds IoT Hub, IoT Edge, and IoT Device Provisioning Service (DPS) specific commands to Azure CLI.

```
az extension add --name azure-iot
```

NOTE

This article uses the newest version of the Azure IoT extension, called `azure-iot`. The legacy version is called `azure-cli-iot-ext`. You should only have one version installed at a time. You can use the command `az extension list` to validate the currently installed extensions.

Use `az extension remove --name azure-cli-iot-ext` to remove the legacy version of the extension.

Use `az extension add --name azure-iot` to add the new version of the extension.

To see what extensions you have installed, use `az extension list`.

Create an IoT hub

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

1. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.
 - **IoT Hub Name:** Enter a name for your hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

Home > New > IoT hub

IoT hub

Microsoft

[X](#)

[Basics](#) [Size and scale](#) [Tags](#) [Review + create](#)

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

Project details

Choose the subscription you'll use to manage deployments and costs. Use resource groups like folders to help you organize and manage resources.

Subscription * ⓘ Personal IoT items

Resource group * ⓘ Create new

Region * ⓘ East Asia

IoT hub name * ⓘ Once your hub is created, this name can't be changed

[Review + create](#) [< Previous](#) [Next: Size and scale >](#) [Automation options](#)

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

[Basics](#) [Size and scale](#) [Tags](#) [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

Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center On

Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

[Advanced settings](#)

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.
If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure resources](#).

The screenshot shows the 'Tags' tab of the IoT hub configuration interface. It displays a table of tags with columns for Name, Value, and Resource. One tag is explicitly listed: 'department' with value 'accounting' is associated with an 'IoT Hub'. There is also an empty row for another tag entry. Navigation buttons at the bottom include 'Review + create', '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

| Name ⓘ | Value ⓘ | Resource |
|------------|--------------|----------|
| department | : accounting | IoT Hub |
| | : | IoT Hub |

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' step of the Azure IoT hub creation wizard. At the top, there are tabs for 'Basics', 'Size and scale', 'Tags', and 'Review + create'. The 'Review + create' tab is highlighted with a red dashed border. Below the tabs, there are two sections: 'Basics' and 'Size and scale'. Under 'Basics', the subscription is set to 'Personal testing', the resource group is 'iot-hubs', the region is 'West US 2', and the IoT hub name is 'you-hub-name'. Under 'Size and scale', the pricing tier is 'S1', there is 1 S1 IoT hub unit, 400,000 messages per day, and a monthly cost of 25.00 USD. An Azure Security Center integration is also listed. Under 'Tags', there is a single tag 'department' with the value 'accounting'. At the bottom of the screen, there are three buttons: a blue 'Create' button with a red border, a light gray '< Previous: Tags' button, and a light gray 'Next >' button. To the right of the 'Next >' button is a link to 'Automation options'.

8. Select **Create** to create your new 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 command in Azure Cloud Shell to create the device identity.

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

MyPythonDevice: This is the name of the device you're registering. It's recommended to use **MyPythonDevice** as shown. If you choose a different name for your device, you'll also need to use that name throughout this article, and update the device name in the sample applications before you run them.

```
az iot hub device-identity create --hub-name {YourIoTHubName} --device-id MyPythonDevice
```

2. Run the following command in Azure Cloud Shell to get the *device connection string* for the device you 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=MyPythonDevice;SharedAccessKey={YourSharedAccessKey}
```

You'll use this value later in the quickstart.

3. You also need the *Event Hubs-compatible endpoint*, *Event Hubs-compatible path*, and *service 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 service --query primaryKey --hub-name {YourIoTHubName}
```

Make a note of these three values, which you'll 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 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 earlier. Then save your changes to `SimulatedDevice.py`.

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

```
pip install azure-iot-device
```

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:

```
IoT Hub Quicksstart #1 - Simulated device
Press Ctrl-C to exit
IoT Hub device sending periodic messages, press Ctrl-C to exit
Sending message: {"temperature": 26.053027015583066,"humidity": 71.56810485306812}
Message successfully sent
Sending message: {"temperature": 25.069543486353563,"humidity": 76.7297746420288}
Message successfully sent
Sending message: {"temperature": 34.41176064073353,"humidity": 61.982043366567815}
Message successfully sent
Sending message: {"temperature": 27.094543621373326,"humidity": 71.02240176418414}
Message successfully sent
Sending message: {"temperature": 33.37008074339154,"humidity": 72.91640031652045}
Message successfully sent
Sending message: {"temperature": 20.131907912756805,"humidity": 61.623356128318555}
Message successfully sent
Sending message: {"temperature": 32.48780379653941,"humidity": 74.17991781031843}
Message successfully sent
Sending message: {"temperature": 26.571487582583135,"humidity": 77.91151664594464}
Message successfully sent
Sending message: {"temperature": 24.499563410882935,"humidity": 73.18689979905011}
Message successfully sent
Sending message: {"temperature": 23.400458361451047,"humidity": 63.13934746260987}
Message successfully sent
Sending message: {"temperature": 24.649674098310825,"humidity": 68.32124008081483}
Message successfully sent
Sending message: {"temperature": 29.687659307001468,"humidity": 63.81772049482646}
Message successfully sent
Sending message: {"temperature": 31.00215871762193,"humidity": 77.8735894768308}
Message successfully sent
Sending message: {"temperature": 30.35144196552421,"humidity": 74.57549179464313}
```

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.

NOTE

The following steps use the synchronous sample, `read_device_to_cloud_messages_sync.py`. You can perform the same steps with the asynchronous sample, `read_device_to_cloud_messages_async.py`.

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

| VARIABLE | VALUE |
|---|---|
| <code>EVENTHUB_COMPATIBLE_ENDPOINT</code> | Replace the value of the variable with the Event Hubs-compatible endpoint you made a note of earlier. |
| <code>EVENTHUB_COMPATIBLE_PATH</code> | Replace the value of the variable with the Event Hubs-compatible path you made a note of earlier. |
| <code>IOTHUB_SAS_KEY</code> | Replace the value of the variable with the service primary key you made a note of earlier. |

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

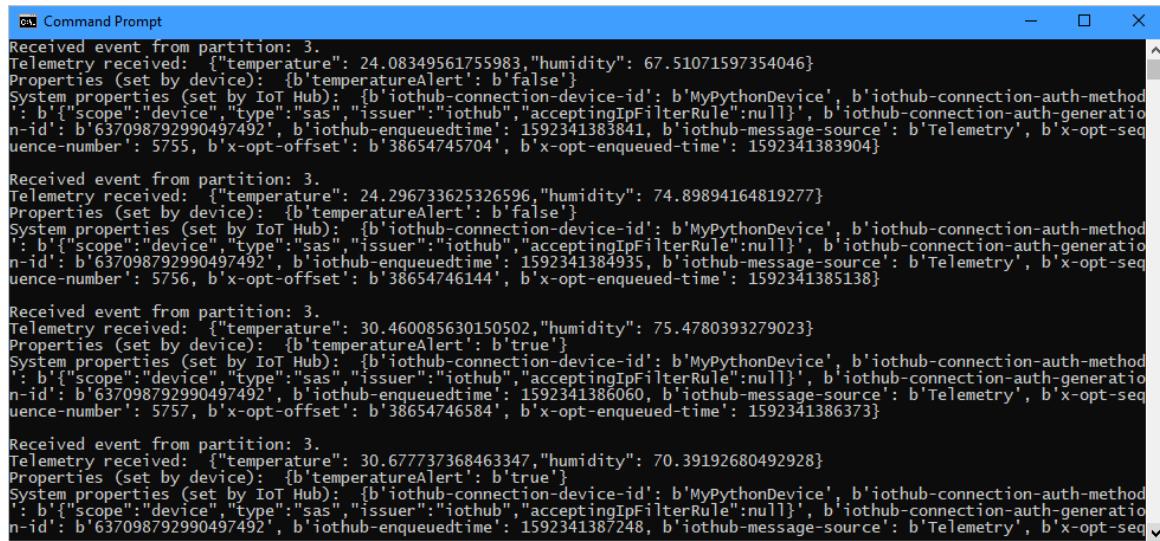
```
pip install azure-eventhub
```

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

```
python read_device_to_cloud_messages_sync.py
```

The following screenshot shows the output as the back-end application receives telemetry sent by the

simulated device to the hub:



```
Command Prompt
Received event from partition: 3.
Telemetry received: {"temperature": 24.08349561755983,"humidity": 67.51071597354046}
Properties (set by device): {b'temperatureAlert': b'false'}
System properties (set by IoT Hub): {[b'iothub-connection-device-id': b'MyPythonDevice', b'iothub-connection-auth-method': b'{"scope":"device","type":"sas","issuer":"iothub","acceptingIpFilterRule":null}', b'iothub-connection-auth-generation-id': b'637098792990497492', b'iothub-enqueuedtime': 1592341383841, b'iothub-message-source': b'Telemetry', b'x-opt-sequence-number': 5755, b'x-opt-offset': b'38654745704', b'x-opt-enqueued-time': 1592341383904}

Received event from partition: 3.
Telemetry received: {"temperature": 24.296733625326596,"humidity": 74.89894164819277}
Properties (set by device): {b'temperatureAlert': b'false'}
System properties (set by IoT Hub): {[b'iothub-connection-device-id': b'MyPythonDevice', b'iothub-connection-auth-method': b'{"scope":"device","type":"sas","issuer":"iothub","acceptingIpFilterRule":null}', b'iothub-connection-auth-generation-id': b'637098792990497492', b'iothub-enqueuedtime': 1592341384935, b'iothub-message-source': b'Telemetry', b'x-opt-sequence-number': 5756, b'x-opt-offset': b'38654746144', b'x-opt-enqueued-time': 1592341385138}

Received event from partition: 3.
Telemetry received: {"temperature": 30.460085630150502,"humidity": 75.4780393279023}
Properties (set by device): {b'temperatureAlert': b'true'}
System properties (set by IoT Hub): {[b'iothub-connection-device-id': b'MyPythonDevice', b'iothub-connection-auth-method': b'{"scope":"device","type":"sas","issuer":"iothub","acceptingIpFilterRule":null}', b'iothub-connection-auth-generation-id': b'637098792990497492', b'iothub-enqueuedtime': 1592341386060, b'iothub-message-source': b'Telemetry', b'x-opt-sequence-number': 5757, b'x-opt-offset': b'38654746584', b'x-opt-enqueued-time': 1592341386373}

Received event from partition: 3.
Telemetry received: {"temperature": 30.677737368463347,"humidity": 70.39192680492928}
Properties (set by device): {b'temperatureAlert': b'true'}
System properties (set by IoT Hub): {[b'iothub-connection-device-id': b'MyPythonDevice', b'iothub-connection-auth-method': b'{"scope":"device","type":"sas","issuer":"iothub","acceptingIpFilterRule":null}', b'iothub-connection-auth-generation-id': b'637098792990497492', b'iothub-enqueuedtime': 1592341387248, b'iothub-message-source': b'Telemetry', b'x-opt-sequence-number': 5758, b'x-opt-offset': b'38654746748', b'x-opt-enqueued-time': 1592341387373}
```

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.

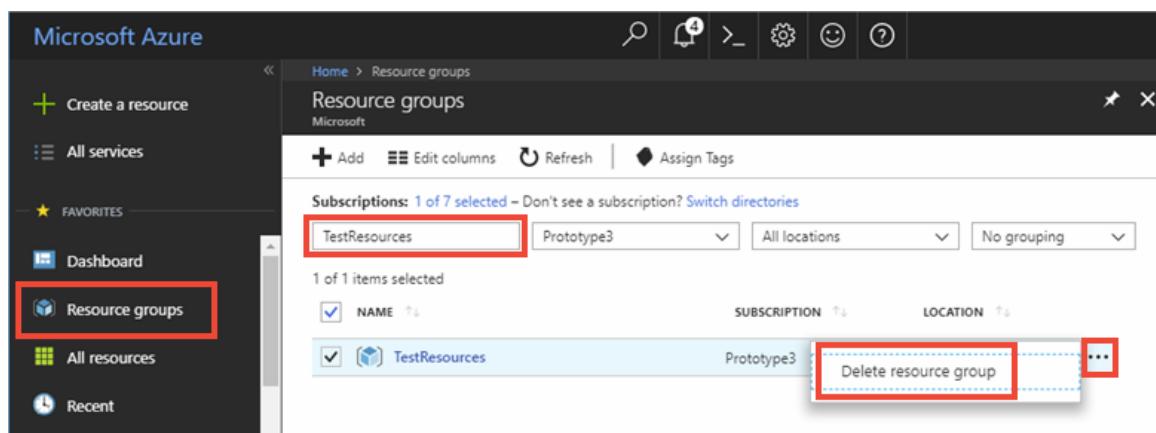
Otherwise, you can delete the Azure resources created in this article to avoid charges.

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 select **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, select ... 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: 'TestResources'. The 'Subscriptions' dropdown shows '1 of 7 selected' and 'TestResources'. The table has columns for NAME, SUBSCRIPTION, and LOCATION. The 'TestResources' row has a checkbox checked and a 'Delete resource group' button with three dots (...). A red box highlights the 'Delete resource group' 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 select **Delete**. After a few moments, the resource group and all of its contained resources are deleted.

Next steps

In this quickstart, you set up 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

7/29/2020 • 8 minutes to read • [Edit Online](#)

In this quickstart, you send telemetry to an Azure IoT Hub from an Android application running on a physical or simulated device. 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. This 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.

Prerequisites

- An Azure account with an active subscription. [Create one for free](#).
- [Android Studio with Android SDK 27](#). For more information, see [android-installation](#). Android SDK 27 is used by the sample in this article.
- [A sample Android application](#). This sample is part of the [azure-iot-samples-java](#) repository.
- Port 8883 open in your firewall. The device sample in this quickstart uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|--|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal . |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the Copy button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by

selecting Cmd+Shift+V on macOS.

4. Select **Enter** to run the code.

Add Azure IoT Extension

Run the following command to add the Microsoft Azure IoT Extension for Azure CLI to your Cloud Shell instance. The IOT Extension adds IoT Hub, IoT Edge, and IoT Device Provisioning Service (DPS) specific commands to Azure CLI.

```
az extension add --name azure-iot
```

NOTE

This article uses the newest version of the Azure IoT extension, called `azure-iot`. The legacy version is called `azure-cli-iot-ext`. You should only have one version installed at a time. You can use the command `az extension list` to validate the currently installed extensions.

Use `az extension remove --name azure-cli-iot-ext` to remove the legacy version of the extension.

Use `az extension add --name azure-iot` to add the new version of the extension.

To see what extensions you have installed, use `az extension list`.

Create an IoT hub

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

1. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.
 - **IoT Hub Name:** Enter a name for your hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags Review + create

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

Project details

Choose the subscription you'll use to manage deployments and costs. Use resource groups like folders to help you organize and manage resources.

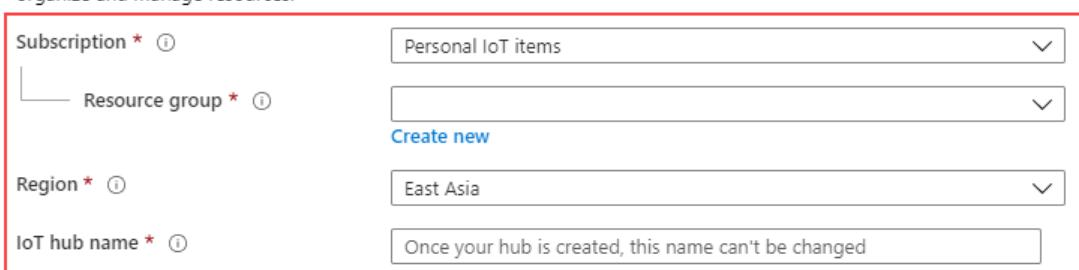
Subscription * ⓘ Personal IoT items

Resource group * ⓘ [Create new](#)

Region * ⓘ East Asia

IoT hub name * ⓘ Once your hub is created, this name can't be changed

Review + create < Previous **Next: Size and scale >** Automation options



5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics **Size and scale** Tags 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 ⓘ Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center On Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

Review + create < Previous: Basics **Next: Tags >** Automation options

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.
If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure resources](#).

The screenshot shows the 'Tags' configuration page for an IoT hub. At the top, there are tabs for 'Basics', 'Size and scale', 'Tags' (which is underlined, indicating it is active), and 'Review + create'. Below the tabs, a note states: 'Tags are name/value pairs. To categorize resources and consolidate billing, apply the same tag to multiple resources and resource groups. Your tags will update automatically if you change your resources.' A 'Learn more' link is provided. A table lists the current tags: one tag named 'department' with the value 'accounting' is associated with an 'IoT Hub' resource. There is also an empty row for another tag entry. At the bottom, there are buttons for 'Review + create', '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' step of the Azure IoT hub creation wizard. The 'Review + create' button is highlighted with a red box. The page displays configuration details under three sections: Basics, Size and scale, and Tags. In the Basics section, the IoT hub name is set to 'you-hub-name'. In the Size and scale section, the Pricing and scale tier is S1, and the Number of S1 IoT hub units is 1. In the Tags section, there is one tag named 'department' with the value 'accounting'. At the bottom, there are buttons for 'Create' (highlighted with a red box), '< Previous: Tags', 'Next >', and 'Automation options'.

8. Select **Create** to create your new 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 command in Azure Cloud Shell to create the device identity.

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

MyAndroidDevice: This is the name of the device you're registering. It's recommended to use **MyAndroidDevice** as shown. If you choose a different name for your device, you'll also need to use that name throughout this article, and update the device name in the sample applications before you run them.

```
az iot hub device-identity create --hub-name {YourIoTHubName} --device-id MyAndroidDevice
```

2. Run the following command 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 MyAndroidDevice --output table
```

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

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

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

Send simulated 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 the device connection string you made a note of earlier.

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

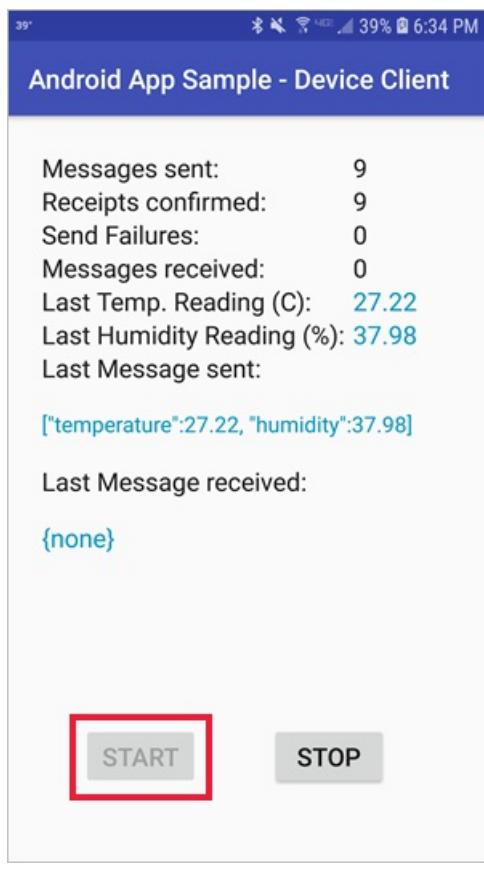
3. In Android Studio, click **File > Sync Project with Gradle Files**. Verify the build completes.

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.

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 chose 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 select **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, select ... then **Delete resource group**.

The screenshot shows the Microsoft Azure portal interface. On the left, the sidebar has 'Resource groups' selected. The main area is titled 'Resource groups' and shows a table with one item: 'TestResources'. The 'NAME' column shows 'TestResources', the 'SUBSCRIPTION' column shows 'Prototype3', and the 'LOCATION' column shows 'All locations'. To the right of the 'NAME' column for 'TestResources', there is a 'Delete resource group' button, which is highlighted with a red box. The 'Resource groups' section also includes a 'Subscriptions' dropdown set to 'TestResources', and buttons for 'Add', 'Edit columns', 'Refresh', and 'Assign Tags'.

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 select **Delete**. After a few moments, the resource group and all of its contained resources are deleted.

Next steps

In this quickstart, you set up 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)

7/29/2020 • 9 minutes to read • [Edit Online](#)

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.

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|---|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal . |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

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 10.2 and iOS 12.2.

- The latest version of [CocoaPods](#).
- Make sure that port 8883 is open in your firewall. The device sample in this quickstart uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).
- Run the following command to add the Microsoft Azure IoT Extension for Azure CLI to your Cloud Shell instance. The IoT Extension adds IoT Hub, IoT Edge, and IoT Device Provisioning Service (DPS) specific commands to Azure CLI.

```
az extension add --name azure-iot
```

NOTE

This article uses the newest version of the Azure IoT extension, called `azure-iot`. The legacy version is called `azure-cli-iot-ext`. You should only have one version installed at a time. You can use the command `az extension list` to validate the currently installed extensions.

Use `az extension remove --name azure-cli-iot-ext` to remove the legacy version of the extension.

Use `az extension add --name azure-iot` to add the new version of the extension.

To see what extensions you have installed, use `az extension list`.

Create an IoT hub

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

1. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.
 - **IoT Hub Name:** Enter a name for your hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags Review + create

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

Project details

Choose the subscription you'll use to manage deployments and costs. Use resource groups like folders to help you organize and manage resources.

Subscription * ⓘ Personal IoT items

Resource group * ⓘ Create new

Region * ⓘ East Asia

IoT hub name * ⓘ Once your hub is created, this name can't be changed

Review + create < Previous **Next: Size and scale >** Automation options

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics **Size and scale** Tags 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

Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center On

Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

Review + create < Previous: Basics **Next: Tags >** Automation options

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.
If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure resources](#).

The screenshot shows the 'Tags' tab of the 'IoT hub' configuration page. The 'Tags' section displays a table with two rows of name-value pairs. The first row has 'Name' as 'department' and 'Value' as 'accounting'. The second row has an empty 'Name' field and an empty 'Value' field. Below the table, there are navigation buttons: 'Review + create' (highlighted in blue), '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

| Name | Value | Resource |
|------------|------------|----------|
| department | accounting | IoT Hub |
| | | IoT Hub |

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' step of the Azure IoT hub creation wizard. The page is titled 'IoT hub' under the Microsoft category. At the top, there are tabs for 'Basics', 'Size and scale', 'Tags', and 'Review + create'. The 'Review + create' tab is highlighted with a red dashed border. Below the tabs, there are two sections: 'Basics' and 'Size and scale'. Under 'Basics', the following details are listed:

| | |
|----------------|------------------|
| Subscription | Personal testing |
| Resource group | iot-hubs |
| Region | West US 2 |
| IoT hub name | you-hub-name |

Under 'Size and scale', 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 |
| Azure Security Center | 0.001 USD per device per month |

Under 'Tags', the following detail is listed:

| | |
|------------|------------|
| department | accounting |
|------------|------------|

At the bottom of the page, there are four buttons: 'Create' (highlighted with a red box), '< Previous: Tags', 'Next >', and 'Automation options'.

8. Select **Create** to create your new 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 command in Azure Cloud Shell to create the device identity.

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

myiOSdevice: This is the name of the device you're registering. It's recommended to use **myiOSdevice** as shown. If you choose a different name for your device, you'll also need to use that name throughout this article, and update the device name in the sample applications before you run them.

```
az iot hub device-identity create --hub-name {YourIoTHubName} --device-id myiOSdevice
```

2. Run the following command 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 myiOSdevice --output table
```

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

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

You'll 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"
```

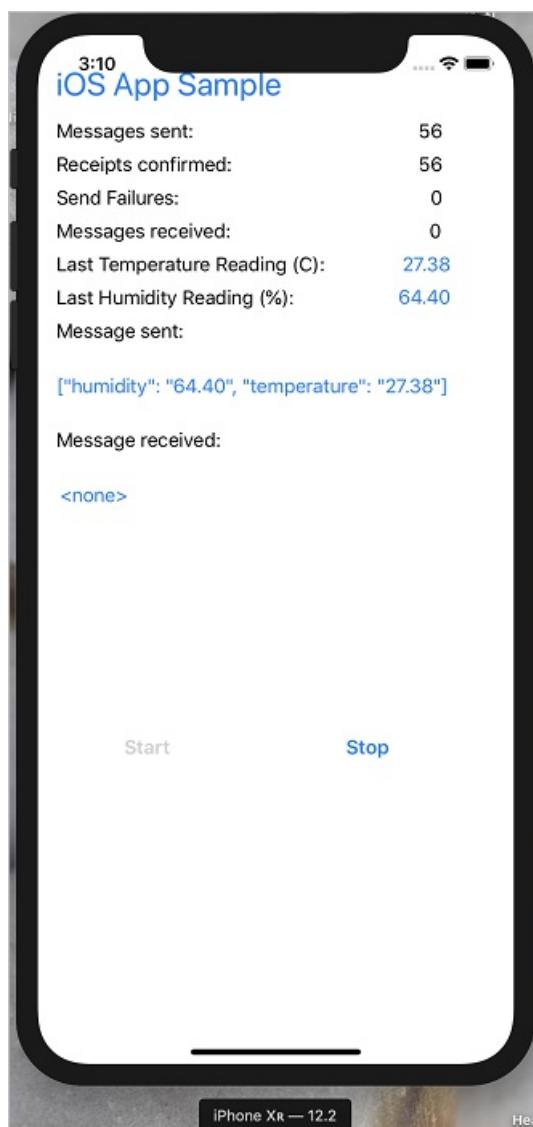
2. Expand the **MQTT Client Sample** project and then expand the folder of the same name.
3. Open **ViewController.swift** for editing in XCode.
4. Search for the **connectionString** variable and update the value with the device connection string that you made a note of earlier.
5. Save your changes.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

The screenshot shows the Xcode interface. The top menu bar includes Apple, Xcode, File, Edit, View, Find, Navigate, and Editor. Below the menu is a toolbar with five icons: red, grey, green, play (highlighted with a red box), and square. To the right of the toolbar is the project name "MQTT Client Sample" and the target "iPhone XR". The left sidebar shows the project structure: "MQTT Client Sample" folder containing "MQTT Client Sample" (a folder), "AppDelegate.swift", "ViewController.swift" (which is selected and highlighted in grey), "Main.storyboard", "Assets.xcassets", "LaunchScreen.storyboard", and "Info.plist". The main editor area displays Swift code:

```
1 // Copyright
2 // Licensed u
3
4 import UIKit
5 import AzureI
6 import Founda
7
8
9 class ViewCon
10
```

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

Secure | https://shell.azure.com

Azure Cloud Shell

(your account) (YOUR DIRECTORY)

```
Bash | ⚡ ? 🌐 🔍 { }

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 select **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, select ... 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' and 'Resource groups'), 'All resources', and 'Recent'. The 'Resource groups' option is highlighted with a red box. The main area is titled 'Resource groups' and shows a table with one item selected. The table has columns for 'NAME', 'SUBSCRIPTION', and 'LOCATION'. The single row contains 'TestResources', 'Prototype3', and 'East US'. To the right of the 'NAME' column, there's a checkmark icon. To the right of the 'SUBSCRIPTION' and 'LOCATION' columns, there are sorting arrows. At the bottom right of the row, there's a blue button labeled 'Delete resource group' with a red box around it, and a '...' button to its right.

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 select **Delete**. After a few moments, the resource group and all of its contained resources are deleted.

Next steps

In this quickstart, 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: Send telemetry from a device to an IoT hub (Xamarin.Forms)

7/29/2020 • 8 minutes to read • [Edit Online](#)

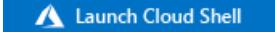
IoT Hub is an Azure service that enables you to ingest high volumes of telemetry from your IoT devices into the cloud for storage or processing. In this article, you send telemetry from a simulated device application to IoT Hub. Then you can view the data from a back-end application.

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

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|--|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal . |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

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 [Visual Studio 2019](#) or [Visual Studio for Mac](#) with Xamarin.Forms tooling installed.
This quickstart was tested with Visual Studio 16.6.0.

- Make sure that port 8883 is open in your firewall. The device sample in this quickstart uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).
- Run the following command to add the Microsoft Azure IoT Extension for Azure CLI to your Cloud Shell instance. The IoT Extension adds IoT Hub, IoT Edge, and IoT Device Provisioning Service (DPS) specific commands to Azure CLI.

```
az extension add --name azure-iot
```

NOTE

This article uses the newest version of the Azure IoT extension, called `azure-iot`. The legacy version is called `azure-cli-iot-ext`. You should only have one version installed at a time. You can use the command `az extension list` to validate the currently installed extensions.

Use `az extension remove --name azure-cli-iot-ext` to remove the legacy version of the extension.

Use `az extension add --name azure-iot` to add the new version of the extension.

To see what extensions you have installed, use `az extension list`.

Create an IoT hub

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

1. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.
 - **IoT Hub Name:** Enter a name for your hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

Home > New > IoT hub

IoT hub

Microsoft

[Basics](#) [Size and scale](#) [Tags](#) [Review + create](#)

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

Project details

Choose the subscription you'll use to manage deployments and costs. Use resource groups like folders to help you organize and manage resources.

Subscription * ⓘ

Resource group * ⓘ
[Create new](#)

Region * ⓘ

IoT hub name * ⓘ

[Review + create](#) [< Previous](#) [Next: Size and scale >](#) [Automation options](#)

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

[Basics](#) [Size and scale](#) [Tags](#) [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 * ⓘ
[Learn how to choose the right IoT hub tier for your solution](#)

Number of S1 IoT hub units ⓘ
Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center On
Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

[Advanced settings](#)

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.

If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure resources](#).

| Name ⓘ | Value ⓘ | Resource | ⋮ |
|------------|------------|----------|---|
| department | accounting | IoT Hub | |
| | | IoT Hub | |

Review + create < Previous: Size and scale Next: Review + create > Automation options

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' step of the Azure IoT hub creation wizard. At the top, there are tabs for 'Basics', 'Size and scale', 'Tags', and 'Review + create'. The 'Review + create' tab is highlighted with a red border. Below the tabs, there are two sections: 'Basics' and 'Size and scale'. Under 'Basics', the following details are listed:

| | |
|----------------|------------------|
| Subscription | Personal testing |
| Resource group | iot-hubs |
| Region | West US 2 |
| IoT hub name | you-hub-name |

Under 'Size and scale', 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 |
| Azure Security Center | 0.001 USD per device per month |

Under 'Tags', the following detail is listed:

| | |
|------------|------------|
| department | accounting |
|------------|------------|

At the bottom of the screen, there are several buttons: a large blue 'Create' button with a red border, a 'Previous: Tags' button, a 'Next >' button, and a 'Automation options' link.

8. Select **Create** to create your new 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 command in Azure Cloud Shell to create the device identity.

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

myXamarinDevice: This is the name of the device you're registering. It's recommended to use **myXamarinDevice** as shown. If you choose a different name for your device, you'll also need to use that name throughout this article, and update the device name in the sample applications before you run them.

```
az iot hub device-identity create --hub-name {YourIoTHubName} --device-id myXamarinDevice
```

2. Run the following command 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 myXamarinDevice --output table
```

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

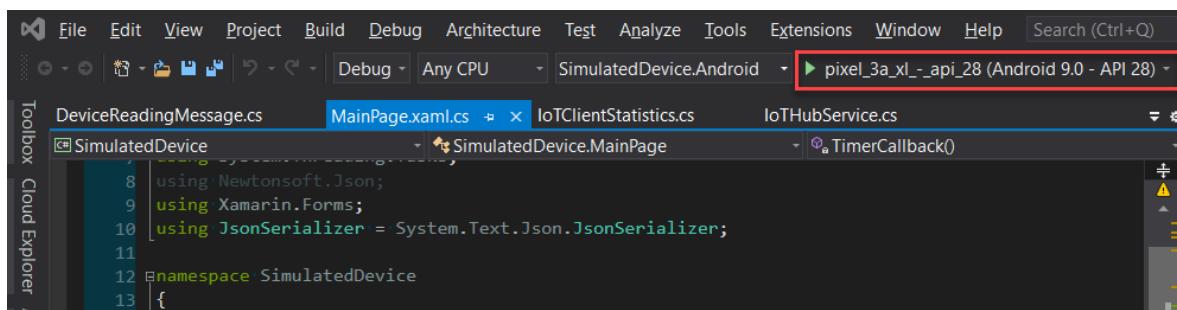
```
HostName={YourIoTHubName}.azure-devices.net;DeviceId=myXamarinDevice;SharedAccessKey={YourSharedAccessKey}
```

You'll use this value later in the quickstart.

Send simulated telemetry

The sample application runs on Windows - via an UWP app - an iOS device or simulator and on Android device or simulator, which connects to a device-specific endpoint on your IoT hub and sends simulated temperature and humidity telemetry.

1. Open the sample workspace in Visual Studio or Visual Studio for Mac.
2. Expand the **SimulatedDevice** project.
3. Open **IoTHubService.cs** for editing in Visual Studio.
4. Search for the **_iotHubConnectionString** variable and update the value with the device connection string that you made a note of earlier.
5. Save your changes.
6. Run the project in the device emulator or a real device with the **Build and run** button or the key shortcut **F5** on Windows or **command + r** on Mac.



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 myXamarinDevice --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 event logs from an IoT device named 'myiOSdevice'. Each event contains a timestamp, temperature, humidity, and device ID.

```
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.

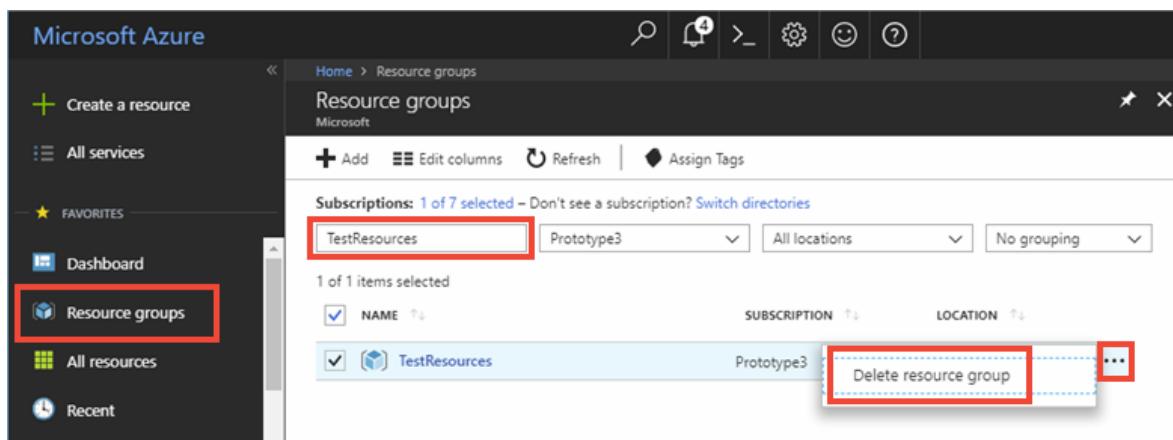
Otherwise, you can delete the Azure resources created in this article to avoid charges.

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 select **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, select ... 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 select **Delete**. After a few moments, the resource group and all of its contained resources are deleted.

Next steps

In this quickstart, you set up an IoT hub, registered a device, sent simulated telemetry to the hub from a Xamarin.Forms application, 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: Use Node.js to control a device connected to an Azure IoT hub

7/29/2020 • 9 minutes to read • [Edit Online](#)

In this quickstart, you use a direct method to control a simulated device connected to Azure IoT Hub. IoT Hub is an Azure service that enables you to manage your IoT devices from the cloud, and ingest high volumes of device telemetry to the cloud for storage or processing. You can use direct methods to remotely change the behavior of a device connected to your IoT hub. This quickstart uses two Node.js applications: a simulated device application that responds to direct methods called from a back-end application and a back-end application that calls the direct methods on the simulated device.

Prerequisites

- An Azure account with an active subscription. [Create one for free](#).
- [Node.js 10+](#).
- [A sample Node.js project](#).
- Port 8883 open in your firewall. The device sample in this quickstart uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

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

```
node --version
```

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|--|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. | <a data-bbox="826 1882 1101 1918" href="#">Launch Cloud Shell |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal . |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

Add Azure IoT Extension

Run the following command to add the Microsoft Azure IoT Extension for Azure CLI to your Cloud Shell instance. The IoT Extension adds IoT Hub, IoT Edge, and IoT Device Provisioning Service (DPS) specific commands to Azure CLI.

```
az extension add --name azure-iot
```

NOTE

This article uses the newest version of the Azure IoT extension, called `azure-iot`. The legacy version is called `azure-cli-iot-ext`. You should only have one version installed at a time. You can use the command `az extension list` to validate the currently installed extensions.

Use `az extension remove --name azure-cli-iot-ext` to remove the legacy version of the extension.

Use `az extension add --name azure-iot` to add the new version of the extension.

To see what extensions you have installed, use `az extension list`.

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. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.
 - **IoT Hub Name:** Enter a name for your hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

The screenshot shows the 'Basics' tab of the IoT hub creation wizard. It includes fields for Subscription, Resource group, Region, and IoT hub name, all enclosed in a red box. Below the form are navigation buttons: 'Review + create' (blue), '< Previous' (disabled), 'Next: Size and scale >' (highlighted with a red box), and 'Automation options'.

Home > New > IoT hub

IoT hub

Microsoft

Basics [Size and scale](#) [Tags](#) [Review + create](#)

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

Project details

Choose the subscription you'll use to manage deployments and costs. Use resource groups like folders to help you organize and manage resources.

Subscription * ⓘ

Resource group * ⓘ [Create new](#)

Region * ⓘ

IoT hub name * ⓘ

[Review + create](#) [< Previous](#) [Next: Size and scale >](#) [Automation options](#)

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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
Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center On
Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.
If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select Next: Tags to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure](#)

resources.

The screenshot shows the 'Tags' section of the IoT hub creation wizard. It displays two tag entries:

| Name | Value | Resource |
|------------|------------|----------|
| department | accounting | IoT Hub |
| | | IoT Hub |

Below the table are navigation buttons: 'Review + create' (highlighted in blue), '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' page for the IoT hub. The 'Tags' section is highlighted with a red box. The 'Create' button at the bottom left is also highlighted with a red box.

| Subscription | Personal testing |
|----------------|------------------|
| Resource group | iot-hubs |
| Region | West US 2 |
| IoT hub name | you-hub-name |

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 |
| Azure Security Center | 0.001 USD per device per month |

Tags

| | |
|------------|------------|
| department | accounting |
|------------|------------|

Bottom navigation buttons: 'Create' (highlighted in red), '< Previous: Tags', 'Next >', and 'Automation options'.

8. Select **Create** to create your new 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 command in Azure Cloud Shell to create the device identity.

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

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

```
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 \
--policy-name service --name {YourIoTHubName} --output table
```

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

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

You use this value later in the quickstart. This service connection string is different from the device connection string you noted in the previous step.

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 acknowledgment 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.

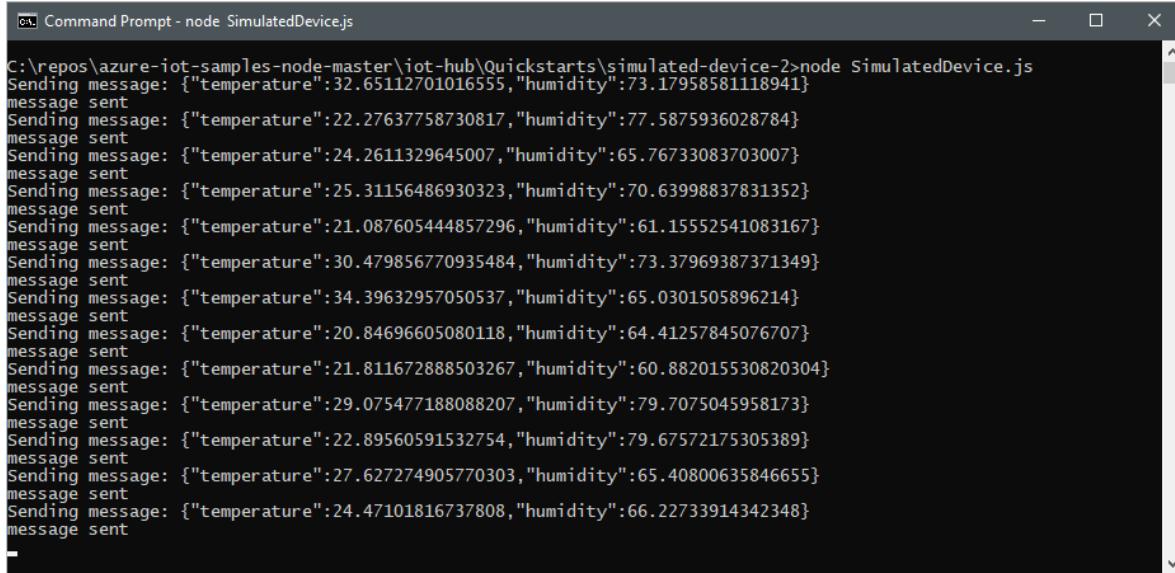
2. Open the **SimulatedDevice.js** file in a text editor of your choice.

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

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 acknowledgments. An IoT Hub back-end application typically runs in the cloud.

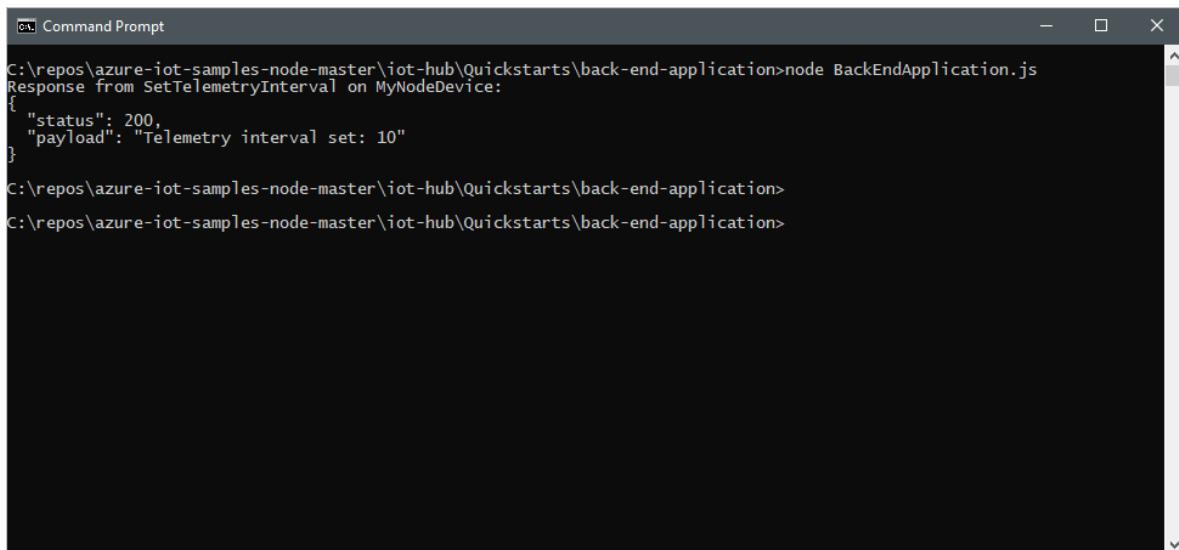
1. In another local terminal window, navigate to the root folder of the sample Node.js project. Then navigate to the **iot-hub\Quickstarts\back-end-application** folder.
2. Open the **BackEndApplication.js** file in a text editor of your choice.

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

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

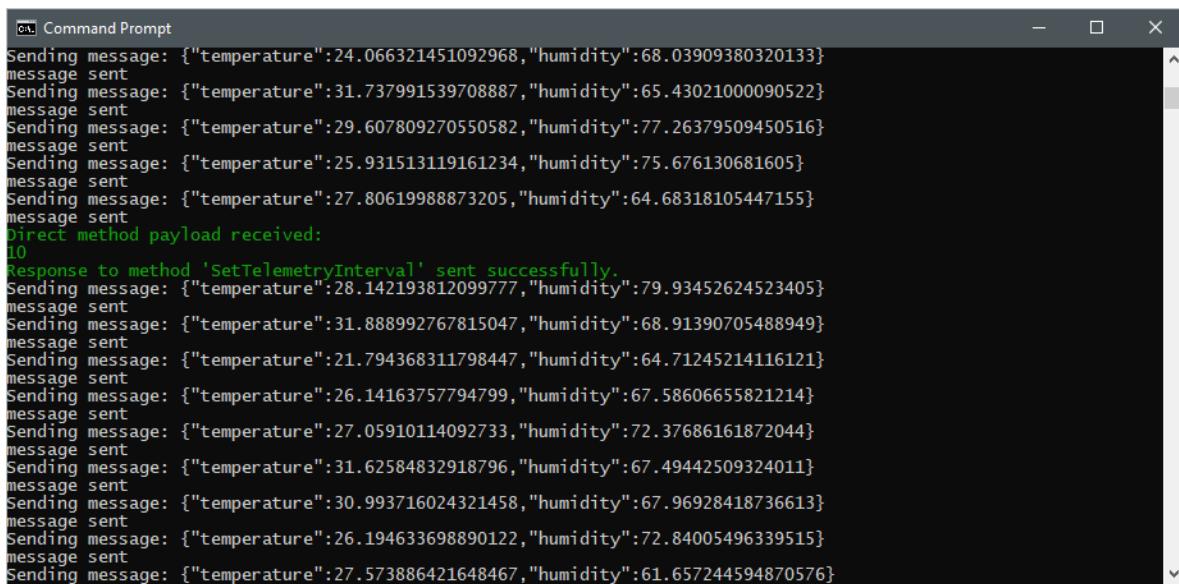
```
npm install  
node BackEndApplication.js
```

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



```
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 select **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, select ... 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'. It displays a table with one item: 'TestResources' (Subscription: Prototype3, Location: All locations). A red box highlights the search bar at the top with 'TestResources'. Another red box highlights the 'Delete resource group' button in the table's context menu.

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 select **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)

7/29/2020 • 10 minutes to read • [Edit Online](#)

IoT Hub is an Azure service that enables you to manage your IoT devices from the cloud, and ingest high volumes of device telemetry to the cloud for storage or processing. 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.

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|--|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal. |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

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
```

Run the following command to add the Microsoft Azure IoT Extension for Azure CLI to your Cloud Shell instance. The IOT Extension adds IoT Hub, IoT Edge, and IoT Device Provisioning Service (DPS) specific commands to Azure CLI.

```
az extension add --name azure-iot
```

NOTE

This article uses the newest version of the Azure IoT extension, called `azure-iot`. The legacy version is called `azure-cli-iot-ext`. You should only have one version installed at a time. You can use the command `az extension list` to validate the currently installed extensions.

Use `az extension remove --name azure-cli-iot-ext` to remove the legacy version of the extension.

Use `az extension add --name azure-iot` to add the new version of the extension.

To see what extensions you have installed, use `az extension list`.

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

Make sure that port 8883 is open in your firewall. The device sample in this quickstart uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

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. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these

limited features, you must select one of the supported regions.

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

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

The screenshot shows the 'Basics' tab of the IoT hub creation wizard. It includes fields for Subscription, Resource group, Region, and IoT hub name, all enclosed in a red box. Below the form are navigation buttons: 'Review + create' (blue), '< Previous' (disabled), 'Next: Size and scale >' (highlighted with a red box), and 'Automation options'.

Home > New > IoT hub

IoT hub
Microsoft

Basics [Size and scale](#) [Tags](#) [Review + create](#)

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

Project details

Choose the subscription you'll use to manage deployments and costs. Use resource groups like folders to help you organize and manage resources.

Subscription * ⓘ

Resource group * ⓘ [Create new](#)

Region * ⓘ

IoT hub name * ⓘ

[Review + create](#) < Previous [Next: Size and scale >](#) [Automation options](#)

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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 Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center On Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.
If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure](#)

resources.

The screenshot shows the 'Tags' section of the IoT hub configuration. It has tabs for Basics, Size and scale, Tags (which is selected), and Review + create. Below the tabs is a note about tags being name/value pairs used for categorization and billing. A table lists a single tag: 'department' with value 'accounting'. There are buttons for Review + create, < Previous: Size and scale, Next: Review + create >, and Automation options.

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' page for the IoT hub. It displays configuration details under three sections: Basics, Size and scale, and Tags. The 'Review + create' tab is highlighted with a red box. The 'Create' button at the bottom left is also highlighted with a red box. Navigation buttons for < Previous: Tags, Next >, and Automation options are at the bottom.

| Section | Setting | Value |
|----------------------------|--------------------------------|------------------|
| Basics | Subscription | Personal testing |
| | Resource group | iot-hubs |
| | Region | West US 2 |
| | IoT hub name | you-hub-name |
| | 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 | |
| Azure Security Center | 0.001 USD per device per month | |
| Tags | | |
| department | accounting | |

8. Select **Create** to create your new 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 command in Azure Cloud Shell to create the device identity.

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

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

```
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 --policy-name service --name {YourIoTHubName} --output table
```

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

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

You use this value later in the quickstart. This service connection string is different from the device connection string you noted in the previous step.

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 acknowledgment 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

earlier. Then save your changes to `SimulatedDevice.cs`.

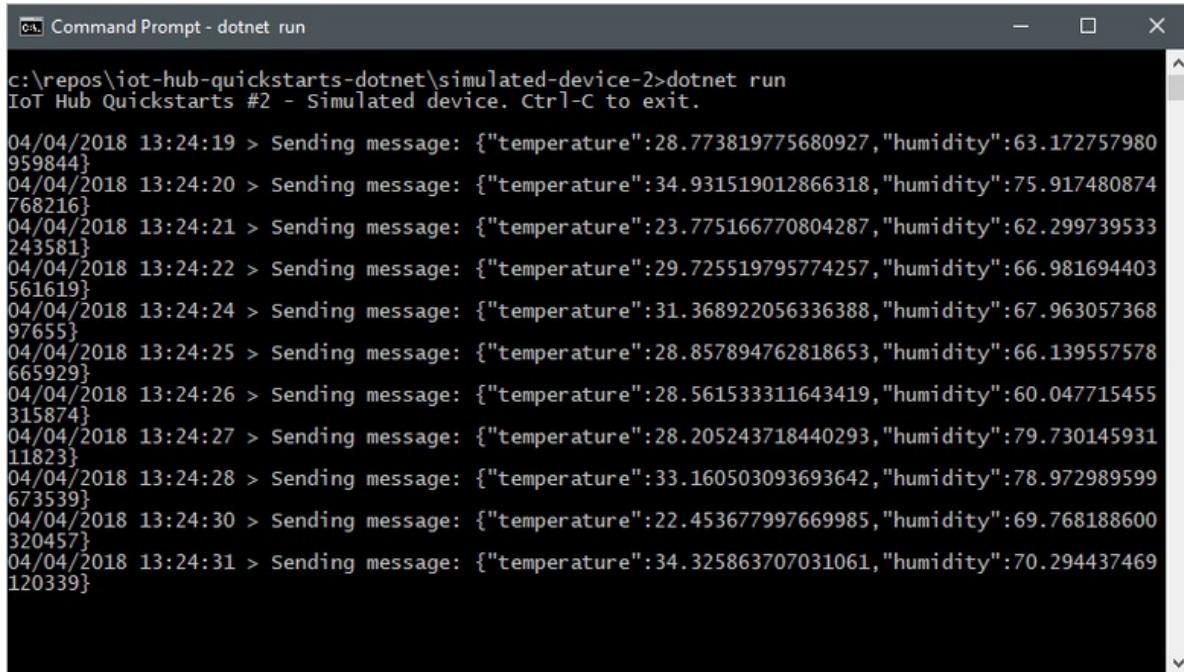
3. In the local terminal window, run the following commands to install the required packages for simulated device application:

```
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 acknowledgments. 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 earlier. Then save your changes to `BackEndApplication.cs`.

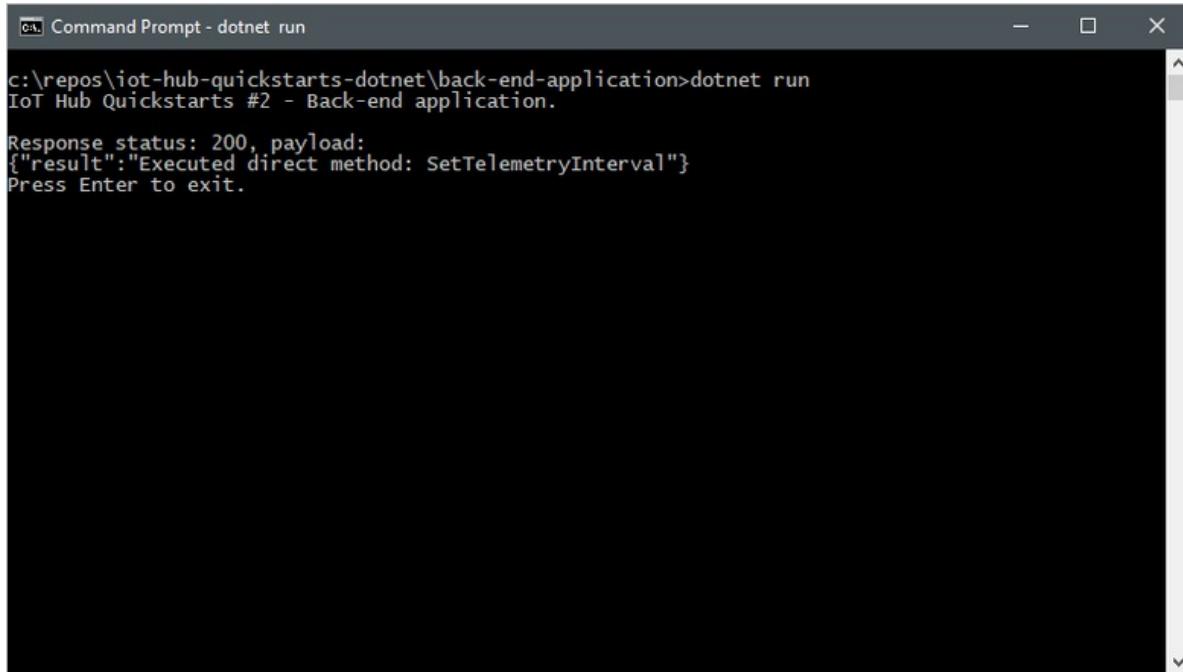
3. In the local terminal window, run the following commands to install the required libraries for the back-end application:

```
dotnet restore
```

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

```
dotnet run
```

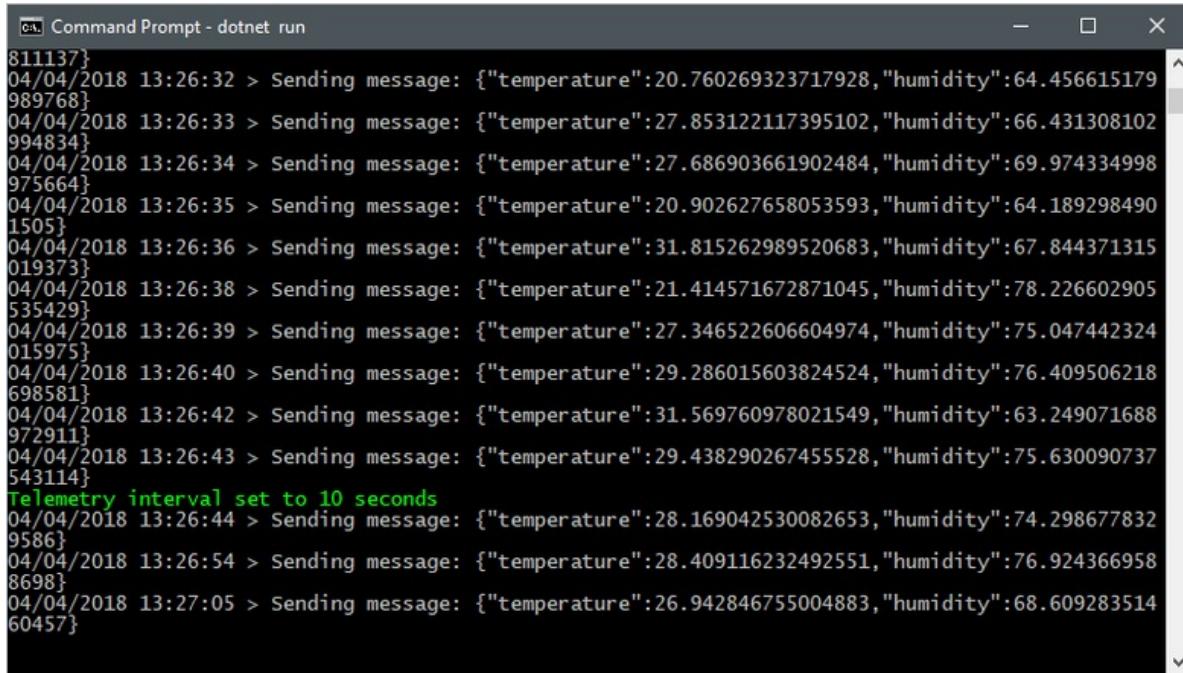
The following screenshot shows the output as the application makes a direct method call to the device and receives an acknowledgment:



```
c:\ repos\iot-hub-quickstarts-dotnet\back-end-application>dotnet run
IoT Hub Quickstarts #2 - Back-end application.

Response status: 200, payload:
{"result": "Executed direct method: SetTelemetryInterval"}
Press Enter to exit.
```

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:



```
811137}
04/04/2018 13:26:32 > Sending message: {"temperature":20.760269323717928,"humidity":64.456615179
989768}
04/04/2018 13:26:33 > Sending message: {"temperature":27.853122117395102,"humidity":66.431308102
994834}
04/04/2018 13:26:34 > Sending message: {"temperature":27.686903661902484,"humidity":69.974334998
975664}
04/04/2018 13:26:35 > Sending message: {"temperature":20.902627658053593,"humidity":64.189298490
1505}
04/04/2018 13:26:36 > Sending message: {"temperature":31.815262989520683,"humidity":67.844371315
019373}
04/04/2018 13:26:38 > Sending message: {"temperature":21.414571672871045,"humidity":78.226602905
535429}
04/04/2018 13:26:39 > Sending message: {"temperature":27.346522606604974,"humidity":75.047442324
015975}
04/04/2018 13:26:40 > Sending message: {"temperature":29.286015603824524,"humidity":76.409506218
698581}
04/04/2018 13:26:42 > Sending message: {"temperature":31.569760978021549,"humidity":63.249071688
972911}
04/04/2018 13:26:43 > Sending message: {"temperature":29.438290267455528,"humidity":75.630090737
543114}
Telemetry interval set to 10 seconds
04/04/2018 13:26:44 > Sending message: {"temperature":28.169042530082653,"humidity":74.298677832
9586}
04/04/2018 13:26:54 > Sending message: {"temperature":28.409116232492551,"humidity":76.924366958
8698}
04/04/2018 13:27:05 > Sending message: {"temperature":26.942846755004883,"humidity":68.609283514
60457}
```

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 select **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, select ... then **Delete resource group**.

The screenshot shows the Microsoft Azure portal interface. On the left, the sidebar has 'Resource groups' selected, highlighted with a red box. The main area is titled 'Resource groups' and shows a list of items. A search bar at the top is also highlighted with a red box. In the list, there is one item named 'TestResources'. To the right of this item, there is a context menu with options, one of which is 'Delete resource group', also highlighted with a red box. The list includes columns for NAME, SUBSCRIPTION, and LOCATION.

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 select **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 Azure IoT hub with Java

7/29/2020 • 10 minutes to read • [Edit Online](#)

In this quickstart, you use a direct method to control a simulated device connected to Azure IoT Hub with a Java application. IoT Hub is an Azure service that enables you to manage your IoT devices from the cloud and ingest high volumes of device telemetry to the cloud for storage or processing. You can use direct methods to remotely change the behavior of a device connected to your IoT hub. This quickstart uses two Java applications: a simulated device application that responds to direct methods called from a back-end application and a service application that calls the direct method on the simulated device.

Prerequisites

- An Azure account with an active subscription. [Create one for free](#).
- Java SE Development Kit 8. In [Java long-term support for Azure and Azure Stack](#), under **Long-term support**, select **Java 8**.
- [Apache Maven 3](#).
- [A sample Java project](#).
- Port 8883 open in your firewall. The device sample in this quickstart uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

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

```
java -version
```

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

```
mvn --version
```

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|--|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |

| OPTION | EXAMPLE/LINK |
|---|--|
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal . |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

Add Azure IoT Extension

Run the following command to add the Microsoft Azure IoT Extension for Azure CLI to your Cloud Shell instance. The IoT Extension adds IoT Hub, IoT Edge, and IoT Device Provisioning Service (DPS) specific commands to Azure CLI.

```
az extension add --name azure-iot
```

NOTE

This article uses the newest version of the Azure IoT extension, called `azure-iot`. The legacy version is called `azure-cli-iot-ext`. You should only have one version installed at a time. You can use the command `az extension list` to validate the currently installed extensions.

Use `az extension remove --name azure-cli-iot-ext` to remove the legacy version of the extension.

Use `az extension add --name azure-iot` to add the new version of the extension.

To see what extensions you have installed, use `az extension list`.

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. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource

group. For more information, see [Manage Azure Resource Manager resource groups](#).

- **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.
- **IoT Hub Name:** Enter a name for your hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

The screenshot shows the 'Create IoT hub' wizard in the Azure portal. The title bar says 'Home > New > IoT hub' and the page title is 'IoT hub'. Below the title, there are tabs: 'Basics' (which is selected), 'Size and scale', 'Tags', and 'Review + create'. A sub-header 'Project details' is present. A note says 'Create an IoT Hub to help you connect, monitor, and manage billions of your IoT assets.' with a 'Learn more' link. The main form area has four sections with red borders around them:

- Subscription ***: A dropdown menu showing 'Personal IoT items'.
- Resource group ***: A dropdown menu with 'Create new' and a 'Create new' button below it.
- Region ***: A dropdown menu showing 'East Asia'.
- IoT hub name ***: A text input field containing 'Once your hub is created, this name can't be changed'.

At the bottom, there are navigation buttons: 'Review + create' (blue), '< Previous', 'Next: Size and scale >' (highlighted with a red box), and 'Automation options'.

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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 [Determines how your IoT hub can scale. You can change this later if your needs increase.](#)

Azure Security Center [On](#) Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.
- If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure](#)

resources.

The screenshot shows the 'Tags' tab of the IoT hub configuration page. It displays a table with two rows of tag entries. The first row has 'Name' as 'department' and 'Value' as 'accounting'. The second row is partially visible. Below the table are navigation buttons: 'Review + create' (highlighted), '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

| Name | Value | Resource |
|------------|------------|----------|
| department | accounting | IoT Hub |
| | | IoT Hub |

Review + create < Previous: Size and scale Next: Review + create > Automation options

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' step of the IoT hub creation process. It lists the configuration settings: Subscription (Personal testing), Resource group (iot-hubs), Region (West US 2), IoT hub name (you-hub-name). Under 'Size and scale', it shows Pricing and scale tier (S1), Number of S1 IoT hub units (1), Messages per day (400,000), Cost per month (25.00 USD), and Azure Security Center (0.001 USD per device per month). Under 'Tags', it shows department (accounting). At the bottom, the 'Create' button is highlighted with a red box, and other buttons include '< Previous: Tags', 'Next >', and 'Automation options'.

Basics

Subscription: Personal testing
Resource group: iot-hubs
Region: West US 2
IoT hub name: you-hub-name

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
Azure Security Center: 0.001 USD per device per month

Tags

department: accounting

Create < Previous: Tags Next > Automation options

8. Select **Create** to create your new 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 command in Azure Cloud Shell to create the device identity.

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

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

```
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 --policy-name service --name {YourIoTHubName} --output table
```

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

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

You use this value later in the quickstart. This service connection string is different from the device connection string you noted in the previous step.

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 acknowledgment 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 earlier. Then save your changes to **SimulatedDevice.java**.

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
log4j:WARN No appenders could be found for logger (com.microsoft.azure.sdk.iot.device.IoTHubConnection).
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 acknowledgments. 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 earlier. Then save your changes to **BackEndApplication.java**.

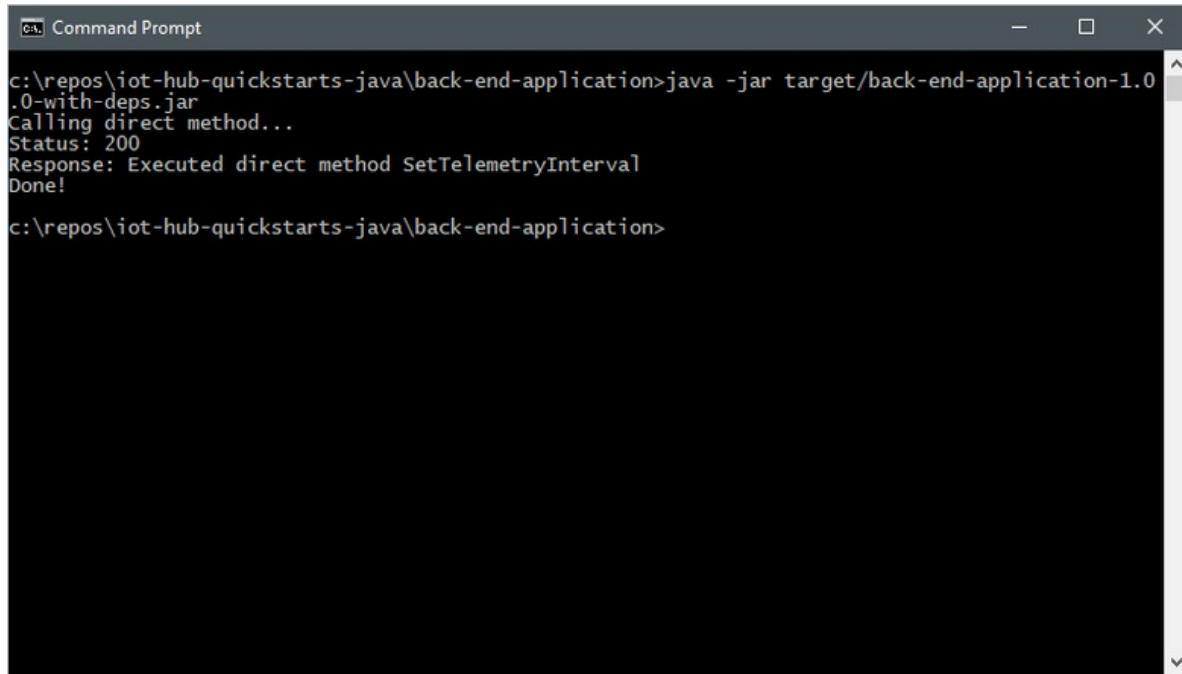
3. In the local terminal window, run the following commands to install the required libraries and build the back-end application:

```
mvn clean package
```

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

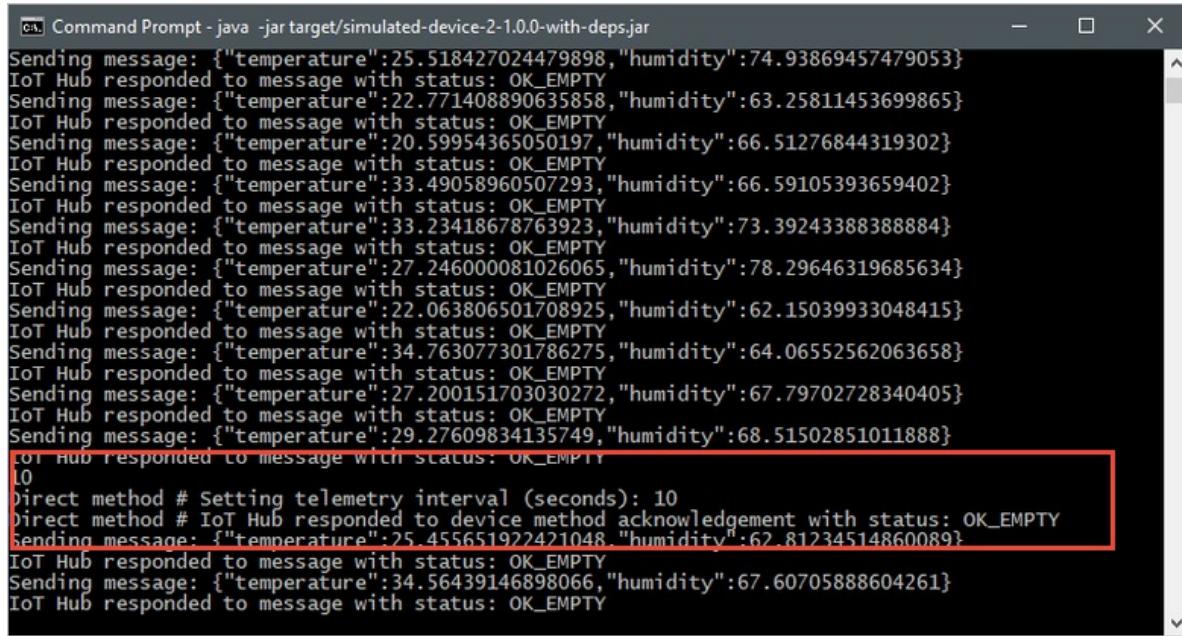
```
java -jar target/back-end-application-1.0.0-with-deps.jar
```

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



```
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:\repos\iot-hub-quickstarts-java\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
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 select **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, select ... 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 list of items. A search bar at the top has 'TestResources' typed into it, also highlighted with a red box. Below the search bar, there's a table with columns: NAME, SUBSCRIPTION, and LOCATION. One row is selected, showing 'TestResources' under NAME, 'Prototype3' under SUBSCRIPTION, and 'East US' under LOCATION. To the right of the row, there's a context menu with options like 'Delete resource group' and '...', with 'Delete resource group' 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 select **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)

7/29/2020 • 9 minutes to read • [Edit Online](#)

In this quickstart, you use a direct method to control a simulated device connected to Azure IoT Hub. IoT Hub is an Azure service that enables you to manage your IoT devices from the cloud and ingest high volumes of device telemetry to the cloud for storage or processing. You can use direct methods to remotely change the behavior of a device connected to your IoT hub. This quickstart uses two Python applications: a simulated device application that responds to direct methods called from a back-end application and a back-end application that calls the direct methods on the simulated device.

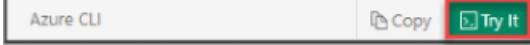
Prerequisites

- An Azure account with an active subscription. [Create one for free](#).
- [Python 3.7+](#). For other versions of Python supported, see [Azure IoT Device Features](#).
- [A sample Python project](#).
- Port 8883 open in your firewall. The device sample in this quickstart uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|--|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal. |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the Copy button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by

selecting **Cmd+Shift+V** on macOS.

4. Select **Enter** to run the code.

Add Azure IoT Extension

Run the following command to add the Microsoft Azure IoT Extension for Azure CLI to your Cloud Shell instance. The IoT Extension adds IoT Hub, IoT Edge, and IoT Device Provisioning Service (DPS) specific commands to Azure CLI.

```
az extension add --name azure-iot
```

NOTE

This article uses the newest version of the Azure IoT extension, called `azure-iot`. The legacy version is called `azure-cli-iot-ext`. You should only have one version installed at a time. You can use the command `az extension list` to validate the currently installed extensions.

Use `az extension remove --name azure-cli-iot-ext` to remove the legacy version of the extension.

Use `az extension add --name azure-iot` to add the new version of the extension.

To see what extensions you have installed, use `az extension list`.

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. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.
 - **IoT Hub Name:** Enter a name for your hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

Home > New > IoT hub

IoT hub

Microsoft

[X](#)

[Basics](#) [Size and scale](#) [Tags](#) [Review + create](#)

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

Project details

Choose the subscription you'll use to manage deployments and costs. Use resource groups like folders to help you organize and manage resources.

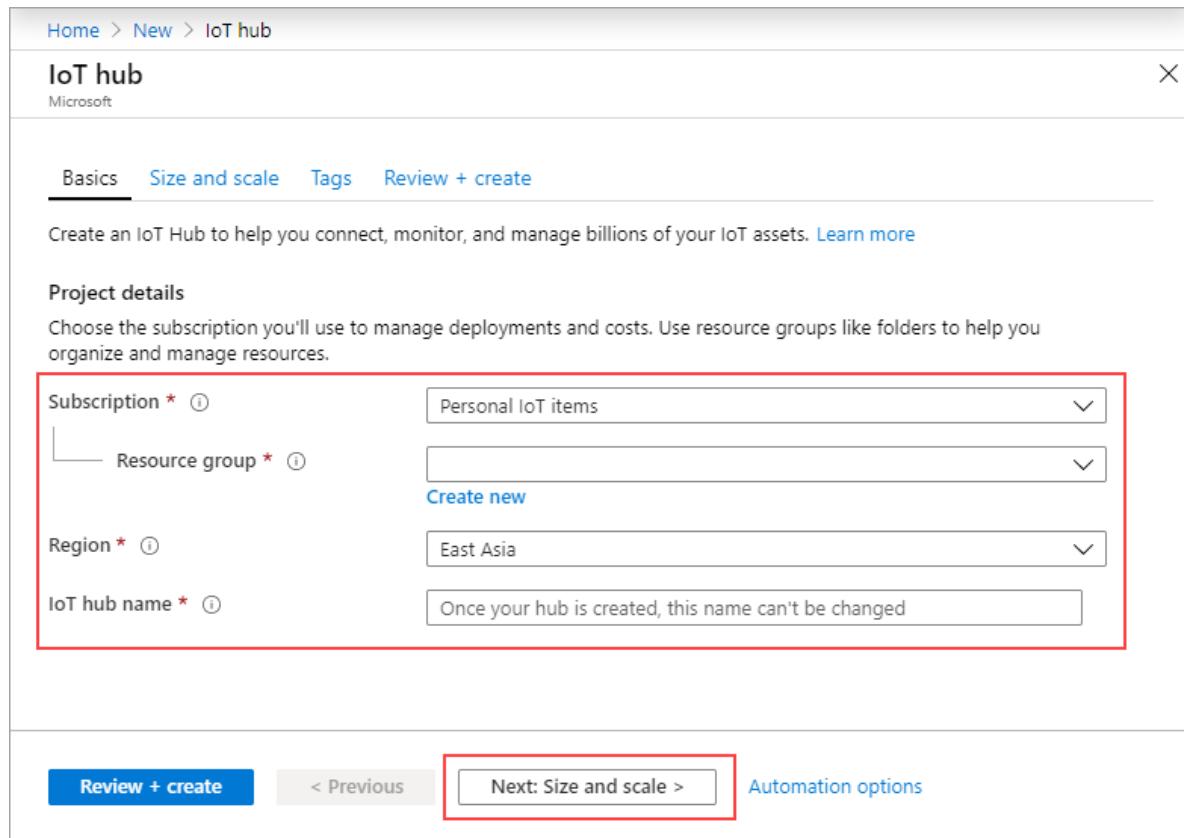
Subscription * ⓘ

Resource group * ⓘ [Create new](#)

Region * ⓘ

IoT hub name * ⓘ

[Review + create](#) [< Previous](#) [Next: Size and scale >](#) [Automation options](#)



5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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 [Determines how your IoT hub can scale. You can change this later if your needs increase.](#)

Azure Security Center [On](#) Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.
- If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure](#)

resources.

The screenshot shows the 'Tags' tab selected in the top navigation bar. Below it, a table lists a single tag entry:

| Name | Value | Resource |
|------------|------------|----------|
| department | accounting | IoT Hub |
| | | IoT Hub |

At the bottom, there are buttons for 'Review + create', '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' step. It displays the configuration details for the IoT hub, including:

- Basics**:
 - Subscription: Personal testing
 - Resource group: iot-hubs
 - Region: West US 2
 - IoT hub name: you-hub-name
- 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
 - Azure Security Center: 0.001 USD per device per month
- Tags**:
 - department: accounting

At the bottom, there are buttons for 'Create' (highlighted with a red box), '< Previous: Tags', 'Next >', and 'Automation options'.

8. Select **Create** to create your new 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 command in Azure Cloud Shell to create the device identity.

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

MyPythonDevice: This is the name of the device you're registering. It's recommended to use **MyPythonDevice** as shown. If you choose a different name for your device, you also need to use that name throughout this article, and update the device name in the sample applications before you run them.

```
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 \
--policy-name service \
--name {YourIoTHubName} \
--output table
```

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

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

You use this value later in the quickstart. This service connection string is different from the device connection string you noted in the previous step.

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 acknowledgment 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 earlier. Then save your changes to `SimulatedDevice.py`.

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

```
pip install azure-iot-device
```

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:

```
Command Prompt - python SimulatedDevice.py
(py37) F:\repos\azure-iot-samples-python\iot-hub\Quickstarts\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": 31.764595313901268,"humidity": 61.59077699258239}
Message sent
Sending message: {"temperature": 32.80925268431869,"humidity": 77.4070803322654}
Message sent
Sending message: {"temperature": 22.60289778169097,"humidity": 62.321647317976336}
Message sent
Sending message: {"temperature": 32.39186381677736,"humidity": 71.9483413320913}
Message sent
Sending message: {"temperature": 26.02554081975368,"humidity": 72.79495633892942}
Message sent
Sending message: {"temperature": 23.202841527317148,"humidity": 61.09035760073166}
Message sent
Sending message: {"temperature": 29.74112862950414,"humidity": 70.15661075943251}
Message sent
Sending message: {"temperature": 30.976137220387507,"humidity": 64.92489301605741}
Message sent
Sending message: {"temperature": 30.332526270005133,"humidity": 65.19700378670453}
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 acknowledgments. 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 earlier. Then save your changes to `BackEndApplication.py`.

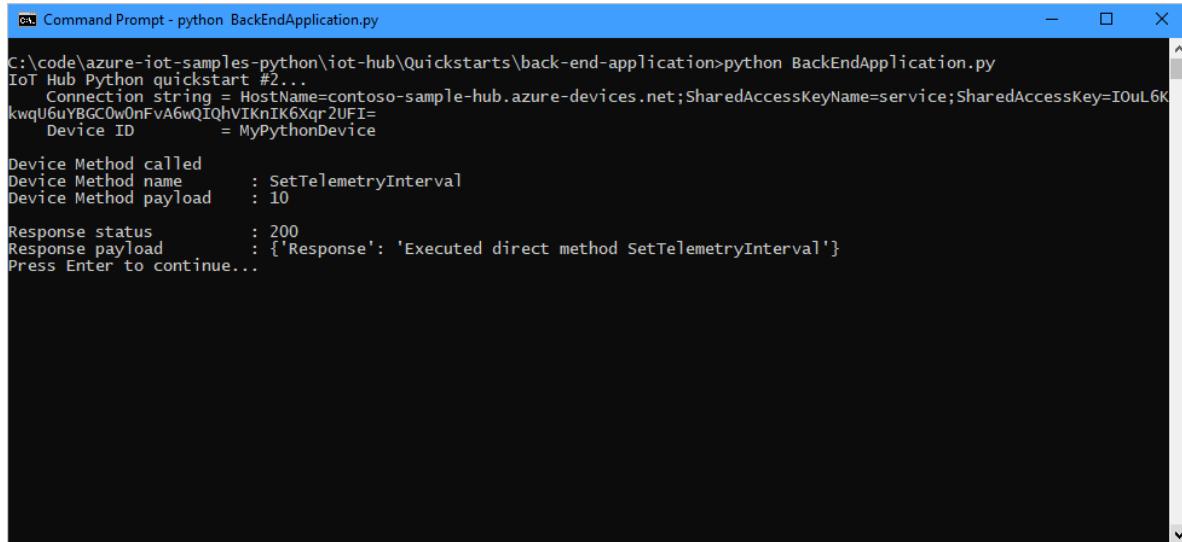
3. In the local terminal window, run the following commands to install the required libraries for the simulated device application:

```
pip install azure-iot-hub
```

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 acknowledgment:

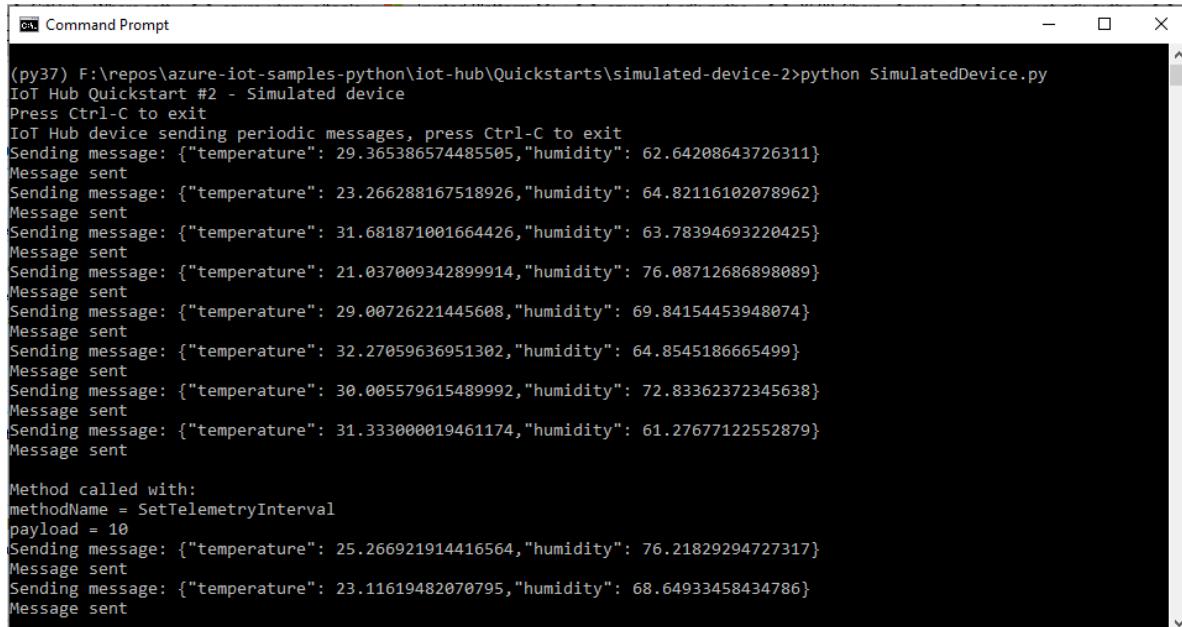


```
Command Prompt - python BackEndApplication.py
C:\code\azure-iot-samples-python\iot-hub\Quickstarts\back-end-application>python BackEndApplication.py
IoT Hub Python quickstart #...
Connection string = HostName=contoso-sample-hub.azure-devices.net;SharedAccessKeyName=service;SharedAccessKey=IOuL6K
kwqU6uYBGCoWnFvA6wQIhVIKnIK6Xqr2UFI=
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:



```
Command Prompt
(py37) F:\repos\azure-iot-samples-python\iot-hub\Quickstarts\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": 29.365386574485505,"humidity": 62.64208643726311}
Message sent
Sending message: {"temperature": 23.266288167518926,"humidity": 64.82116102078962}
Message sent
Sending message: {"temperature": 31.681871001664426,"humidity": 63.78394693220425}
Message sent
Sending message: {"temperature": 21.037009342899914,"humidity": 76.08712686898089}
Message sent
Sending message: {"temperature": 29.00726221445608,"humidity": 69.84154453948074}
Message sent
Sending message: {"temperature": 32.27059636951302,"humidity": 64.8545186665499}
Message sent
Sending message: {"temperature": 30.005579615489992,"humidity": 72.83362372345638}
Message sent
Sending message: {"temperature": 31.333000019461174,"humidity": 61.27677122552879}
Message sent

Method called with:
methodName = SetTelemetryInterval
payload = 10
Sending message: {"temperature": 25.266921914416564,"humidity": 76.21829294727317}
Message sent
Sending message: {"temperature": 23.11619482070795,"humidity": 68.64933458434786}
Message sent
```

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 select **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, select ... then **Delete resource group**.

The screenshot shows the Microsoft Azure portal interface. On the left, the sidebar 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 'Subscriptions: 1 of 7 selected'. A red box highlights the 'TestResources' subscription name in the filter bar. Another red box highlights the 'Delete resource group' option in the context menu for the selected resource group row.

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 select **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)

7/29/2020 • 12 minutes to read • [Edit Online](#)

In this quickstart, you use a direct method to control a simulated device connected to Azure IoT Hub. IoT Hub is an Azure service that enables you to manage your IoT devices from the cloud and ingest high volumes of device telemetry to the cloud for storage or processing. You can use direct methods to remotely change the behavior of a device connected to your IoT hub. This quickstart uses two applications: a simulated device application that responds to direct methods called from a back-end service application and a service application that calls the direct method on the Android device.

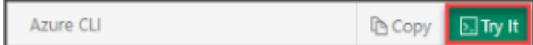
Prerequisites

- An Azure account with an active subscription. [Create one for free](#).
- [Android Studio with Android SDK 27](#). For more information, see [Install Android Studio](#).
- [Git](#).
- [Device SDK sample Android application](#), included in [Azure IoT Samples \(Java\)](#).
- [Service SDK sample Android application](#), included in [Azure IoT Samples \(Java\)](#).
- Port 8883 open in your firewall. The device sample in this quickstart uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|--|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal. |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.

2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

Add Azure IoT Extension

Run the following command to add the Microsoft Azure IoT Extension for Azure CLI to your Cloud Shell instance. The IoT Extension adds IoT Hub, IoT Edge, and IoT Device Provisioning Service (DPS) specific commands to Azure CLI.

```
az extension add --name azure-iot
```

NOTE

This article uses the newest version of the Azure IoT extension, called `azure-iot`. The legacy version is called `azure-cli-iot-ext`. You should only have one version installed at a time. You can use the command `az extension list` to validate the currently installed extensions.

Use `az extension remove --name azure-cli-iot-ext` to remove the legacy version of the extension.

Use `az extension add --name azure-iot` to add the new version of the extension.

To see what extensions you have installed, use `az extension list`.

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. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.
 - **IoT Hub Name:** Enter a name for your hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

Home > New > IoT hub

IoT hub

Microsoft

X

Basics Size and scale Tags Review + create

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

Project details

Choose the subscription you'll use to manage deployments and costs. Use resource groups like folders to help you organize and manage resources.

Subscription * ⓘ Personal IoT items

Resource group * ⓘ Create new

Region * ⓘ East Asia

IoT hub name * ⓘ Once your hub is created, this name can't be changed

Review + create < Previous **Next: Size and scale >** Automation options

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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 [Determines how your IoT hub can scale. You can change this later if your needs increase.](#)

Azure Security Center [On](#) Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.
- If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure](#)

resources.

The screenshot shows the 'Tags' tab selected in the top navigation bar. Below it, a table lists a single tag entry:

| Name | Value | Resource |
|------------|------------|----------|
| department | accounting | IoT Hub |
| | | IoT Hub |

At the bottom, there are buttons for 'Review + create', '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' step. It displays the configuration details for the IoT hub, including:

- Basics**:
 - Subscription: Personal testing
 - Resource group: iot-hubs
 - Region: West US 2
 - IoT hub name: you-hub-name
- 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
 - Azure Security Center: 0.001 USD per device per month
- Tags**:
 - department: accounting

At the bottom, there are buttons for 'Create' (highlighted with a red box), '< Previous: Tags', 'Next >', and 'Automation options'.

8. Select **Create** to create your new 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 command in Azure Cloud Shell to create the device identity.

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

MyAndroidDevice: This is the name of the device you're registering. It's recommended to use **MyAndroidDevice** as shown. If you choose a different name for your device, you also need to use that name throughout this article, and update the device name in the sample applications before you run them.

```
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 --policy-name service --name {YourIoTHubName} --output table
```

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

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

You use this value later in the quickstart. This service connection string is different from the device connection string you noted in the previous step.

Listen for direct method calls

Both of the samples for this quickstart are part of the `azure-iot-samples-java` repository on GitHub. Download or clone the [azure-iot-samples-java](#) repository.

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 acknowledgment 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 the device connection string you made a note of earlier.

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

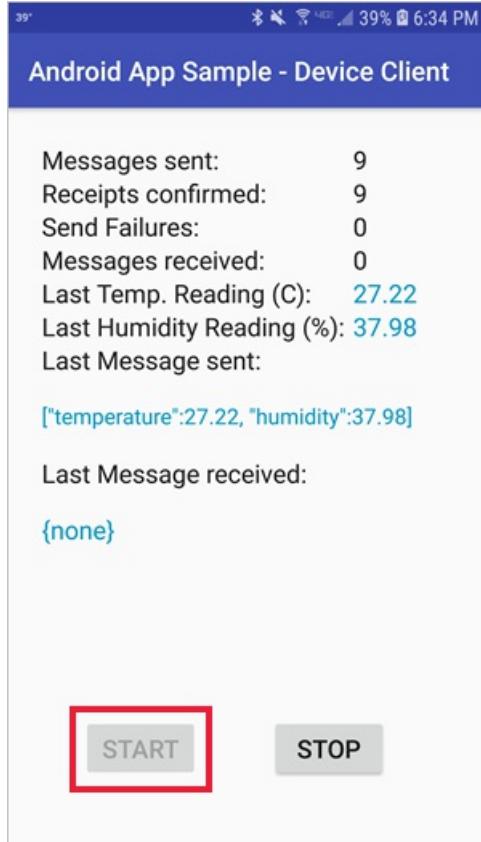
3. In Android Studio, click **File > Sync Project with Gradle Files**. Verify the build completes.

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.

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:



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 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'
```

By default, the telemetry app sends telemetry from the Android device every five 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 acknowledgments.

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's 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 examples that align more closely with a typical 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. Update the values for the **ConnectionString** and **DeviceId** properties with the service connection string you noted earlier and the Android device ID you registered.

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

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

NOTE

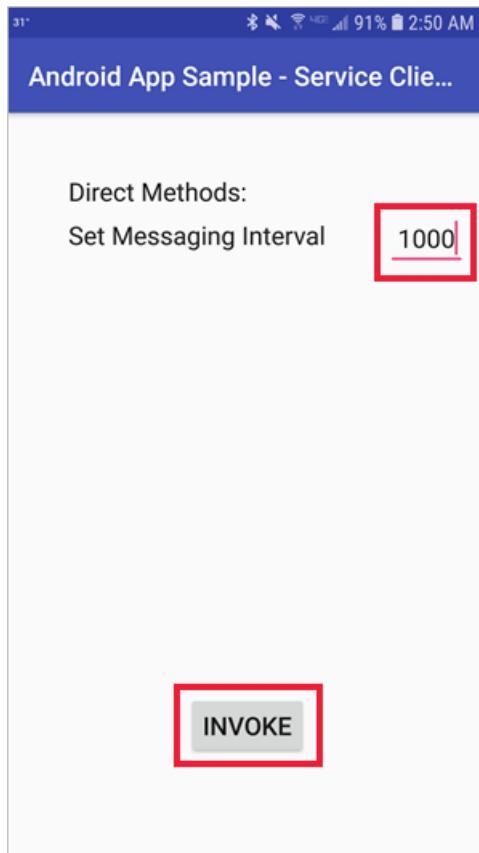
If the project sync fails, it may be for one of the following reasons:

- 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.

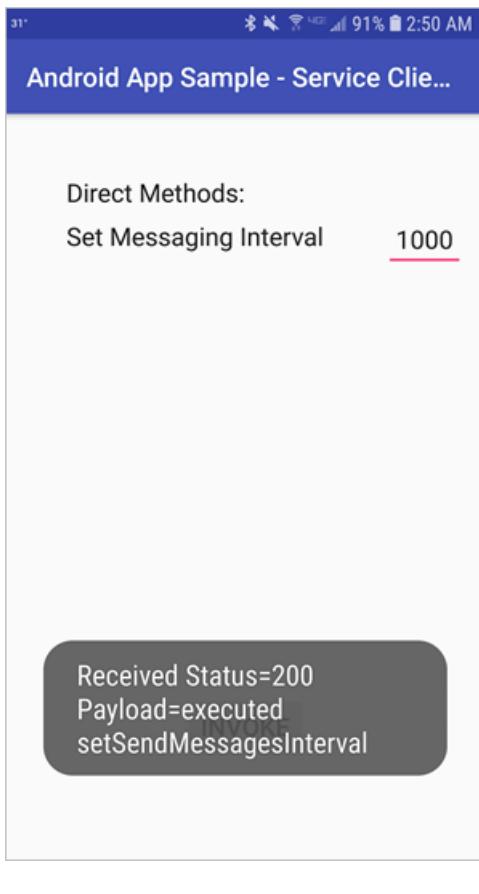
4. Once the build has completed, click **Run > Run 'app'**. Configure the app to run on a separate physical Android device or an Android emulator. For more information on running an Android app on a physical device or emulator, see [Run your app](#).

5. 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.



6. The app will receive an acknowledgment 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 select **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, select ... 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 select **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 a device application in C# via IoT Hub device streams (preview)

3/6/2020 • 9 minutes to read • [Edit Online](#)

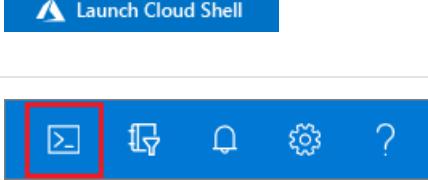
Azure IoT Hub currently supports device streams as a [preview feature](#).

[IoT Hub device streams](#) allow service and device applications to communicate in a secure and firewall-friendly manner. This quickstart involves two C# applications that take advantage of device streams to send data back and forth (echo).

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|--|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal . | |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

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

Prerequisites

- The preview of device streams is currently supported only for IoT hubs that are created in the following regions:
 - Central US
 - Central US EUAP
 - North Europe
 - Southeast Asia

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

- Download the [.NET Core SDK for multiple platforms from .NET](#).
- Verify the current version of C# on your development machine by using the following command:

```
dotnet --version
```

- Add the Azure IoT Extension for Azure CLI to your Cloud Shell instance by running the following command. The IOT Extension adds IoT Hub, IoT Edge, and IoT Device Provisioning Service (DPS)-specific commands to the Azure CLI.

```
az extension add --name azure-iot
```

NOTE

This article uses the newest version of the Azure IoT extension, called `azure-iot`. The legacy version is called `azure-cli-iot-ext`. You should only have one version installed at a time. You can use the command `az extension list` to validate the currently installed extensions.

Use `az extension remove --name azure-cli-iot-ext` to remove the legacy version of the extension.

Use `az extension add --name azure-iot` to add the new version of the extension.

To see what extensions you have installed, use `az extension list`.

- Download the [Azure IoT C# samples](#) and extract the ZIP archive. You need it on both the device side and the service side.

Create an IoT hub

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

- Sign in to the [Azure portal](#).
- From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
- Select **IoT Hub** from the search results, and then select **Create**.
- On the **Basics** tab, complete the fields as follows:
 - Subscription:** Select the subscription to use for your hub.
 - Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.
 - IoT Hub Name:** Enter a name for your hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

The screenshot shows the 'Basics' step of the IoT hub creation wizard. The page title is 'IoT hub' under 'Microsoft'. The navigation bar includes 'Home > New > IoT hub'. Below the title, there are tabs: 'Basics' (underlined), 'Size and scale', 'Tags', and 'Review + create'. A descriptive text 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 fields for 'Subscription' (Personal IoT items), 'Resource group' (dropdown with 'Create new' link), 'Region' (East Asia), and 'IoT hub name' (Once your hub is created, this name can't be changed). The bottom navigation bar includes 'Review + create' (button), '< Previous', 'Next: Size and scale >' (button with a red box around it), and 'Automation options'.

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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
Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center On
Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.
If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure](#)

resources.

The screenshot shows the 'Tags' step in the IoT hub creation wizard. At the top, there are tabs for 'Basics', 'Size and scale', 'Tags', and 'Review + create'. Below the tabs, a note says: 'Tags are name/value pairs. To categorize resources and consolidate billing, apply the same tag to multiple resources and resource groups. Your tags will update automatically if you change your resources.' A 'Learn more' link is provided. A table lists tags: one row has 'Name' as 'department' and 'Value' as 'accounting', with the 'Resource' column showing 'IoT Hub'. There is also an empty row for another tag. At the bottom, there are buttons for 'Review + create' (highlighted in blue), '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' step in the IoT hub creation wizard. The 'Review + create' tab is highlighted with a red box. The page displays the following information:

- Basics**:
 - Subscription: Personal testing
 - Resource group: iot-hubs
 - Region: West US 2
 - IoT hub name: you-hub-name
- 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
 - Azure Security Center: 0.001 USD per device per month
- Tags**:
 - department:accounting

At the bottom, there are buttons for 'Create' (highlighted in red), '< Previous: Tags', 'Next >', and 'Automation options'.

8. Select **Create** to create your new 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 use Azure Cloud Shell to

register a simulated device.

1. To create the device identity, run the following command in Cloud Shell:

NOTE

- Replace the *YourIoTHubName* placeholder with the name you chose for your IoT hub.
- For the name of the device you're registering, it's recommended to use *MyDevice* as shown. If you choose a different name for your device, use that name throughout this article, and update the device name in the sample applications before you run them.

```
az iot hub device-identity create --hub-name {YourIoTHubName} --device-id MyDevice
```

2. To get the *device connection string* for the device that you just registered, run the following command in Cloud Shell:

NOTE

Replace the *YourIoTHubName* placeholder with the name you chose for your IoT hub.

```
az iot hub device-identity show-connection-string --hub-name {YourIoTHubName} --device-id MyDevice --output table
```

Note the returned device connection string for later use in this quickstart. It looks like the following example:

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

3. 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:

NOTE

Replace the *YourIoTHubName* placeholder with the name you chose for your IoT hub.

```
az iot hub show-connection-string --policy-name service --name {YourIoTHubName} --output table
```

Note the returned service connection string for later use in this quickstart. It looks like the following example:

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

Communicate between the device and the service via device streams

In this section, you run both the device-side application and the service-side application and communicate between the two.

Run the service-side application

In a local terminal window, navigate to the `iot-hub/Quickstarts/device-streams-echo/service` directory in your unzipped project folder. Keep the following information handy:

| PARAMETER NAME | PARAMETER VALUE |
|-------------------------|---|
| ServiceConnectionString | The service connection string of your IoT hub. |
| MyDevice | The identifier of the device you created earlier. |

Compile and run the code with the following commands:

```
cd ./iot-hub/Quickstarts/device-streams-echo/service/
# Build the application
dotnet build

# Run the application
# In Linux or macOS
dotnet run "{ServiceConnectionString}" "MyDevice"

# In Windows
dotnet run {ServiceConnectionString} MyDevice
```

The application will wait for the device application to become available.

NOTE

A timeout occurs if the device-side application doesn't respond in time.

Run the device-side application

In another local terminal window, navigate to the `iot-hub/Quickstarts/device-streams-echo/device` directory in your unzipped project folder. Keep the following information handy:

| PARAMETER NAME | PARAMETER VALUE |
|------------------------|---|
| DeviceConnectionString | The device connection string of your IoT Hub. |

Compile and run the code with the following commands:

```
cd ./iot-hub/Quickstarts/device-streams-echo/device/
# Build the application
dotnet build

# Run the application
# In Linux or macOS
dotnet run "{DeviceConnectionString}"

# In Windows
dotnet run {DeviceConnectionString}
```

At the end of the last step, the service-side application initiates a stream to your device. After the stream is established, the application sends a string buffer to the service over the stream. In this sample, the service-side application simply echoes back the same data to the device, which demonstrates a successful bidirectional communication between the two applications.

Console output on the device side:

```
Command Prompt - dotnet run {DeviceConnectionString}
C:\...\azure-iot-samples-csharp-master\iot-hub\Quickstarts\device-streams-echo\device>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:

```
Command Prompt - dotnet run {ServiceConnectionString} MyDevice
C:\...\azure-iot-samples-csharp-master\iot-hub\Quickstarts\device-streams-echo\service>dotnet run {ServiceConnectionString} MyDevice

Stream response received: Name=TestStream IsAccepted=True
Starting streaming
Done streaming
Stream response received: Name=TestStream IsAccepted=True
Starting streaming
Done streaming
```

The traffic being sent over the stream is tunneled through the IoT hub rather than sent directly. The benefits provided are detailed in [Device streams benefits](#).

Clean up resources

If you plan to continue to the next recommended article, you can keep and reuse the resources you've already created.

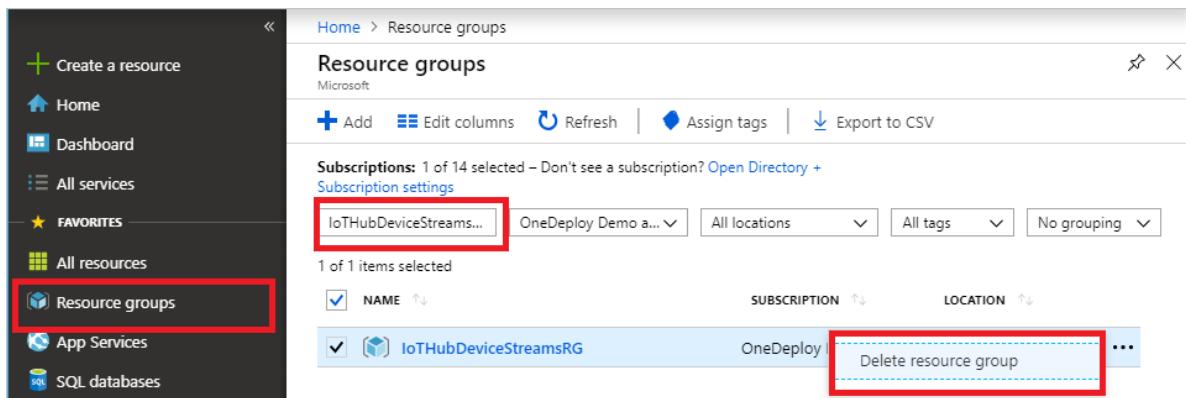
Otherwise, to avoid charges, you can delete the Azure resources that you created in this article.

IMPORTANT

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

To delete a resource group by name:

1. Sign in to the [Azure portal](#), and then select **Resource groups**.
2. In the **Filter by name** box, enter the name of the resource group that contains your IoT hub.
3. In the result list, to the right of your resource group, select the ellipsis (...), and then select **Delete resource group**.



4. To confirm the deletion of the resource group, reenter the resource group name, and then select **Delete**. After a few moments, the resource group and all its contained resources are deleted.

Next steps

In this quickstart, you set up an IoT hub, registered a device, established a device stream between C# applications on the device and service sides, and used the stream to send data back and forth between the applications.

To learn more about device streams, see:

[Device streams overview](#)

Quickstart: Communicate to a device application in Node.js via IoT Hub device streams (preview)

3/6/2020 • 9 minutes to read • [Edit Online](#)

In this quickstart, you run a service-side application and set up communication between a device and service by using device streams. Azure IoT Hub device streams allow service and device applications to communicate in a secure and firewall-friendly manner. During public preview, the 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.

Prerequisites

- Completion of [Communicate to device apps in C via IoT Hub device streams](#) or [Communicate to device apps in C# via IoT Hub device streams](#).
- An Azure account with an active subscription. [Create one for free](#).
- [Nodejs 10+](#).
- [A sample Node.js project](#).

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

```
node --version
```

Microsoft Azure IoT Hub currently supports device streams as a [preview feature](#).

IMPORTANT

The preview of device streams is currently only supported for IoT Hubs created in the following regions:

- Central US
- Central US EUAP
- North Europe
- Southeast Asia

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|--|---------------------------|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. | Azure CLI |

| OPTION | EXAMPLE/LINK |
|---|--|
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal . |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

Add Azure IoT Extension

Run the following command to add the Microsoft Azure IoT Extension for Azure CLI to your Cloud Shell instance. The IoT Extension adds IoT Hub, IoT Edge, and IoT Device Provisioning Service (DPS) commands to Azure CLI.

```
az extension add --name azure-iot
```

NOTE

This article uses the newest version of the Azure IoT extension, called `azure-iot`. The legacy version is called `azure-cli-iot-ext`. You should only have one version installed at a time. You can use the command `az extension list` to validate the currently installed extensions.

Use `az extension remove --name azure-cli-iot-ext` to remove the legacy version of the extension.

Use `az extension add --name azure-iot` to add the new version of the extension.

To see what extensions you have installed, use `az extension list`.

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. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter **IoT Hub** in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).

- **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.
- **IoT Hub Name:** Enter a name for your hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

The screenshot shows the 'Basics' step of the IoT hub creation wizard. At the top, there's a breadcrumb navigation: Home > New > IoT hub. The page title is 'IoT hub' with a Microsoft logo. Below the title, there are tabs: Basics (which is underlined), Size and scale, Tags, and Review + create. A sub-instruction 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, all marked with a red asterisk indicating they are required:

- Subscription: Personal IoT items
- Resource group: (empty dropdown) with a 'Create new' link below it.
- Region: East Asia
- IoT hub name: Once your hub is created, this name can't be changed

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

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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
Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center On
Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.
If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure](#)

resources.

The screenshot shows the 'Tags' section of the IoT hub configuration. It displays a table with one row of data:

| Name | Value | Resource |
|------------|------------|----------|
| department | accounting | IoT Hub |
| | | IoT Hub |

Below the table are navigation buttons: 'Review + create' (highlighted in blue), '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' page for the IoT hub. The configuration includes:

- Basics**:
 - Subscription: Personal testing
 - Resource group: iot-hubs
 - Region: West US 2
 - IoT hub name: you-hub-name
- 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
 - Azure Security Center: 0.001 USD per device per month
- Tags**:
 - department: accounting

At the bottom are buttons: 'Create' (highlighted in red), '< Previous: Tags', 'Next >', and 'Automation options'.

8. Select **Create** to create your new 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 command in Azure Cloud Shell to create the device identity.

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

MyDevice: This is the name for the device you're registering. It's recommended to use **MyDevice** as shown. If you choose a different name for your device, you also need to use that name throughout this article, and update the device name in the sample applications before you run them.

```
az iot hub device-identity create --hub-name {YourIoTHubName} --device-id MyDevice
```

2. 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 --name {YourIoTHubName} --output table
```

Note the returned service connection string for later use in this quickstart. It looks like the following example:

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

Communicate between device and service via device streams

In this section, you run both the device-side application and the service-side application and communicate between the two.

Run the device-side application

As mentioned earlier, IoT Hub Node.js SDK only supports device streams on the service side. For a device-side application, use one of the accompanying device programs available in these quickstarts:

- [Communicate to device apps in C via IoT Hub device streams](#)
- [Communicate to device apps in C# via IoT Hub device streams](#)

Ensure the device-side application is running before proceeding to the next step.

Run the service-side application

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 it to send and receive data.

Assuming the device-side application is running, follow the steps below in a local terminal window 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="{ServiceConnectionString}"
export STREAMING_TARGET_DEVICE="MyDevice"

# In Windows
SET IOTHUB_CONNECTION_STRING={ServiceConnectionString}
SET STREAMING_TARGET_DEVICE=MyDevice
```

Change the ServiceConnectionString placeholder to match your service connection string, and **MyDevice** to match your device ID if you gave yours a different name.

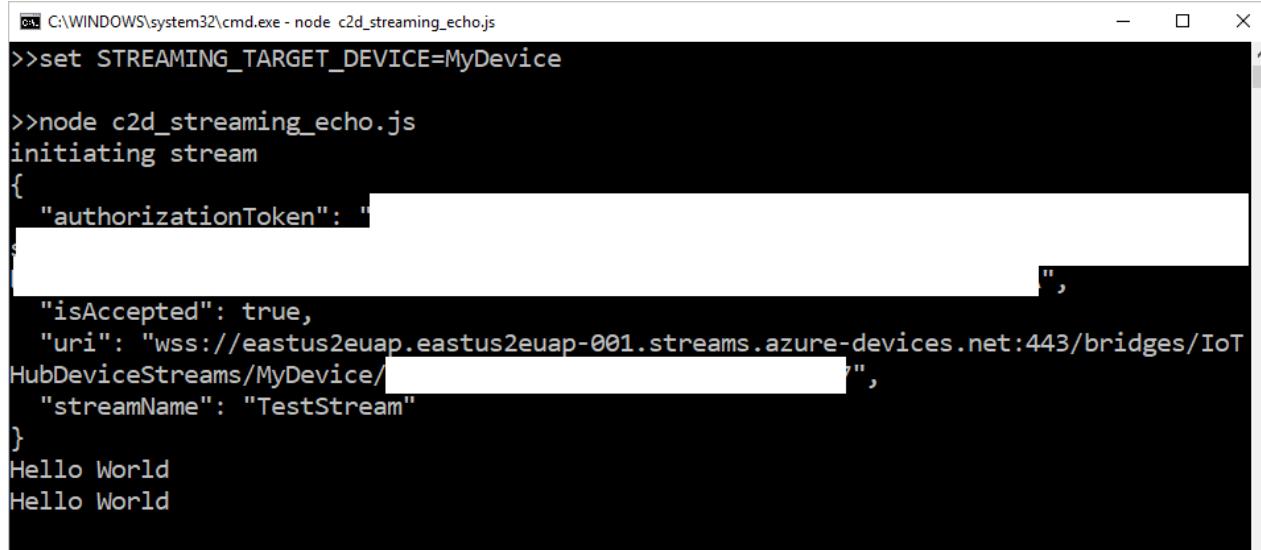
- Navigate to `Quickstarts/device-streams-service` in your unzipped project folder and run the sample using node.

```
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.



```
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 plan to continue to the next recommended article, you can keep and reuse the resources you've already created.

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

IMPORTANT

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

To delete a resource group by name:

1. Sign in to the [Azure portal](#), and then select **Resource groups**.
2. In the **Filter by name** box, enter the name of the resource group that contains your IoT hub.
3. In the result list, to the right of your resource group, select the ellipsis (...), and then select **Delete resource group**.

The screenshot shows the Azure Resource Groups blade. On the left, there's a sidebar with navigation links: Create a resource, Home, Dashboard, All services, Favorites, All resources, and Resource groups (which is highlighted with a red box). The main area shows a table of resource groups. At the top of the table, there are filters for Subscriptions, OneDeploy Demo a..., All locations, All tags, and No grouping. A search bar also shows 'IoTHubDeviceStreams...'. The table has columns for NAME, SUBSCRIPTION, and LOCATION. One item is listed: 'IoTHubDeviceStreamsRG' under 'OneDeploy'. To the right of this row, there's a 'Delete resource group' button, which is also highlighted with a red box. The table header includes sorting arrows for each column.

4. To confirm the deletion of the resource group, reenter the resource group name, and then select **Delete**. After a few moments, the resource group and all its contained resources are deleted.

Next steps

In this quickstart, you 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)

5/21/2020 • 9 minutes to read • [Edit Online](#)

Azure IoT Hub currently supports device streams as a [preview feature](#).

[IoT Hub device streams](#) allow service and device applications to communicate in a secure and firewall-friendly manner. During public preview, the C SDK supports device streams on the device side only. As a result, this quickstart covers instructions to run only the device-side application. To run a corresponding service-side application, see these articles:

- [Communicate to device apps in C# via IoT Hub device streams](#)
- [Communicate to device apps in Node.js via IoT Hub device streams](#)

The device-side C application in this quickstart has the following functionality:

- Establish a device stream to an IoT device.
- Receive data that's sent from the service-side application and echo it back.

The code demonstrates the initiation process of a device stream, as well as how to use it to send and receive data.

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|---|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal . |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.

4. Select **Enter** to run the code.

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

Prerequisites

You need the following prerequisites:

- Install [Visual Studio 2019](#) with the **Desktop development with C++** workload enabled.
- Install the latest version of [Git](#).
- Run the following command to add the Azure IoT Extension for Azure CLI to your Cloud Shell instance. The IOT Extension adds IoT Hub, IoT Edge, and IoT Device Provisioning Service (DPS)-specific commands to the Azure CLI.

```
az extension add --name azure-iot
```

NOTE

This article uses the newest version of the Azure IoT extension, called `azure-iot`. The legacy version is called `azure-cli-iot-ext`. You should only have one version installed at a time. You can use the command `az extension list` to validate the currently installed extensions.

Use `az extension remove --name azure-cli-iot-ext` to remove the legacy version of the extension.

Use `az extension add --name azure-iot` to add the new version of the extension.

To see what extensions you have installed, use `az extension list`.

The preview of device streams is currently supported only for IoT hubs that are created in the following regions:

- Central US
- Central US EUAP
- North Europe
- Southeast Asia

Prepare the development environment

For this quickstart, you use the [Azure IoT device SDK for C](#). You prepare a development environment used to clone and build the [Azure IoT C SDK](#) from GitHub. The SDK on GitHub includes the sample code that's used in this quickstart.

NOTE

Before you begin this procedure, be sure that Visual Studio is installed with the **Desktop development with C++** workload.

1. Install the [CMake build system](#) as described on the download page.
2. Open a command prompt or Git Bash shell. Run the following commands to clone the [Azure IoT C SDK](#) GitHub repository:

```
git clone -b public-preview https://github.com/Azure/azure-iot-sdk-c.git
cd azure-iot-sdk-c
git submodule update --init
```

This operation should take a few minutes.

3. Create a *cmake* subdirectory in the root directory of the git repository, and navigate to that folder. Run the following commands from the *azure-iot-sdk-c* directory:

```
mkdir cmake
cd cmake
```

4. Run the following commands from the *cmake* directory to build a version of the SDK that's specific to your development client platform.

- In Linux:

```
cmake ..
make -j
```

- In Windows, open a [Developer Command Prompt for Visual Studio](#). Run the command for your version of Visual Studio. This quickstart uses Visual Studio 2019. These commands create a Visual Studio solution for the simulated device in the *cmake* directory.

```
rem For VS2015
cmake .. -G "Visual Studio 14 2015"

rem Or for VS2017
cmake .. -G "Visual Studio 15 2017"

rem Or for VS2019
cmake .. -G "Visual Studio 16 2019"

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. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the + **Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these

limited features, you must select one of the supported regions.

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

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

The screenshot shows the 'Basics' step of the IoT hub creation wizard. At the top, there's a breadcrumb navigation: Home > New > IoT hub. The title is 'IoT hub' with a Microsoft logo. Below the title are four tabs: Basics (underlined), Size and scale, Tags, and Review + create. A sub-header 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 asks to choose a subscription, resource group, region, and IoT hub name. The 'Subscription' dropdown is set to 'Personal IoT items'. The 'Resource group' dropdown has a placeholder 'Create new' and a 'Create new' button. The 'Region' dropdown is set to 'East Asia'. The 'IoT hub name' input field contains 'Once your hub is created, this name can't be changed'. A red box highlights the 'Subscription', 'Resource group', 'Region', and 'IoT hub name' fields. At the bottom, there are buttons for 'Review + create' (blue), '< Previous' (disabled), 'Next: Size and scale >' (highlighted with a red box), and 'Automation options'.

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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 Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center On Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.
If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure](#)

resources.

The screenshot shows the 'Tags' tab of the IoT hub configuration page. It displays a table with two rows of tag entries:

| Name | Value | Resource | Actions |
|------------|------------|----------|---------|
| department | accounting | IoT Hub | ... |
| | | IoT Hub | |

Below the table are navigation buttons: 'Review + create' (highlighted in blue), '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' step of the IoT hub creation process. It displays the following configuration details:

| Section | Setting | Value |
|----------------------------|--------------------------------|------------------|
| Basics | Subscription | Personal testing |
| | Resource group | iot-hubs |
| | Region | West US 2 |
| | IoT hub name | you-hub-name |
| | 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 | |
| Azure Security Center | 0.001 USD per device per month | |
| Tags | | |
| department | accounting | |

At the bottom are buttons: 'Create' (highlighted in red), '< Previous: Tags', 'Next >', and 'Automation options'.

8. Select **Create** to create your new hub. Creating the hub takes a few minutes.

Register a device

You must register a device with your IoT hub before it can connect. In this section, you use Azure Cloud Shell with

the IoT Extension to register a simulated device.

1. To create the device identity, run the following command in Cloud Shell:

NOTE

- Replace the *YourIoTHubName* placeholder with the name you chose for your IoT hub.
- For the name of the device you're registering, it's recommended to use *MyDevice* as shown. If you choose a different name for your device, use that name throughout this article, and update the device name in the sample applications before you run them.

```
az iot hub device-identity create --hub-name {YourIoTHubName} --device-id MyDevice
```

2. To get the *device connection string* for the device that you just registered, run the following command in Cloud Shell:

NOTE

Replace the *YourIoTHubName* placeholder with the name you chose for your IoT hub.

```
az iot hub device-identity show-connection-string --hub-name {YourIoTHubName} --device-id MyDevice --output table
```

Note the returned device connection string for later use in this quickstart. It looks like the following example:

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

Communicate between the device and the service via device streams

In this section, you run both the device-side application and the service-side application and communicate between the two.

Run the device-side application

To run the device-side application, follow these steps:

1. Provide your device credentials by editing the `iothub_client_c2d_streaming_sample.c` source file in the `iothub_client/samples/iothub_client_c2d_streaming_sample` folder and adding your device connection string.

```
/* Paste in your iothub connection string */
static const char* connectionString = "{DeviceConnectionString}";
```

2. Compile the code with the following commands:

```
# In Linux
# Go to the sample's folder cmake/iothub_client/samples/iothub_client_c2d_streaming_sample
make -j
```

```
rem In Windows  
rem Go to the cmake folder at the root of repo  
cmake --build . -- /m /p:Configuration=Release
```

3. Run the compiled program:

```
# In Linux  
# Go to the sample's folder cmake/iothub_client/samples/iothub_client_c2d_streaming_sample  
../iothub_client_c2d_streaming_sample
```

```
rem In Windows  
rem Go to the 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 previously, the IoT Hub C SDK supports device streams on the device side only. To build and run the accompanying service-side application, follow the instructions in one of the following quickstarts:

- [Communicate to a device app in C# via IoT Hub device streams](#)
- [Communicate to a device app in Node.js via IoT Hub device streams](#)

Clean up resources

If you plan to continue to the next recommended article, you can keep and reuse the resources you've already created.

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

IMPORTANT

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

To delete a resource group by name:

1. Sign in to the [Azure portal](#), and then select **Resource groups**.
2. In the **Filter by name** box, enter the name of the resource group that contains your IoT hub.
3. In the result list, to the right of your resource group, select the ellipsis (...), and then select **Delete resource group**.

The screenshot shows the Azure Resource Groups blade. On the left, there's a navigation menu with options like 'Create a resource', 'Home', 'Dashboard', 'All services', 'FAVORITES' (which includes 'All resources', 'Resource groups', 'App Services', and 'SQL databases'), and 'Resource groups' (which is highlighted with a red box). The main area is titled 'Resource groups' and shows a table with one item selected: 'IoTHubDeviceStreams...'. 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'. To the right of the table, there's a 'More options' button (...). A red box highlights the 'Delete resource group' link next to the more options button.

4. To confirm the deletion of the resource group, reenter the resource group name, and then select **Delete**.
After a few moments, the resource group and all its contained resources are deleted.

Next steps

In this quickstart, you set up an IoT hub, registered a device, established a device stream between a C application on the device and another application on the service side, and used the stream to send data back and forth between the applications.

To learn more about device streams, see:

[Device streams overview](#)

Quickstart: Enable SSH and RDP over an IoT Hub device stream by using a C# proxy application (preview)

4/2/2020 • 12 minutes to read • [Edit Online](#)

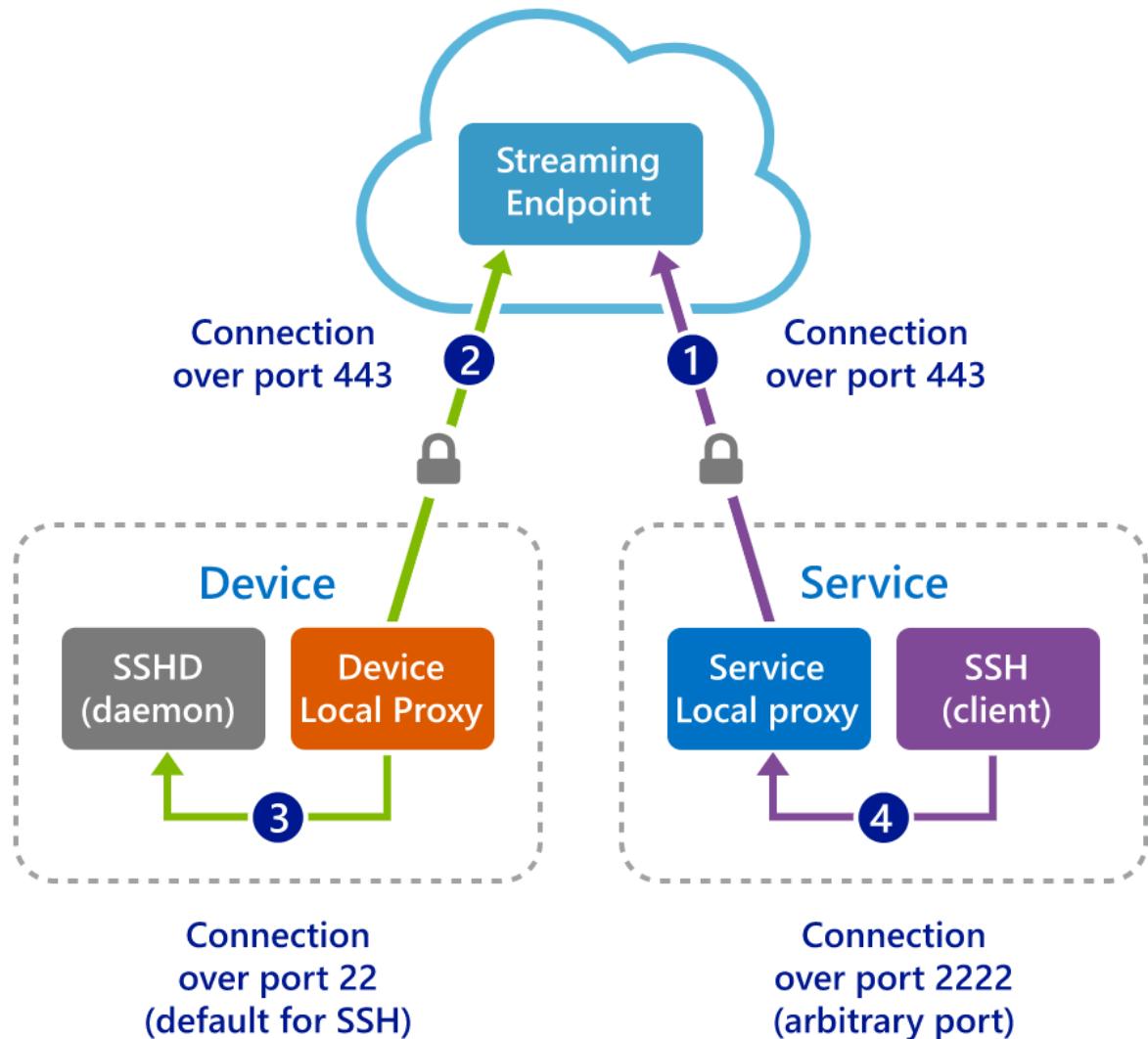
Microsoft Azure IoT Hub currently supports device streams as a [preview feature](#).

[IoT Hub device streams](#) allow service and device applications to communicate in a secure and firewall-friendly manner. This quickstart guide involves two C# applications that enable client-server application traffic (such as Secure Shell [SSH] and Remote Desktop Protocol [RDP]) to be sent over a device stream that's established through an IoT hub. For an overview of the setup, see [Local proxy application sample for SSH or RDP](#).

This article first describes the setup for SSH (using port 22) and then describes how to modify the setup's port for RDP. Because device streams are application- and protocol-agnostic, the same sample can be modified to accommodate other types of application traffic. This modification usually involves only changing the communication port to the one that's used by the intended application.

How it works

The following figure illustrates how the device-local and service-local proxy applications in this sample enable end-to-end connectivity between the SSH client and SSH daemon processes. Here, we assume that the daemon is running on the same device as the device-local proxy application.



1. The service-local proxy application connects to the IoT hub and initiates a device stream to the target device.
2. The device-local proxy application completes the stream initiation handshake and establishes an end-to-end streaming tunnel through the IoT hub's streaming endpoint to the service side.
3. The device-local proxy application connects to the SSH daemon that's listening on port 22 on the device. This setting is configurable, as described in the "Run the device-local proxy application" section.
4. The service-local proxy application waits for new SSH connections from a user by listening on a designated port, which in this case is port 2222. This setting is configurable, as described in the "Run the service-local proxy application" section. When the user connects via the SSH client, the tunnel enables SSH application traffic to be transferred between the SSH client and server application.

NOTE

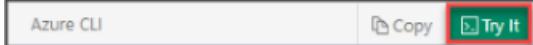
SSH traffic that's sent over a device stream is tunneled through the IoT hub's streaming endpoint rather than sent directly between service and device. For more information, see the [benefits of using IoT Hub device streams](#).

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell

preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|--|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal. |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

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

Prerequisites

- The preview of device streams is currently supported only for IoT hubs that are created in the following regions:
 - Central US
 - Central US EUAP
 - Southeast Asia
 - North Europe
- The two sample applications that you run in this quickstart are written in C#. You need the .NET Core SDK 2.1.0 or later on your development machine.

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

- Verify the current version of C# on your development machine by using the following command:

```
dotnet --version
```

- Run the following command to add the Azure IoT Extension for Azure CLI to your Cloud Shell instance. The IOT Extension adds IoT Hub, IoT Edge, and IoT Device Provisioning Service (DPS)-specific commands to the Azure CLI.

```
az extension add --name azure-iot
```

```
az extension add --name azure-iot
```

NOTE

This article uses the newest version of the Azure IoT extension, called `azure-iot`. The legacy version is called `azure-cli-iot-ext`. You should only have one version installed at a time. You can use the command `az extension list` to validate the currently installed extensions.

Use `az extension remove --name azure-cli-iot-ext` to remove the legacy version of the extension.

Use `az extension add --name azure-iot` to add the new version of the extension.

To see what extensions you have installed, use `az extension list`.

- Download the [Azure IoT C# samples](#), and extract the ZIP archive.
- A valid user account and credential on the device (Windows or Linux) used to authenticate the user.

Create an IoT hub

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

1. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.
 - **IoT Hub Name:** Enter a name for your hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

Home > New > IoT hub

IoT hub

Microsoft

X

Basics Size and scale Tags Review + create

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

Project details

Choose the subscription you'll use to manage deployments and costs. Use resource groups like folders to help you organize and manage resources.

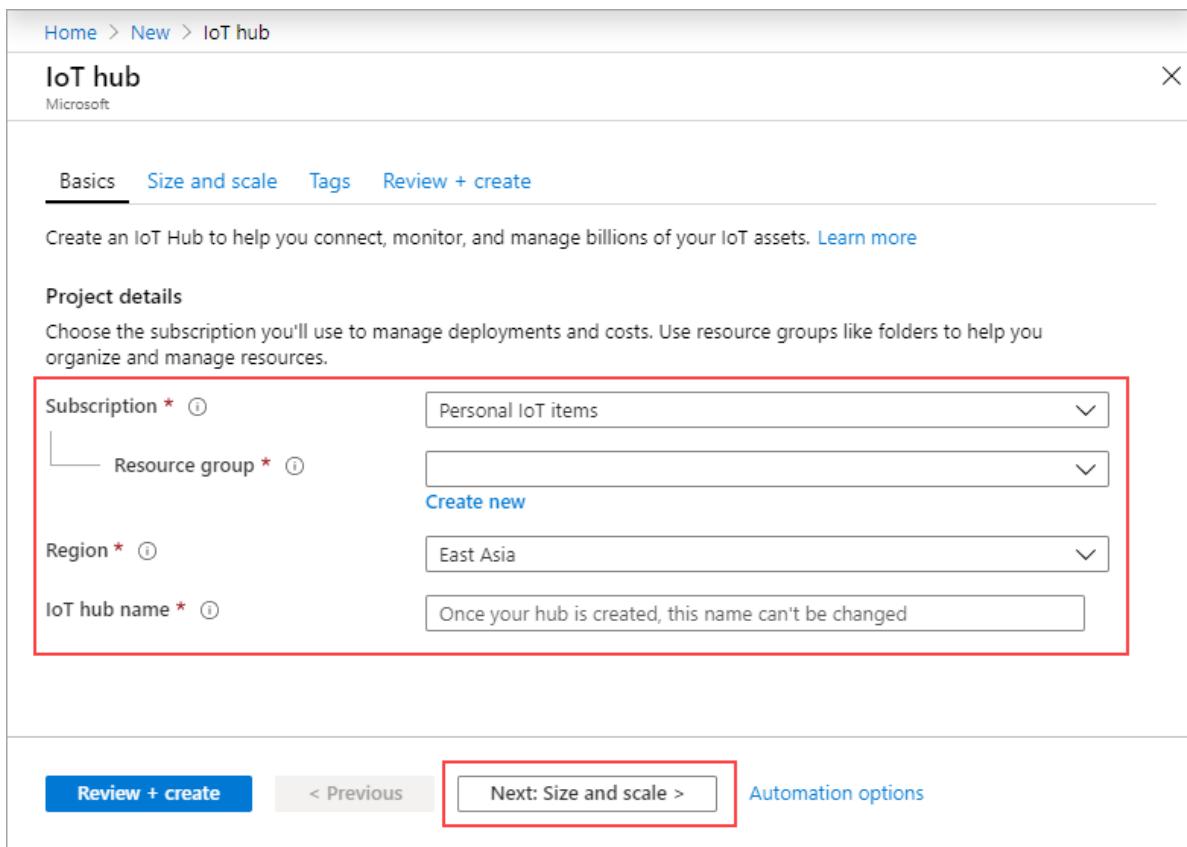
Subscription * ⓘ Personal IoT items

Resource group * ⓘ Create new

Region * ⓘ East Asia

IoT hub name * ⓘ Once your hub is created, this name can't be changed

Review + create < Previous **Next: Size and scale >** Automation options



5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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
 Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center [On](#) Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.

If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure](#)

resources.

The screenshot shows the 'Tags' step in the IoT hub creation wizard. It displays a table of resource tags. One tag is defined: 'department' with value 'accounting'. There is also an empty row for another tag. Navigation buttons at the bottom include 'Review + create' (highlighted), '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

| Name | Value | Resource | Actions |
|------------|------------|----------|-----------------------------|
| department | accounting | IoT Hub | ... Delete |
| | | IoT Hub | |

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' step in the IoT hub creation wizard. It lists the configuration details: Subscription (Personal testing), Resource group (iot-hubs), Region (West US 2), IoT hub name (you-hub-name). Under 'Size and scale', it shows Pricing tier (S1), Number of units (1), Messages per day (400,000), Cost per month (25.00 USD), and Azure Security Center integration (0.001 USD per device per month). Under 'Tags', it shows 'department' with value 'accounting'. A red box highlights the 'Create' button at the bottom left, and another red box highlights the 'Review + create' tab in the top navigation bar.

8. Select **Create** to create your new 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 Azure Cloud Shell

to register a simulated device.

1. To create the device identity, run the following command in Cloud Shell:

NOTE

- Replace the *YourIoTHubName* placeholder with the name you chose for your IoT hub.
- For the name of the device you're registering, it's recommended to use *MyDevice* as shown. If you choose a different name for your device, use that name throughout this article, and update the device name in the sample applications before you run them.

```
az iot hub device-identity create --hub-name {YourIoTHubName} --device-id MyDevice
```

2. To get the *device connection string* for the device that you just registered, run the following commands in Cloud Shell:

NOTE

Replace the *YourIoTHubName* placeholder with the name you chose for your IoT hub.

```
az iot hub device-identity show-connection-string --hub-name {YourIoTHubName} --device-id MyDevice --output table
```

Note the returned device connection string for later use in this quickstart. It looks like the following example:

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

3. To connect to your IoT hub and establish a device stream, you also need the *service connection string* from your IoT hub to enable the service-side application. The following command retrieves this value for your IoT hub:

NOTE

Replace the *YourIoTHubName* placeholder with the name you chose for your IoT hub.

```
az iot hub show-connection-string --policy-name service --name {YourIoTHubName} --output table
```

Note the returned service connection string for later use in this quickstart. It looks like the following example:

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

SSH to a device via device streams

In this section, you establish an end-to-end stream to tunnel SSH traffic.

Run the device-local proxy application

In a local terminal window, navigate to the `device-streams-proxy/device` directory in your unzipped project folder. Keep the following information handy:

| ARGUMENT NAME | ARGUMENT VALUE |
|------------------------|--|
| DeviceConnectionString | The device connection string of the device that you created earlier. |
| targetServiceHostName | The IP address where the SSH server listens. The address would be <code>localhost</code> if it were the same IP where the device-local proxy application is running. |
| targetServicePort | The port that's used by your application protocol (for SSH, by default, this would be port 22). |

Compile and run the code with the following commands:

```
cd ./iot-hub/Quickstarts/device-streams-proxy/device/

# Build the application
dotnet build

# Run the application
# In Linux or macOS
dotnet run ${DeviceConnectionString} localhost 22

# In Windows
dotnet run {DeviceConnectionString} localhost 22
```

Run the service-local proxy application

In another local terminal window, navigate to `iot-hub/quickstarts/device-streams-proxy/service` in your unzipped project folder. Keep the following information handy:

| PARAMETER NAME | PARAMETER VALUE |
|-------------------------|--|
| ServiceConnectionString | The service connection string of your IoT Hub. |
| MyDevice | The identifier of the device you created earlier. |
| localPortNumber | A local port that your SSH client will connect to. We use port 2222 in this sample, but you could use other arbitrary numbers. |

Compile and run the code with the following commands:

```
cd ./iot-hub/Quickstarts/device-streams-proxy/service/

# Build the application
dotnet build

# Run the application
# In Linux or macOS
dotnet run ${ServiceConnectionString} MyDevice 2222

# In Windows
dotnet run {ServiceConnectionString} MyDevice 2222
```

Run the SSH client

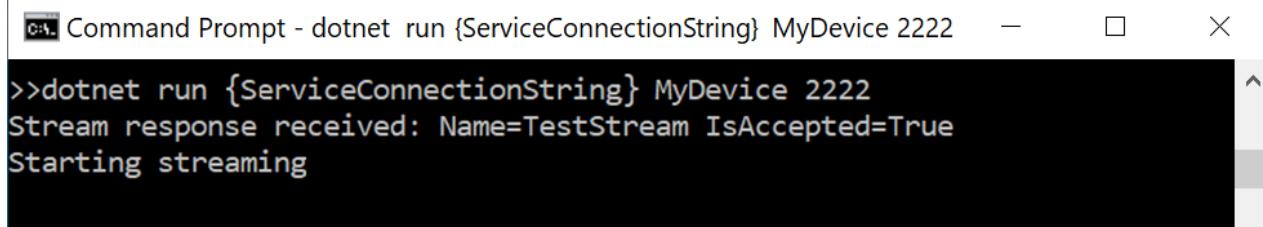
Now use your SSH client application and connect to service-local proxy application on port 2222 (instead of the

SSH daemon directly).

```
ssh {username}@localhost -p 2222
```

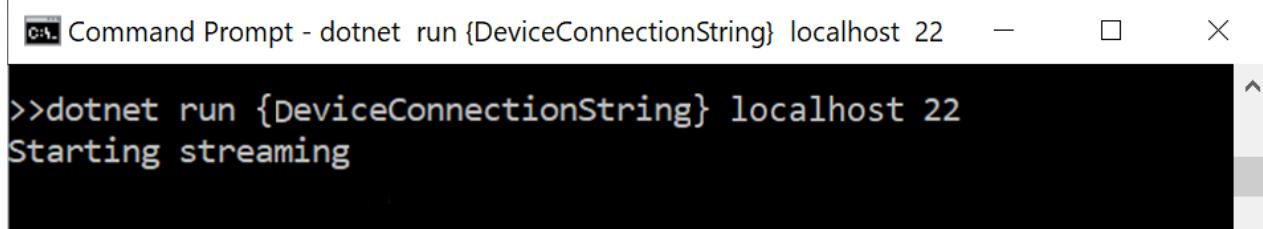
At this point, the SSH sign-in window prompts you to enter your credentials.

Console output on the service side (the service-local proxy application listens on port 2222):



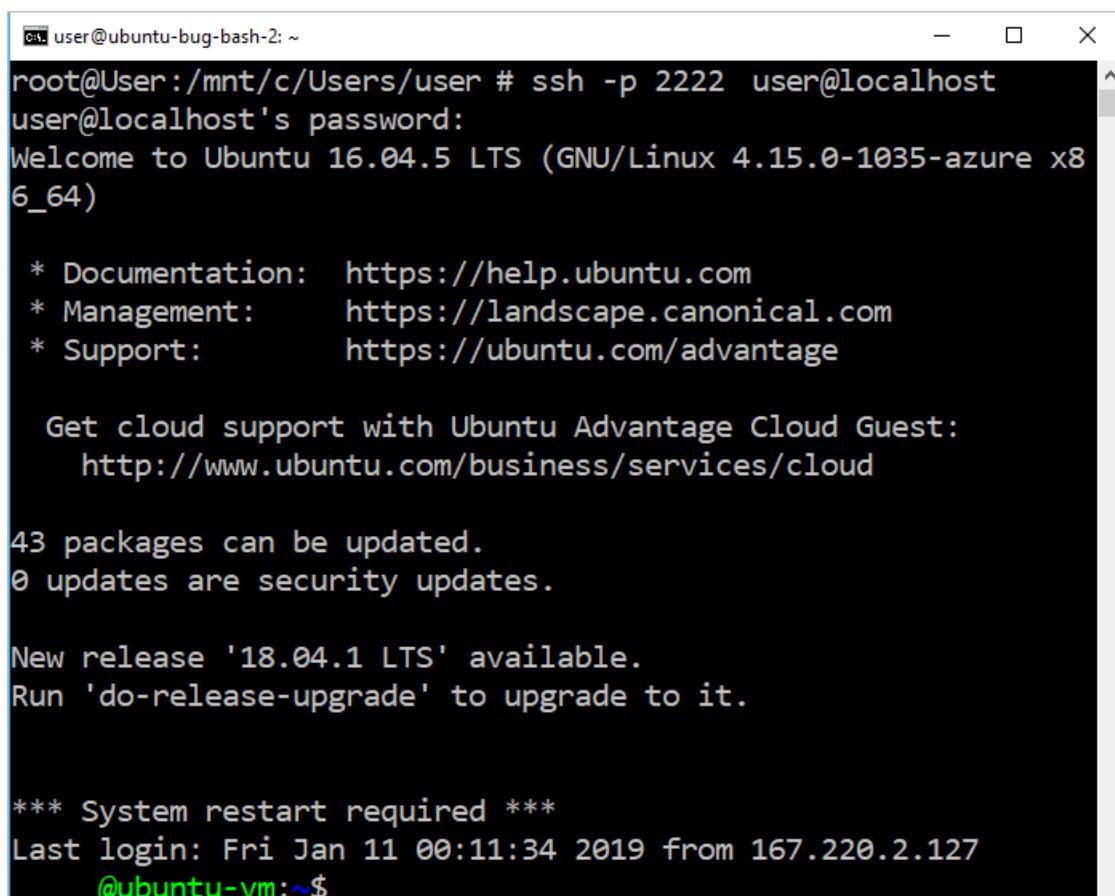
```
Command Prompt - dotnet run {ServiceConnectionString} MyDevice 2222
>>dotnet run {ServiceConnectionString} MyDevice 2222
Stream response received: Name=TestStream IsAccepted=True
Starting streaming
```

Console output on the device-local proxy application, which connects to the SSH daemon at *IP_address:22*:



```
Command Prompt - dotnet run {DeviceConnectionString} localhost 22
>>dotnet run {DeviceConnectionString} localhost 22
Starting streaming
```

Console output of the SSH client application. The SSH client communicates to the SSH daemon by connecting to port 22, which the service-local proxy application is listening on:



```
user@ubuntu-bug-bash-2: ~
root@User:/mnt/c/Users/user # ssh -p 2222 user@localhost
user@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 similar to the setup for SSH (described above). You use the RDP destination IP and port 3389

instead and use the RDP client (instead of the SSH client).

Run the device-local proxy application (RDP)

In a local terminal window, navigate to the `device-streams-proxy/device` directory in your unzipped project folder. Keep the following information handy:

| ARGUMENT NAME | ARGUMENT VALUE |
|-------------------------------------|---|
| <code>DeviceConnectionString</code> | The device connection string of the device that you created earlier. |
| <code>targetServiceHostName</code> | The hostname or IP address where RDP server runs. The address would be <code>localhost</code> if it were the same IP where the device-local proxy application is running. |
| <code>targetServicePort</code> | The port used by your application protocol (for RDP, by default, this would be port 3389). |

Compile and run the code with the following commands:

```
cd ./iot-hub/Quickstarts/device-streams-proxy/device

# Run the application
# In Linux or macOS
dotnet run ${DeviceConnectionString} localhost 3389

# In Windows
dotnet run {DeviceConnectionString} localhost 3389
```

Run the service-local proxy application (RDP)

In another local terminal window, navigate to `device-streams-proxy/service` in your unzipped project folder. Keep the following information handy:

| PARAMETER NAME | PARAMETER VALUE |
|--------------------------------------|---|
| <code>ServiceConnectionString</code> | The service connection string of your IoT Hub. |
| <code>MyDevice</code> | The identifier of the device you created earlier. |
| <code>localPortNumber</code> | A local port that 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 with the following commands:

```
cd ./iot-hub/Quickstarts/device-streams-proxy/service/

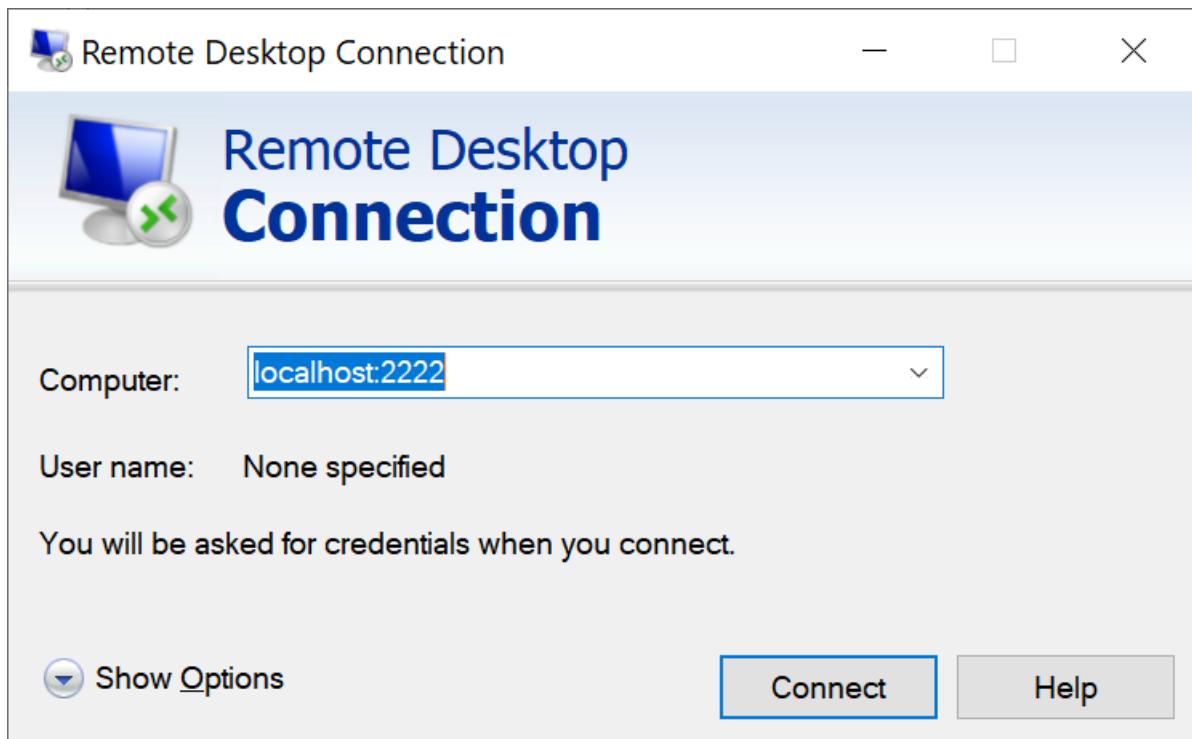
# Build the application
dotnet build

# Run the application
# In Linux or macOS
dotnet run ${ServiceConnectionString} MyDevice 2222

# In Windows
dotnet run {ServiceConnectionString} MyDevice 2222
```

Run RDP client

Now use your RDP client application and connect to the service-local proxy application on port 2222 (this was an arbitrary available port that you chose earlier).



Clean up resources

If you plan to continue to the next recommended article, you can keep and reuse the resources you've already created.

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

IMPORTANT

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

To delete a resource group by name:

1. Sign in to the [Azure portal](#), and then select **Resource groups**.
2. In the **Filter by name** box, enter the name of the resource group that contains your IoT hub.
3. In the result list, to the right of your resource group, select the ellipsis (...), and then select **Delete resource group**.

The screenshot shows the Azure Resource Groups page. On the left, there's a sidebar with options like 'Create a resource', 'Home', 'Dashboard', 'All services', 'FAVORITES' (which includes 'All resources', 'Resource groups', 'App Services', and 'SQL databases'), and 'Resource groups' (which 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 'All locations'. A red box highlights the 'Delete resource group' button in the '... More options' menu for the selected resource group.

4. To confirm the deletion of the resource group, reenter the resource group name, and then select **Delete**.
After a few moments, the resource group and all its contained resources are deleted.

Next steps

In this quickstart, you set up an IoT hub, registered a device, deployed device-local and service-local proxy applications to establish a device stream through the IoT hub, and used the proxy applications to tunnel SSH or RDP traffic. The same paradigm can accommodate other client-server protocols, where the server runs on the device (for example, the SSH daemon).

To learn more about device streams, see:

[Device streams overview](#)

Quickstart: Enable SSH and RDP over an IoT Hub device stream by using a Node.js proxy application (preview)

3/6/2020 • 9 minutes to read • [Edit Online](#)

In this quickstart, you enable Secure Shell (SSH) and Remote Desktop Protocol (RDP) traffic to be sent to the device over a device stream. Azure IoT Hub device streams allow service and device applications to communicate in a secure and firewall-friendly manner. This quickstart describes the execution of a Node.js proxy application that's running on the service side. During public preview, the Node.js SDK supports device streams on the service side only. As a result, this quickstart covers instructions to run only the service-local proxy application.

Prerequisites

- Completion of [Enable SSH and RDP over IoT Hub device streams by using a C proxy application](#) or [Enable SSH and RDP over IoT Hub device streams by using a C# proxy application](#).
- An Azure account with an active subscription. [Create one for free](#).
- [Node.js 10+](#).
- [A sample Node.js project](#).

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

```
node --version
```

Microsoft Azure IoT Hub currently supports device streams as a [preview feature](#).

IMPORTANT

The preview of device streams is currently only supported for IoT Hubs created in the following regions:

- Central US
- Central US EUAP
- North Europe
- Southeast Asia

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|--|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal. |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the Copy button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

Add Azure IoT Extension

Add the Azure IoT Extension for Azure CLI to your Cloud Shell instance by running the following command. The IoT Extension adds IoT Hub, IoT Edge, and IoT Device Provisioning Service (DPS)-specific commands to the Azure CLI.

```
az extension add --name azure-iot
```

NOTE

This article uses the newest version of the Azure IoT extension, called `azure-iot`. The legacy version is called `azure-cli-iot-ext`. You should only have one version installed at a time. You can use the command `az extension list` to validate the currently installed extensions.

Use `az extension remove --name azure-cli-iot-ext` to remove the legacy version of the extension.

Use `az extension add --name azure-iot` to add the new version of the extension.

To see what extensions you have installed, use `az extension list`.

Create an IoT hub

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

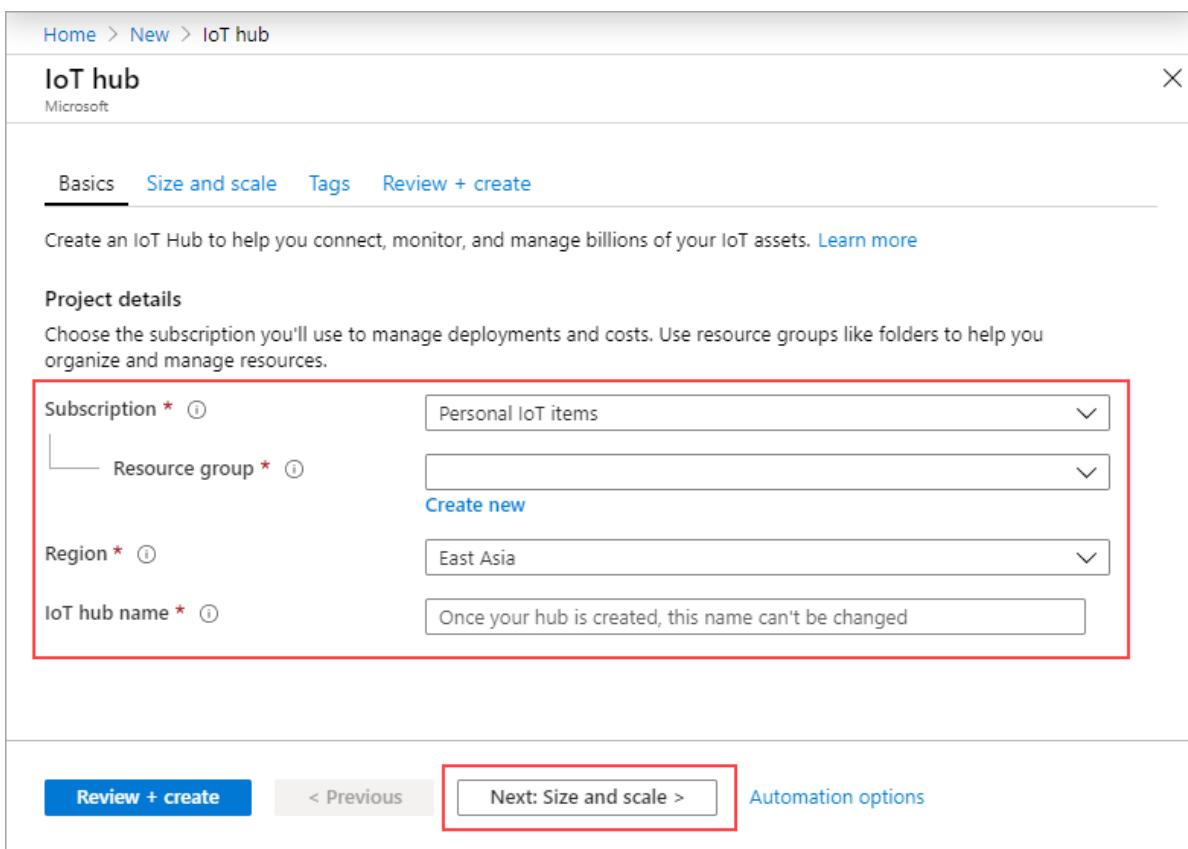
This section describes how to create an IoT hub using the [Azure portal](#).

1. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter **IoT Hub** in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.

- **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
- **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.
- **IoT Hub Name:** Enter a name for your hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.



Home > New > IoT hub

IoT hub
Microsoft

Basics [Size and scale](#) [Tags](#) [Review + create](#)

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

Project details

Choose the subscription you'll use to manage deployments and costs. Use resource groups like folders to help you organize and manage resources.

| | |
|------------------|--|
| Subscription * | Personal IoT items |
| Resource group * | <input type="text"/> Create new |
| Region * | East Asia |
| IoT hub name * | Once your hub is created, this name can't be changed |

[Review + create](#) [< Previous](#) [Next: Size and scale >](#) [Automation options](#)

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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
 Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center On
 Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.

If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure](#)

resources.

The screenshot shows the 'Tags' tab of the IoT hub configuration page. It displays a table with one row of data:

| Name | Value | Resource |
|------------|------------|----------|
| department | accounting | IoT Hub |
| | | IoT Hub |

Below the table are navigation buttons: 'Review + create' (highlighted in blue), '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' step of the IoT hub creation process. It lists the following configuration details:

| Setting | Value |
|----------------------------|--------------------------------|
| Subscription | Personal testing |
| Resource group | iot-hubs |
| Region | West US 2 |
| IoT hub name | you-hub-name |
| 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 |
| Azure Security Center | 0.001 USD per device per month |
| Tags | |
| department | accounting |

At the bottom are buttons: 'Create' (highlighted in red), '< Previous: Tags', 'Next >', and 'Automation options'.

8. Select **Create** to create your new hub. Creating the hub takes a few minutes.

Register a device

If you completed [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 section, you use Azure Cloud Shell to register a simulated device.

1. To create the device identity, run the following command in Cloud Shell:

NOTE

- Replace the *YourIoTHubName* placeholder with the name you chose for your IoT hub.
- For the name of the device you're registering, it's recommended to use *MyDevice* as shown. If you choose a different name for your device, use that name throughout this article, and update the device name in the sample applications before you run them.

```
az iot hub device-identity create --hub-name {YourIoTHubName} --device-id MyDevice
```

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

NOTE

Replace the *YourIoTHubName* placeholder with the name you chose for your IoT hub.

```
az iot hub show-connection-string --policy-name service --name {YourIoTHubName} --output table
```

Note the returned service connection string for later use in this quickstart. It looks like the following example:

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

SSH to a device via device streams

In this section, you establish an end-to-end stream to tunnel SSH traffic.

Run the device-local proxy application

As mentioned earlier, the IoT Hub Node.js SDK supports device streams on the service side only. For the device-local application, use a device proxy application that's available in one of the following quickstarts:

- [Enable SSH and RDP over IoT Hub device streams by using a C proxy application](#)
- [Enable SSH and RDP over IoT Hub device streams by using a C# proxy application](#)

Before you proceed to the next step, ensure that the device-local proxy application is running. For an overview of the setup, see [Local Proxy Sample](#).

Run the service-local proxy application

This article describes the setup for SSH (by using port 22) and then describes how to modify the setup for RDP (which uses port 3389). Because device streams are application- and protocol-agnostic, you can modify the same sample to accommodate other types of client-server application traffic, usually by modifying the communication port.

With the device-local proxy application running, run the service-local proxy application that's written in Node.js by doing the following in a local terminal window:

1. For environment variables, provide your service credentials, the target device ID where the SSH daemon runs, and the port number for the proxy that's running on the device.

```

# In Linux
export IOTHUB_CONNECTION_STRING="{ServiceConnectionString}"
export STREAMING_TARGET_DEVICE="MyDevice"
export PROXY_PORT=2222

# In Windows
SET IOTHUB_CONNECTION_STRING={ServiceConnectionString}
SET STREAMING_TARGET_DEVICE=MyDevice
SET PROXY_PORT=2222

```

Change the ServiceConnectionString placeholder to match your service connection string, and MyDevice to match your device ID if you gave yours a different name.

2. Navigate to the `Quickstarts/device-streams-service` directory in your unzipped project folder. Use the following code to run the service-local proxy application:

```

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

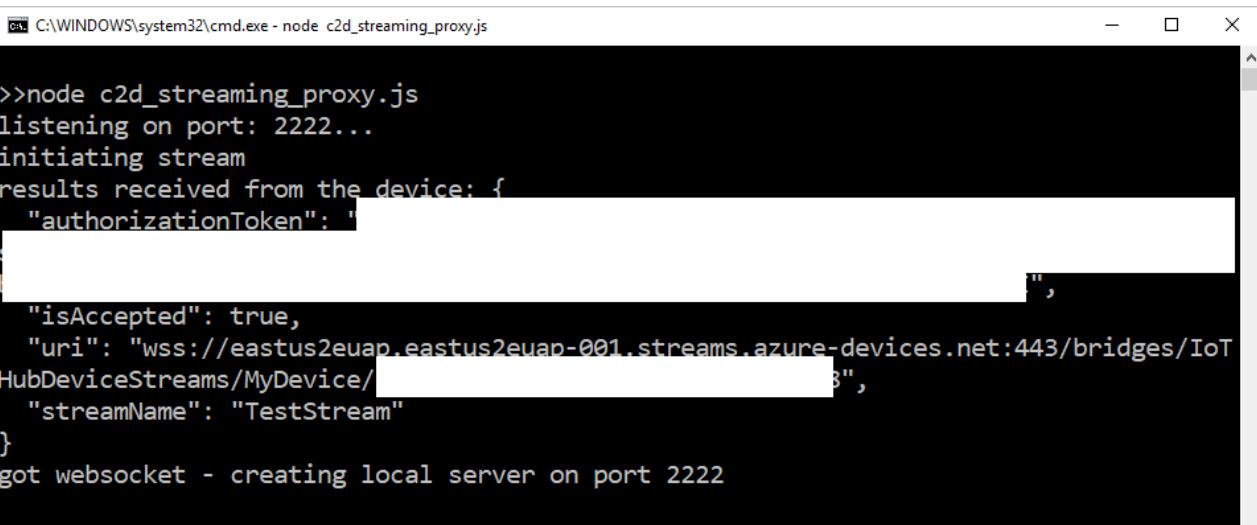
# Run the service-local proxy application
node proxy.js

```

SSH to your device via device streams

In Linux, run SSH by using `ssh $USER@localhost -p 2222` on a terminal. In Windows, use your favorite SSH client (for example, PuTTY).

Console output on the service-local after SSH session is established (the service-local proxy application listens on port 2222):



```

C:\WINDOWS\system32\cmd.exe - node c2d_streaming_proxy.js
>>node c2d_streaming_proxy.js
listening on port: 2222...
initiating stream
results received from the device: {
  "authorizationToken": "XXXXXXXXXXXXXX",
  "streamName": "TestStream"
}
got websocket - creating local server on port 2222

```

Console output of the SSH client application (SSH client communicates to SSH daemon by connecting to port 22, where the service-local proxy application is listening):

```
user@ubuntu-bug-bash-2: ~
root@User:/mnt/c/Users/user # ssh -p 2222 user@localhost
user@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.

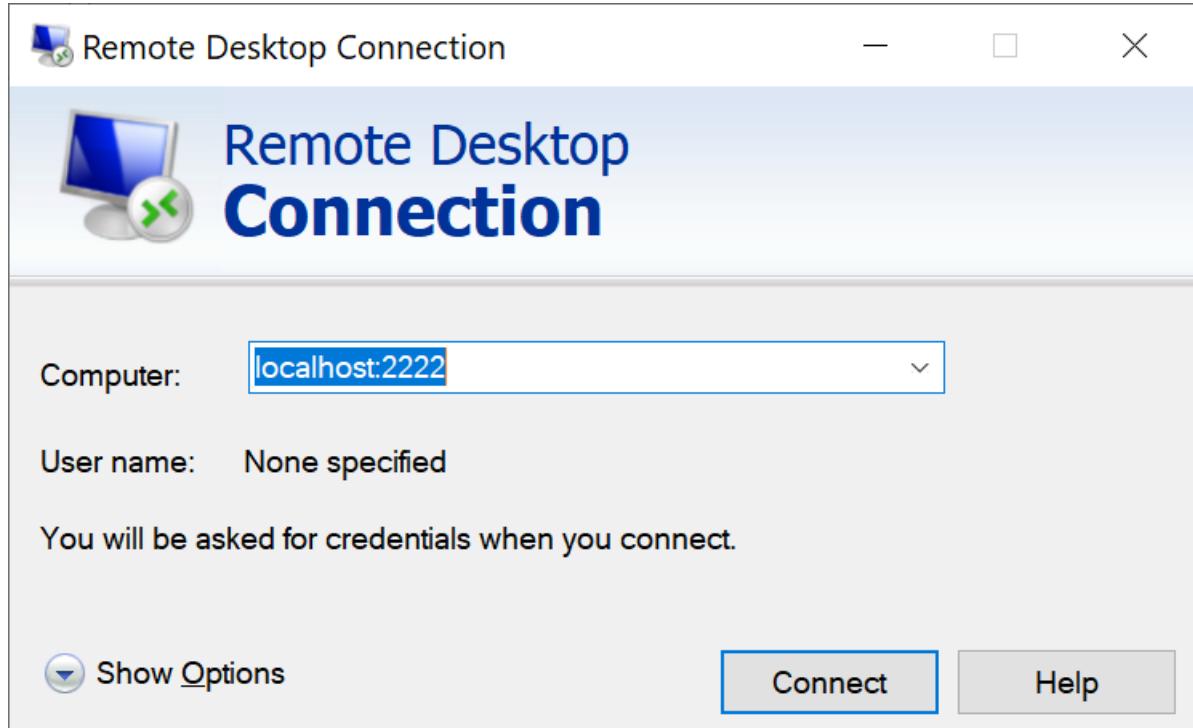
*** System restart required ***
Last login: Fri Jan 11 00:11:34 2019 from 167.220.2.127
@ubuntu-vm:~$
```

RDP to your device via device streams

Now use your RDP client application and connect to the service proxy on port 2222, an arbitrary port that 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 plan to continue to the next recommended article, you can keep and reuse the resources you've already created.

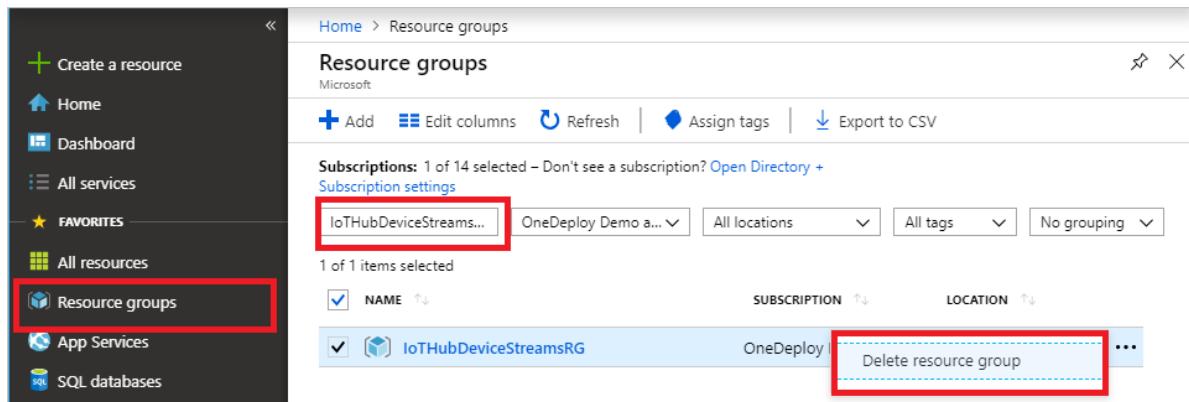
Otherwise, to avoid charges, you can delete the Azure resources that you created in this article.

IMPORTANT

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

To delete a resource group by name:

1. Sign in to the [Azure portal](#), and then select **Resource groups**.
2. In the **Filter by name** box, enter the name of the resource group that contains your IoT hub.
3. In the result list, to the right of your resource group, select the ellipsis (...), and then select **Delete resource group**.



4. To confirm the deletion of the resource group, reenter the resource group name, and then select **Delete**. After a few moments, the resource group and all its contained resources are deleted.

Next steps

In this quickstart, you set up an IoT hub, registered a device, and deployed a service proxy application to enable RDP and SSH on an IoT device. The RDP and SSH traffic will be tunneled via a device stream through the IoT hub. This process eliminates the need for direct connectivity to the device.

To learn more about device streams, see:

[Device streams overview](#)

Quickstart: Enable SSH and RDP over an IoT Hub device stream by using a C proxy application (preview)

5/21/2020 • 11 minutes to read • [Edit Online](#)

Azure IoT Hub currently supports device streams as a [preview feature](#).

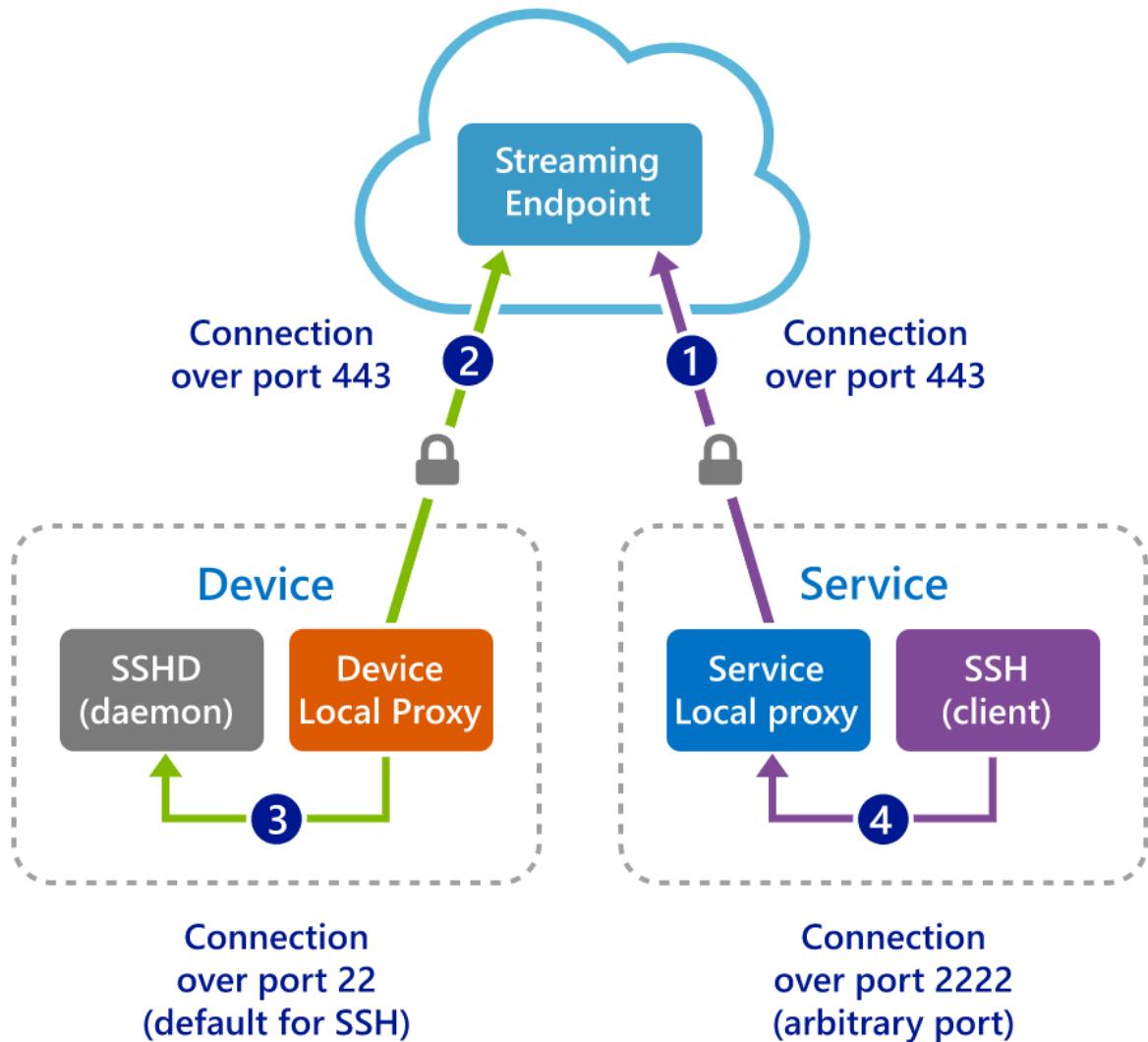
[IoT Hub device streams](#) allow service and device applications to communicate in a secure and firewall-friendly manner. For an overview of the setup, see [the Local Proxy Sample page](#).

This quickstart describes the setup for tunneling Secure Shell (SSH) traffic (using port 22) through device streams. The setup for Remote Desktop Protocol (RDP) traffic is similar and requires a simple configuration change. Because device streams are application- and protocol-agnostic, you can modify this quickstart to accommodate other types of application traffic.

How it works

The following figure illustrates how the device- and service-local proxy programs enable end-to-end connectivity between the SSH client and SSH daemon processes. During public preview, the C SDK supports device streams on the device side only. As a result, this quickstart covers instructions to run only the device-local proxy application. To build and run the accompanying service-side application, follow the instructions in one of the following quickstarts:

- [SSH/RDP over IoT Hub device streams using C# proxy](#)
- [SSH/RDP over IoT Hub device streams using NodeJS proxy](#).



1. The service-local proxy connects to the IoT hub and starts a device stream to the target device.
2. The device-local proxy completes the stream initiation handshake and establishes an end-to-end streaming tunnel through the IoT hub's streaming endpoint to the service side.
3. The device-local proxy connects to the SSH daemon that's listening on port 22 on the device. This setting is configurable, as described in the "Run the device-local proxy application" section.
4. The service-local proxy waits for new SSH connections from a user by listening on a designated port, which in this case is port 2222. This setting is configurable, as described in the "Run the device-local proxy application" section. When the user connects via SSH client, the tunnel enables SSH application traffic to be transferred between the SSH client and server programs.

NOTE

SSH traffic that's sent over a device stream is tunneled through the IoT hub's streaming endpoint rather than sent directly between service and device. For more information, see the [benefits of using IoT Hub device streams](#). Furthermore, the figure illustrates the SSH daemon that's running on the same device (or machine) as the device-local proxy. In this quickstart, providing the SSH daemon IP address allows the device-local proxy and the daemon to run on different machines as well.

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell

preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|--|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal . |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

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

Prerequisites

- The preview of device streams is currently supported only for IoT hubs that are created in the following regions:
 - Central US
 - Central US EUAP
 - North Europe
 - Southeast Asia
- Install [Visual Studio 2019](#) with the [Desktop development with C++](#) workload enabled.
- Install the latest version of [Git](#).
- Run the following command to add the Azure IoT Extension for Azure CLI to your Cloud Shell instance. The IOT Extension adds IoT Hub, IoT Edge, and IoT Device Provisioning Service (DPS)-specific commands to the Azure CLI.

```
az extension add --name azure-iot
```

NOTE

This article uses the newest version of the Azure IoT extension, called `azure-iot`. The legacy version is called `azure-cli-iot-ext`. You should only have one version installed at a time. You can use the command `az extension list` to validate the currently installed extensions.

Use `az extension remove --name azure-cli-iot-ext` to remove the legacy version of the extension.

Use `az extension add --name azure-iot` to add the new version of the extension.

To see what extensions you have installed, use `az extension list`.

Prepare the development environment

For this quickstart, you use the [Azure IoT device SDK for C](#). You prepare a development environment used to clone and build the [Azure IoT C SDK](#) from GitHub. The SDK on GitHub includes the sample code that's used in this quickstart.

1. Download the [CMake build system](#).

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

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

```
git clone -b public-preview https://github.com/Azure/azure-iot-sdk-c.git
cd azure-iot-sdk-c
git submodule update --init
```

This operation should take a few minutes.

3. Create a `cmake` subdirectory in the root directory of the git repository, and navigate to that folder. Run the following commands from the `azure-iot-sdk-c` directory:

```
mkdir cmake
cd cmake
```

4. Run the following commands from the `cmake` directory to build a version of the SDK that's specific to your development client platform.

- In Linux:

```
cmake ..
make -j
```

- In Windows, run the following commands in Developer Command Prompt for Visual Studio 2015 or 2017. A Visual Studio solution for the simulated device will be generated in the `cmake` directory.

```
rem For VS2015
cmake .. -G "Visual Studio 14 2015"

rem Or for VS2017
cmake .. -G "Visual Studio 15 2017"

rem Or for VS2019
cmake .. -G "Visual Studio 16 2019"

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. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.
 - **IoT Hub Name:** Enter a name for your hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

Home > New > IoT hub

IoT hub

Microsoft

X

Basics Size and scale Tags Review + create

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

Project details

Choose the subscription you'll use to manage deployments and costs. Use resource groups like folders to help you organize and manage resources.

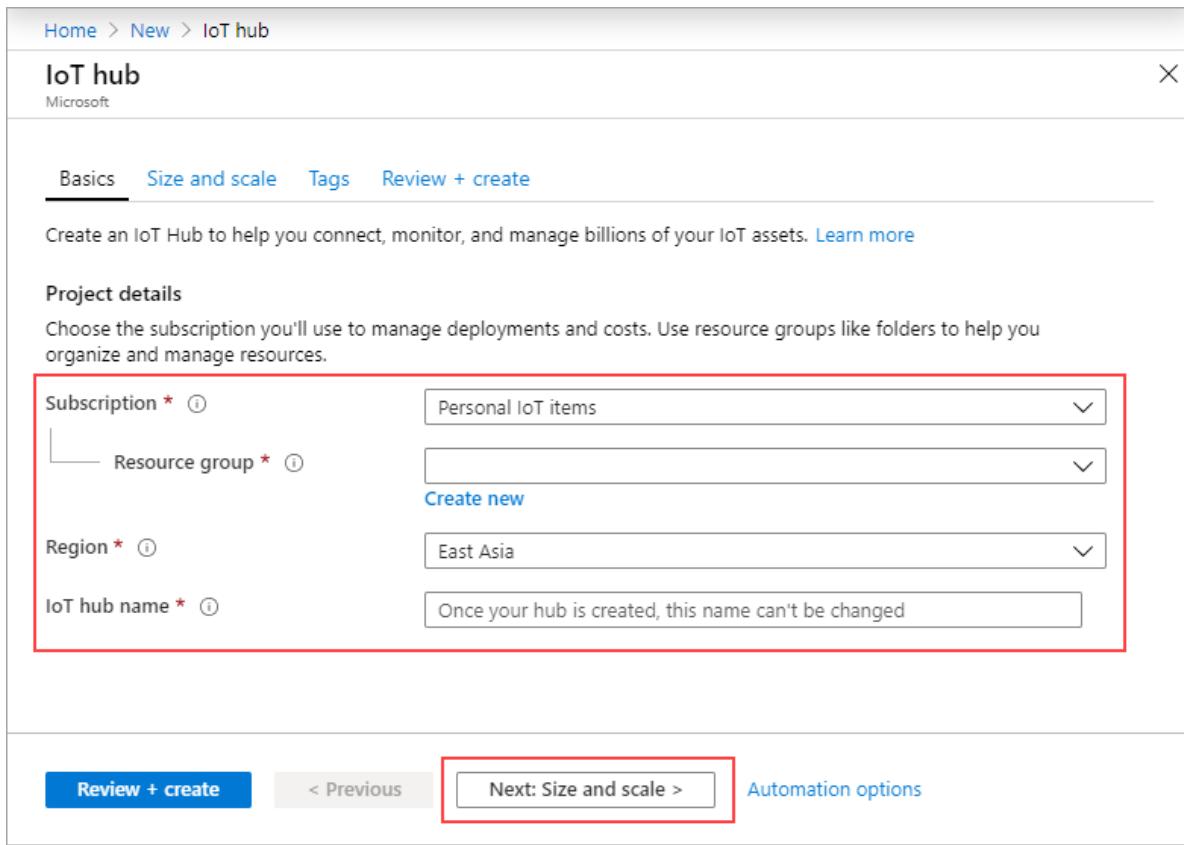
Subscription * ⓘ Personal IoT items

Resource group * ⓘ [Create new](#)

Region * ⓘ East Asia

IoT hub name * ⓘ Once your hub is created, this name can't be changed

Review + create < Previous **Next: Size and scale >** Automation options



5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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 Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center On Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.
If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure](#)

resources.

The screenshot shows the 'Tags' tab of the IoT hub configuration page. It displays a table with two rows of tag entries:

| Name | Value | Resource | Actions |
|------------|------------|----------|---------|
| department | accounting | IoT Hub | ... |
| | | IoT Hub | |

Below the table are navigation buttons: 'Review + create' (highlighted), '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' step of the IoT hub creation process. It lists the following configuration details:

| Setting | Value |
|----------------------------|--------------------------------|
| Subscription | Personal testing |
| Resource group | iot-hubs |
| Region | West US 2 |
| IoT hub name | you-hub-name |
| 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 |
| Azure Security Center | 0.001 USD per device per month |
| Tags | |
| department | accounting |

At the bottom are buttons: 'Create' (highlighted), '< Previous: Tags', 'Next >', and 'Automation options'.

8. Select **Create** to create your new 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 use Azure Cloud Shell

with the [IoT extension](#) to register a simulated device.

1. To create the device identity, run the following command in Cloud Shell:

NOTE

- Replace the *YourIoTHubName* placeholder with the name you chose for your IoT hub.
- For the name of the device you're registering, it's recommended to use *MyDevice* as shown. If you choose a different name for your device, use that name throughout this article, and update the device name in the sample applications before you run them.

```
az iot hub device-identity create --hub-name {YourIoTHubName} --device-id MyDevice
```

2. To get the *device connection string* for the device that you just registered, run the following commands in Cloud Shell:

NOTE

Replace the *YourIoTHubName* placeholder with the name you chose for your IoT hub.

```
az iot hub device-identity show-connection-string --hub-name {YourIoTHubName} --device-id MyDevice --output table
```

Note the returned device connection string for later use in this quickstart. It looks like the following example:

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

SSH to a device via device streams

In this section, you establish an end-to-end stream to tunnel SSH traffic.

Run the device-local proxy application

1. Edit the source file `iothub_client_c2d_streaming_proxy_sample.c` in the folder

`iothub_client/samples/iothub_client_c2d_streaming_proxy_sample`, and provide your device connection string, target device IP/hostname, and the SSH port 22:

```
/* Paste in your device connection string */
static const char* connectionString = "{ConnectionString}";
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.
```

2. Compile the sample:

```
# 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
```

3. Run the compiled program on the device:

```
# In Linux  
# Go to the 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 the 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 in the "How it works" section, establishing an end-to-end stream to tunnel SSH traffic requires a local proxy at each end (on both the service and the device sides). During public preview, the IoT Hub C SDK supports device streams on the device side only. To build and run the service-local proxy, follow the instructions in one of the following quickstarts:

- [SSH/RDP over IoT Hub device streams using C# proxy apps](#)
- [SSH/RDP over IoT Hub device streams using Node.js proxy apps](#)

Establish an SSH session

After both the device- and service-local proxies are running, use your SSH client program and connect to the service-local proxy on port 2222 (instead of the SSH daemon directly).

```
ssh {username}@localhost -p 2222
```

At this point, the SSH sign-in window prompts you to enter your credentials.

The following image shows the console output on the device-local proxy, which connects to the SSH daemon at `IP_address:22`:

```
user@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&$url=wss%3A%2F%2Feastus2euap.eastus2euap-001.streams.azure-devices.net%3A443%2Fbridges%2FIoTHubDeviceStreams%2FMyDevice%2Fa58c0fa0bb62409aa15518e609062f86&$auth=eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJleHAiOjE1NDcxNjYzODUsImp0aSI6Ij1kb19pODBsRXJEcS1KdlVpT3pENWdTN0NsbuQ0dGNkMnpTN2JZY1hsVnMiLCJpb3RodWIiOiJJb1RIdWJEZXZpY2VTdHJ1YW1zIn0._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)
```

The following image shows the console output of the SSH client program. The SSH client communicates to the SSH daemon by connecting to port 2222, which the service-local proxy is listening on:

```
user@ubuntu-vm: ~
root@User:/mnt/c/Users/user # ssh -p 2222 user@localhost
user@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 plan to continue to the next recommended article, you can keep and reuse the resources you've already created.

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

IMPORTANT

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

To delete a resource group by name:

1. Sign in to the [Azure portal](#), and then select **Resource groups**.
2. In the **Filter by name** box, enter the name of the resource group that contains your IoT hub.
3. In the result list, to the right of your resource group, select the ellipsis (...), and then select **Delete resource group**.

The screenshot shows the Azure Resource Groups blade. On the left, the navigation menu has 'Resource groups' selected. The main area displays a table of resource groups. A red box highlights the 'IoTHubDeviceStreams...' entry in the 'NAME' column. Another red box highlights the 'Delete resource group' button in the ellipsis column for this row.

4. To confirm the deletion of the resource group, reenter the resource group name, and then select **Delete**. After a few moments, the resource group and all its contained resources are deleted.

Next steps

In this quickstart, you 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.

To learn more about device streams, see:

[Device streams overview](#)

Tutorial: Use the Azure CLI and Azure portal to configure IoT Hub message routing

7/29/2020 • 13 minutes to read • [Edit Online](#)

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 Queues, Service Bus Topics, and Event Hubs. To configure custom message routing, you 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. If a message doesn't match any of the defined routing queries, it is routed to the default endpoint.

In this 2-part tutorial, you learn how to set up and use these custom routing queries with IoT Hub. You route messages from an IoT device to one of multiple endpoints, including blob storage and a Service Bus queue. Messages to the Service Bus queue are picked up by a Logic App and sent via e-mail. Messages that do not have custom message routing defined are sent to the default endpoint, then picked up by Azure Stream Analytics and viewed in a Power BI visualization.

To complete Parts 1 and 2 of this tutorial, you perform the following tasks:

Part I: Create resources, set up message routing

- Create the resources -- an IoT hub, a storage account, a Service Bus queue, and a simulated device. This can be done using the Azure portal, an Azure Resource Manager template, the Azure CLI, or Azure PowerShell.
- Configure the endpoints and message routes in IoT Hub for the storage account and Service Bus queue.

Part II: Send messages to the hub, view routed results

- 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

- For Part 1 of this tutorial:
 - You must have an Azure subscription. If you don't have an Azure subscription, create a [free account](#) before you begin.
- For Part 2 of this tutorial:
 - You must have completed Part 1 of this tutorial, and have the resources still available.
 - [Install Visual Studio](#).
 - Have access to a Power BI account to analyze the default endpoint's stream analytics. ([Try Power BI for free](#).)
 - Have a work or school account for sending notification e-mails.
 - Make sure that port 8883 is open in your firewall. The sample in this tutorial uses MQTT protocol,

which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|--|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal . |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

Create base resources

Before you can configure the message routing, you need to create an IoT hub, a storage account, and a Service Bus queue. These resources can be created using one of the four articles that are available for Part 1 of this tutorial: the Azure portal, an Azure Resource Manager template, the Azure CLI, or Azure PowerShell.

Use the same resource group and location for all of the resources. Then at the end, you can remove all of the resources in one step by deleting the resource group.

Below is a summary of the steps to be performed in the following sections:

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. (If creating a Resource Manager template, this is done after deploying the template.)

Use the Azure CLI to create the base resources

This tutorial uses the Azure CLI to create the base resources, then uses the [Azure portal](#) to show how to configure message routing and set up the virtual device for testing.

Copy and paste the script below into Cloud Shell and press Enter. It runs the script one line at a time. This will create the base resources for this tutorial, including the storage account, IoT Hub, Service Bus Namespace, and Service Bus queue.

There are several resource names that must be globally unique, such as the IoT Hub name and the storage account name. To make this easier, those resource names are appended with a random alphanumeric value called *randomValue*. The *randomValue* is generated once at the top of the script and appended to the resource names as needed throughout the script. If you don't want it to be random, you can set it to an empty string or to a specific value.

TIP

A tip about debugging: this script uses the continuation symbol (the backslash \) to make the script more readable. If you have a problem running the script, make sure your Cloud Shell session is running `bash` and that there are no spaces after any of the backslashes.

```
# This retrieves the subscription id of the account
#   in which you're logged in.
# This field is used to set up the routing queries.
subscriptionID=$(az account show --query id)

# Concatenate this number onto the resources that have to be globally unique.
# You can set this to "" or to a specific value if you don't want it to be random.
# This retrieves a random value.
randomValue=$RANDOM

# Set the values for the resource names that
#   don't have to be globally unique.
location=westus
resourceGroup=ContosoResources
iotHubConsumerGroup=ContosoConsumers
containerName=contosoresults

# 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 value to the end.
iotHubName=ContosoTestHub$randomValue
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 for the 'events' endpoint.
az iot hub consumer-group create --hub-name $iotHubName \
    --name ContosoEvents
```

```

az iot hub consumer-group create --hub-name $IoTHubName \
--name $IoTConsumerGroup

# The storage account name must be globally unique,
# so add a random value to the end.
storageAccountName=contosostorage$randomValue
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 "storage account key = " $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 value to the end.
sbNamespace=ContosoSBNamespace$randomValue
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 value to the end.
sbQueueName=ContosoSBQueue$randomValue
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

```

Now that the base resources are set up, you can configure the message routing in the [Azure portal](#).

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). In the next tutorial, you send messages to the IoT Hub and see them routed to the different destinations.

| VALUE | RESULT |
|-----------------|-------------------------|
| level="storage" | Write to Azure Storage. |

| VALUE | RESULT |
|------------------|---|
| 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. |

The first step is to set up the endpoint to which the data will be routed. The second step is to set up the message route that uses that endpoint. After setting up the routing, you can view the endpoints and message routes in the portal.

Route 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 routing is set up, messages where the **level** property is set to **storage** are written to a storage account automatically.

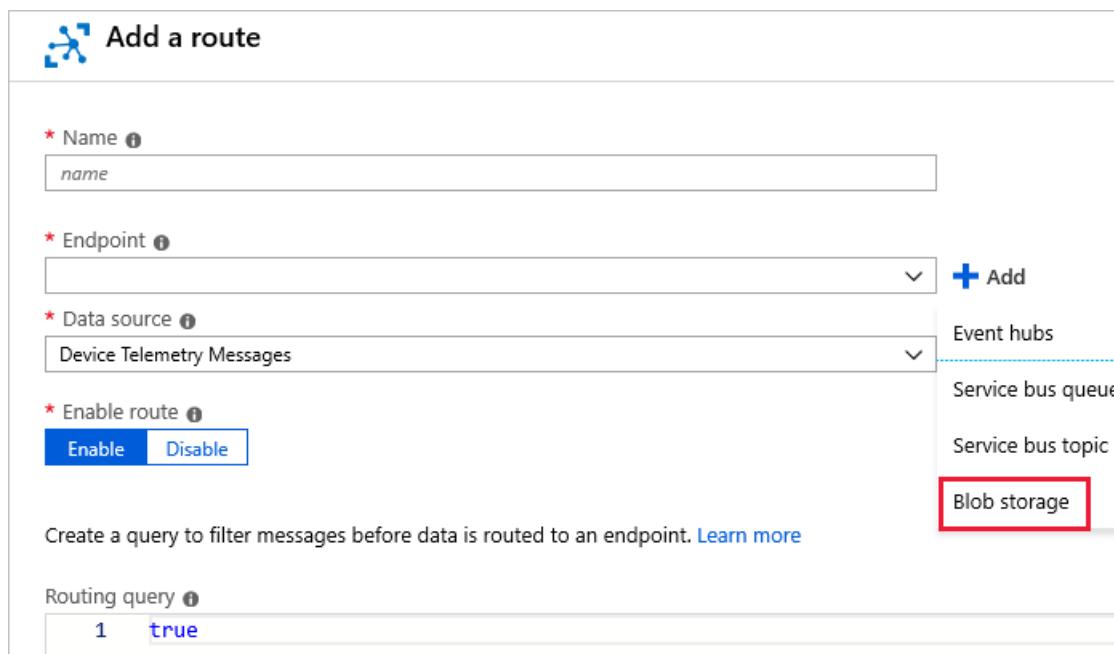
NOTE

The data can be written to blob storage in either the [Apache Avro](#) format, which is the default, or JSON (preview).

The capability to encode JSON format is in preview in all regions in which IoT Hub is available, except East US, West US and West Europe. The encoding format can be only set at the time the blob storage endpoint is configured. The format cannot be changed for an endpoint that has already been set up. When using JSON encoding, you must set the **contentType** to JSON and the **contentEncoding** to UTF-8 in the message system properties.

For more detailed information about using a blob storage endpoint, please see [guidance on routing to storage](#).

1. In the [Azure portal](#), select **Resource Groups**, then select your resource group. This tutorial uses **ContosoResources**.
2. Select the IoT hub under the list of resources. This tutorial uses **ContosoTestHub**.
3. Select **Message Routing**. In the **Message Routing** pane, select **+Add**. On the **Add a Route** pane, select **+Add** next to the **Endpoint** field to show the supported endpoints, as displayed in the following picture:



4. Select **Blob storage**. You see the **Add a storage endpoint** pane.

 Add a storage endpoint

Route your telemetry and device messages to Azure Storage.

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

Chunk size window

Encoding

File name format *

The format must contain {iothub}, {partition}, {YYYYY}, {MM}, {DD}, {HH} and {mm} in any order.
If multiple files are created within the same minute, the filename format would be ContosoTestHub23/0/2020/06/23/23/00-01.avro

5. Enter a name for the endpoint. This tutorial uses **ContosoStorageEndpoint**.
6. Select **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**. You return to the **Add a storage endpoint** pane and see the selections you made.
7. Set the encoding to AVRO or JSON. For the purpose of this tutorial, use the defaults for the rest of the fields. This field will be greyed out if the region selected does not support JSON encoding.,

NOTE

You can set the format of the blob name using the **Blob file name format**. The default is `{iothub}/{partition}/{YYYYY}/{MM}/{DD}/{HH}/{mm}`. The format must contain {iothub}, {partition}, {YYYYY}, {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.

8. Select **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 **ContosoStorageRoute**.

Endpoint: This shows the endpoint you just set up.

Data source: Select **Device Telemetry Messages** from the dropdown list.

Enable route: Be sure this field is set to `enabled`.

Routing query: Enter `level="storage"` as the query string.

 Add a route

* Name [?](#)
ContosoStorageRoute ✓

* Endpoint [?](#)
ContosoStorageEndpoint ▼ + 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="storage"

Save

Select **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.

Route 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 route 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, select your IoT hub, then select **Message Routing**.
2. In the **Message Routing** pane, select **+Add**.
3. On the **Add a Route** pane, Select **+Add** next to the Endpoint field. Select **Service Bus Queue**. You see the **Add Service Bus Endpoint** pane.



Add a service bus endpoint

Route your telemetry and device messages to Azure Service Bus and add publisher and subscriber capability.

* Endpoint name ?

ContosoSBQueueEndpoint2

Choose an existing service bus

Add an existing service bus queue or topic that shares a subscription with this IoT hub.

* Service bus namespace ?

ContosoSBNamespace32083

* Service bus queue ?

contososbqueue32083

Create

4. Fill in the fields:

Endpoint Name: Enter a name for the endpoint. This tutorial uses **ContosoSBQueueEndpoint**.

Service Bus Namespace: Use the dropdown list to select the service bus namespace you set up in the preparation steps. This tutorial uses **ContosoSBNamespace**.

Service Bus queue: Use the dropdown list to select the Service Bus queue. This tutorial uses **contososbqueue**.

5. Select **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 **ContosoSBQueueRoute**.

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 [?](#)
ContosoSBQueueRoute

* Endpoint [?](#)
ContosoSBQueueEndpoint [▼](#) [+ 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"

7. Select **Save**. When it returns to the Routes pane, you see both of your new routes, as displayed here.

Search (Ctrl+ /) [IoT devices](#) [Automatic Device Management](#) [IoT Edge](#) [IoT device configuration](#) [File upload](#) [Message routing](#) [Overview](#) [Security Alerts](#)

Send data from your devices to endpoints that you choose.

Routes [Custom endpoints](#) [Enrich messages](#)

[Disable fallback route](#) [+ Add](#) [Test all routes](#) [Delete](#)

| Name | Data Source | Routing Query | Endpoint | Enabled |
|---------------------|----------------|------------------|------------------------|---------|
| ContosoStorageRoute | DeviceMessages | level="storage" | ContosoStorageEndpoint | true |
| ContosoSBQueueRoute | DeviceMessages | level="critical" | ContosoSBQueueEndpoint | true |

8. You can see the custom endpoints you set up by selecting the **Custom Endpoints** tab.

Search (Ctrl+ /) [Settings](#) [Shared access policies](#) [Identity](#) [Pricing and scale](#) [Networking](#) [Certificates](#) [Built-in endpoints](#) [Failover](#) [Properties](#) [Locks](#) [Export template](#) [Explorers](#) [Query explorer](#) [IoT devices](#) [Automatic Device Management](#) [IoT Edge](#) [IoT device configuration](#) [File upload](#) [Message routing](#)

Send data from your devices to endpoints that you choose.

Custom endpoints [Routes](#) [Enrich messages](#)

Choose which Azure services will receive your messages. You can add up to 10 endpoints to an IoT hub.

[+ Add](#) [Synchronize keys](#) [Delete](#) [Refresh](#)

- [Event Hubs](#)
- [Service Bus queue](#)

Recommended for scenarios that require minimized processing. Messages can be locked or deleted after being read.

| Name | Namespace | Queue | Authentication type | Status |
|------------------------|---------------------|------------------|---------------------|---|
| ContosoSBQueueEndpoint | contosobnamespace23 | contososbqueue23 | Key-based | Healthy. |
- [Service Bus topic](#)
- [Storage](#)

Recommended for archiving data.

| Name | Container name | Encoding format | Batch frequency(sec.) | Filename format | Authentication type | Status |
|----------------------|--------------------|-----------------|-----------------------|--|---------------------|---|
| ContosoStorageEnd... | contosoresources23 | AVRO | 100 | (iothub)/(partition)/(YYYY)/(MM)/(DD)/(HH)/(mm).avro | Key-based | Healthy. |

9. Close the Message Routing pane, which returns you to the Resource group pane.

Create a simulated device

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 or when using an Azure Resource Manager template. The following steps tell you how to create the simulated device using the [Azure portal](#).

1. Open the [Azure portal](#) and log into your Azure account.
2. Select **Resource groups** and then choose your resource group. This tutorial uses **ContosoResources**.
3. In the list of resources, select your IoT hub. This tutorial uses **ContosoTestHub**. Select **IoT Devices** from the Hub pane.
4. Select **+ 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. Select **Save**.

The screenshot shows the 'Create a device' dialog box. At the top, there's a header with a gear icon and the title 'Create a device'. Below the header is a search bar with the placeholder 'Find Certified for Azure IoT devices in the Device Catalog'. The main form area contains the following fields:

- Device ID**: The input field contains 'Contoso-Test-Device' with a green checkmark icon to its right.
- Authentication type**: A radio button group with 'Symmetric key' selected, and 'X.509 Self-Signed' and 'X.509 CA Signed' as options.
- Primary key**: An input field with the placeholder 'Enter your primary key'.
- Secondary key**: An input field with the placeholder 'Enter your secondary key'.
- Auto-generate keys**: A checkbox that is checked.
- Connect this device to an IoT hub**: A button group with 'Enable' selected and 'Disable' as an option.
- Parent device (preview)**: A section showing 'No parent device' and a link 'Set a parent device'.

At the bottom of the dialog is a large blue 'Save' button.

5. Now that it's been created, select the device to see the generated keys. Select the Copy icon on the Primary key and save it somewhere such as Notepad for the testing phase of this tutorial.

Device details
Contoso-Test-Device

Save Message to device Direct method Device twin Add module identity Regenerate keys Refresh

Device Id: Contoso-Test-Device

Primary key: 72ChD3RXLC0crdWRwgYCjtJKPyQ8fECETukWYVdN97w=

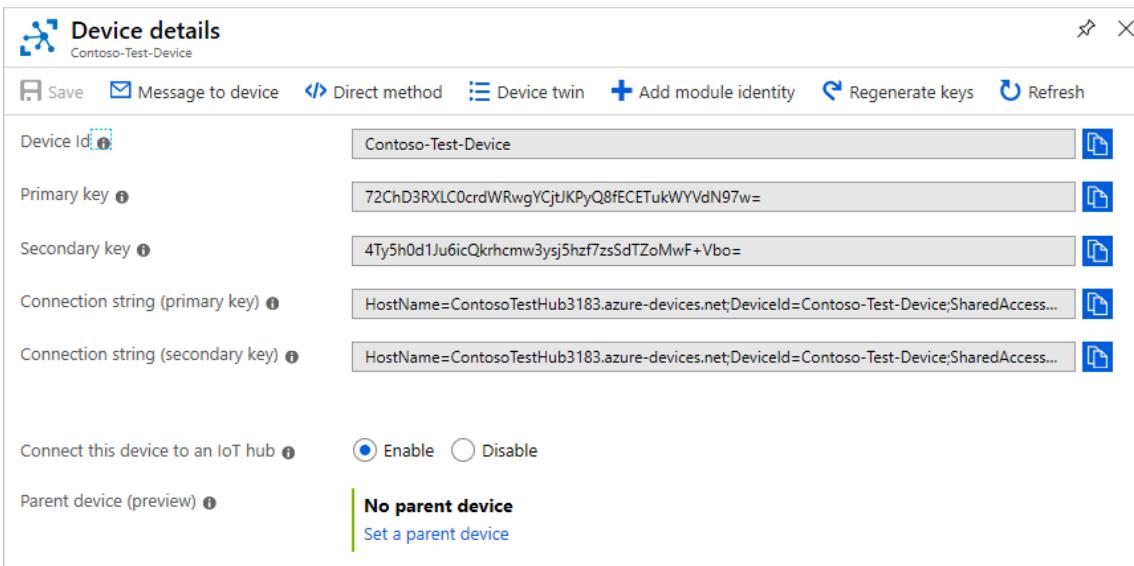
Secondary key: 4Ty5h0d1Ju6icQkrhcmw3ysj5hzf7zsSdTZoMwF+Vbo=

Connection string (primary key): HostName=ContosoTestHub3183.azure-devices.net;DeviceId=Contoso-Test-Device;SharedAccess...

Connection string (secondary key): HostName=ContosoTestHub3183.azure-devices.net;DeviceId=Contoso-Test-Device;SharedAccess...

Connect this device to an IoT hub: Enable Disable

Parent device (preview): No parent device [Set a parent device](#)



Next steps

Now that you have the resources set up and the message routes configured, advance to the next tutorial to learn how to send messages to the IoT hub and see them be routed to the different destinations.

[Part 2 - View the message routing results](#)

Tutorial: Use an Azure Resource Manager template to configure IoT Hub message routing

11/14/2019 • 16 minutes to read • [Edit Online](#)

[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 Queues, Service Bus Topics, and Event Hubs. To configure custom message routing, you 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. If a message doesn't match any of the defined routing queries, it is routed to the default endpoint.

In this 2-part tutorial, you learn how to set up and use these custom routing queries with IoT Hub. You route messages from an IoT device to one of multiple endpoints, including blob storage and a Service Bus queue. Messages to the Service Bus queue are picked up by a Logic App and sent via e-mail. Messages that do not have custom message routing defined are sent to the default endpoint, then picked up by Azure Stream Analytics and viewed in a Power BI visualization.

To complete Parts 1 and 2 of this tutorial, you perform the following tasks:

Part I: Create resources, set up message routing

- Create the resources -- an IoT hub, a storage account, a Service Bus queue, and a simulated device. This can be done using the Azure portal, an Azure Resource Manager template, the Azure CLI, or Azure PowerShell.
- Configure the endpoints and message routes in IoT Hub for the storage account and Service Bus queue.

Part II: Send messages to the hub, view routed results

- 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

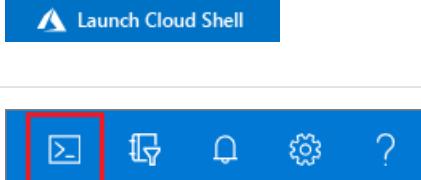
- For Part 1 of this tutorial:
 - You must have an Azure subscription. If you don't have an Azure subscription, create a [free account](#) before you begin.
- For Part 2 of this tutorial:
 - You must have completed Part 1 of this tutorial, and have the resources still available.
 - Install [Visual Studio](#).
 - Have access to a Power BI account to analyze the default endpoint's stream analytics. ([Try Power BI for free](#).)
 - Have a work or school account for sending notification e-mails.
 - Make sure that port 8883 is open in your firewall. The sample in this tutorial uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network

environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|--|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal . |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

Create base resources

Before you can configure the message routing, you need to create an IoT hub, a storage account, and a Service Bus queue. These resources can be created using one of the four articles that are available for Part 1 of this tutorial: the Azure portal, an Azure Resource Manager template, the Azure CLI, or Azure PowerShell.

Use the same resource group and location for all of the resources. Then at the end, you can remove all of the resources in one step by deleting the resource group.

Below is a summary of the steps to be performed in the following sections:

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. (If creating a Resource Manager template, this is done after deploying the template.)

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). In the next tutorial, you send messages to the IoT Hub and see them routed to the different destinations.

| 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. |

The first step is to set up the endpoint to which the data will be routed. The second step is to set up the message route that uses that endpoint. After setting up the routing, you can view the endpoints and message routes in the portal.

Download the template and parameters file

For the second part of this tutorial, you download and run a Visual Studio application to send messages to the IoT Hub. There is a folder in that download that contains the Azure Resource Manager template and parameters file, as well as the Azure CLI and PowerShell scripts.

Go ahead and download the [Azure IoT C# Samples](#) now. Unzip the master.zip file. The Resource Manager template and the parameters file are in /iot-hub/Tutorials/Routing/SimulatedDevice/resources/ as **template_iothub.json** and **template_iothub_parameters.json**.

Create your resources

You're going to use an Azure Resource Manager (RM) template to create all of your resources. The Azure CLI and PowerShell scripts can be run a few lines at a time. An RM template is deployed in one step. This article shows you the sections separately to help you understand each one. Then it will show you how to deploy the template, and create the virtual device for testing. After the template is deployed, you can view the message routing configuration in the portal.

There are several resource names that must be globally unique, such as the IoT Hub name and the storage account name. To make naming the resources easier, those resource names are set up to append a random alphanumeric value generated from the current date/time.

If you look at the template, you'll see where variables are set up for these resources that take the parameter passed in and concatenate *randomValue* to the parameter.

The following section explains the parameters used.

Parameters

Most of these parameters have default values. The ones ending with *_in* are concatenated with *randomValue* to make them globally unique.

randomValue: This value is generated from the current date/time when you deploy the template. This field is not in the parameters file, as it is generated in the template itself.

subscriptionId: This field is set for you to the subscription into which you are deploying the template. This field is not in the parameters file since it is set for you.

IoTHubName_in: This field is the base IoT Hub name, which is concatenated with the randomValue to be globally unique.

location: This field is the Azure region into which you are deploying, such as "westus".

consumer_group: This field is the consumer group set for messages coming through the routing endpoint. It is used to filter results in Azure Stream Analytics. For example, there is the whole stream where you get everything, or if you have data coming through with consumer_group set to **Contoso**, then you can set up an Azure Stream Analytics stream (and Power BI report) to show only those entries. This field is used in part 2 of this tutorial.

sku_name: This field is the scaling for the IoT Hub. This value must be S1 or above; a free tier does not work for this tutorial because it does not allow multiple endpoints.

sku_units: This field goes with the **sku_name**, and is the number of IoT Hub units that can be used.

d2c_partitions: This field is the number of partitions used for the event stream.

storageAccountName_in: This field is the name of the storage account to be created. Messages are routed to a container in the storage account. This field is concatenated with the randomValue to make it globally unique.

storageContainerName: This field is the name of the container in which the messages routed to the storage account are stored.

storage_endpoint: This field is the name for the storage account endpoint used by the message routing.

service_bus_namespace_in: This field is the name of the Service Bus namespace to be created. This value is concatenated with the randomValue to make it globally unique.

service_bus_queue_in: This field is the name of the Service Bus queue used for routing messages. This value is concatenated with the randomValue to make it globally unique.

AuthRules_sb_queue: This field is the authorization rules for the service bus queue, used to retrieve the connection string for the queue.

Variables

These values are used in the template, and are mostly derived from parameters.

queueAuthorizationRuleResourceId: This field is the ResourceId for the authorization rule for the Service Bus queue. ResourceId is in turn used to retrieve the connection string for the queue.

iotHubName: This field is the name of the IoT Hub after having randomValue concatenated.

storageAccountName: This field is the name of the storage account after having randomValue concatenated.

service_bus_namespace: This field is the namespace after having randomValue concatenated.

service_bus_queue: This field is the Service Bus queue name after having randomValue concatenated.

sbVersion: The version of the Service Bus API to use. In this case, it is "2017-04-01".

Resources: Storage account and container

The first resource created is the storage account, along with the container to which messages are routed. The container is a resource under the storage account. It has a **dependsOn** clause for the storage account, requiring the storage account be created before the container.

Here's what this section looks like:

```
{  
    "type": "Microsoft.Storage/storageAccounts",  
    "name": "[variables('storageAccountName')]",  
    "apiVersion": "2018-07-01",  
    "location": "[parameters('location')]",  
    "sku": {  
        "name": "Standard_LRS",  
        "tier": "Standard"  
    },  
    "kind": "Storage",  
    "properties": {},  
    "resources": [  
        {  
            "type": "blobServices/containers",  
            "apiVersion": "2018-07-01",  
            "name": "[concat('default/', parameters('storageContainerName'))]",  
            "properties": {  
                "publicAccess": "None"  
            },  
            "dependsOn": [  
                "[resourceId('Microsoft.Storage/storageAccounts', variables('storageAccountName'))]"  
            ]  
        }  
    ]  
}
```

Resources: Service Bus namespace and queue

The second resource created is the Service Bus namespace, along with the Service Bus queue to which messages are routed. The SKU is set to standard. The API version is retrieved from the variables. It is also set to activate the Service Bus namespace when it deploys this section (status:Active).

```
{  
    "type": "Microsoft.ServiceBus/namespaces",  
    "comments": "The Sku should be 'Standard' for this tutorial.",  
    "sku": {  
        "name": "Standard",  
        "tier": "Standard"  
    },  
    "name": "[variables('service_bus_namespace')]",  
    "apiVersion": "[variables('sbVersion')]",  
    "location": "[parameters('location')]",  
    "properties": {  
        "provisioningState": "Succeeded",  
        "metricId": "[concat('a4295411-5eff-4f81-b77e-276ab1ccda12:', variables('service_bus_namespace'))]",  
        "serviceBusEndpoint": "[concat('https://',  
variables('service_bus_namespace'), '.servicebus.windows.net:443/')]",  
        "status": "Active"  
    },  
    "dependsOn": []  
}
```

This section creates the Service Bus queue. This part of the script has a `dependsOn` clause that ensures the namespace is created before the queue.

```
{
  "type": "Microsoft.ServiceBus/namespaces/queues",
  "name": "[concat(variables('service_bus_namespace'), '/', variables('service_bus_queue'))]",
  "apiVersion": "[variables('sbVersion')]",
  "location": "[parameters('location')]",
  "scale": null,
  "properties": {},
  "dependsOn": [
    "[resourceId('Microsoft.ServiceBus/namespaces', variables('service_bus_namespace'))]"
  ]
}
```

Resources: IoT Hub and message routing

Now that the storage account and Service Bus queue have been created, you create the IoT Hub that routes messages to them. The RM template uses `dependsOn` clauses so it doesn't try to create the hub before the Service Bus resources and the storage account have been created.

Here's the first part of the IoT Hub section. This part of the template sets up the dependencies and starts with the properties.

```
{
  "apiVersion": "2018-04-01",
  "type": "Microsoft.Devices/IotHubs",
  "name": "[variables('IoTHubName')]",
  "location": "[parameters('location')]",
  "dependsOn": [
    "[resourceId('Microsoft.Storage/storageAccounts', variables('storageAccountName'))]",
    "[resourceId('Microsoft.ServiceBus/namespaces', variables('service_bus_namespace'))]",
    "[resourceId('Microsoft.ServiceBus/namespaces/queues', variables('service_bus_namespace'),
variables('service_bus_queue'))]"
  ],
  "properties": {
    "eventHubEndpoints": {}
  },
  "events": {
    "retentionTimeInDays": 1,
    "partitionCount": "[parameters('d2c_partitions')]"
  }
},
```

The next section is the section for the message routing configuration for the IoT Hub. First is the section for the endpoints. This part of the template sets up the routing endpoints for the Service Bus queue and the storage account, including the connection strings.

To create the connection string for the queue, you need the `queueAuthorizationRulesResourceId`, which is retrieved inline. To create the connection string for the storage account, you retrieve the primary storage key and then use it in the format for the connection string.

The endpoint configuration is also where you set the blob format to `AVRO` or `JSON`.

NOTE

The data can be written to blob storage in either the [Apache Avro](#) format, which is the default, or JSON (preview).

The capability to encode JSON format is in preview in all regions in which IoT Hub is available, except East US, West US and West Europe. The encoding format can be only set at the time the blob storage endpoint is configured. The format cannot be changed for an endpoint that has already been set up. When using JSON encoding, you must set the `contentType` to `JSON` and the `contentEncoding` to `UTF-8` in the message system properties.

For more detailed information about using a blob storage endpoint, please see [guidance on routing to storage](#).

```

"routing": {
    "endpoints": {
        "serviceBusQueues": [
            {
                "connectionString": "[Concat('Endpoint=sb://',variables('service_bus_namespace'),'.servicebus.windows.net/;SharedAccessKeyName=',parameters('AuthRules_sb_queue'),';SharedAccessKey=',listkeys(variables('queueAuthorizationRuleResourceId')),variables('sbVersion')).primaryKey,';EntityPath=',variables('service_bus_queue'))]",

                "name": "[parameters('service_bus_queue_endpoint')]",
                "subscriptionId": "[parameters('subscriptionId')]",
                "resourceGroup": "[resourceGroup().Name]"
            }
        ],
        "serviceBusTopics": [],
        "eventHubs": [],
        "storageContainers": [
            {
                "connectionString": "[Concat('DefaultEndpointsProtocol=https;AccountName=',variables('storageAccountName'),';AccountKey=',listKeys(resourceId('Microsoft.Storage/storageAccounts', variables('storageAccountName')), providers('Microsoft.Storage', 'storageAccounts').apiVersions[0]).keys[0].value)]",

                "containerName": "[parameters('storageContainerName')]",
                "fileNameFormat": "{iothub}/{partition}/{YYYY}/{MM}/{DD}/{HH}/{mm}",
                "batchFrequencyInSeconds": 100,
                "maxChunkSizeInBytes": 104857600,
                "encoding": "avro",
                "name": "[parameters('storage_endpoint')]",
                "subscriptionId": "[parameters('subscriptionId')]",
                "resourceGroup": "[resourceGroup().Name]"
            }
        ]
    },
}

```

This next section is for the message routes to the endpoints. There is one set up for each endpoint, so there is one for the Service Bus queue and one for the storage account container.

Remember that the query condition for the messages being routed to storage is `level="storage"`, and the query condition for the messages being routed to the Service Bus queue is `level="critical"`.

```

"routes": [
    {
        "name": "contosoStorageRoute",
        "source": "DeviceMessages",
        "condition": "level=\"storage\"",
        "endpointNames": [
            "[parameters('storage_endpoint')]"
        ],
        "isEnabled": true
    },
    {
        "name": "contosoSBQueueRoute",
        "source": "DeviceMessages",
        "condition": "level=\"critical\"",
        "endpointNames": [
            "[parameters('service_bus_queue_endpoint')]"
        ],
        "isEnabled": true
    }
],

```

This json shows the rest of the IoT Hub section, which contains default information and the SKU for the hub.

```

    "fallbackRoute": {
        "name": "$fallback",
        "source": "DeviceMessages",
        "condition": "true",
        "endpointNames": [
            "events"
        ],
        "isEnabled": true
    }
},
"storageEndpoints": {
    "$default": {
        "sasTtlAsIso8601": "PT1H",
        "connectionString": "",
        "containerName": ""
    }
},
"messagingEndpoints": {
    "fileNotifications": {
        "lockDurationAsIso8601": "PT1M",
        "ttlAsIso8601": "PT1H",
        "maxDeliveryCount": 10
    }
},
"enableFileUploadNotifications": false,
"cloudToDevice": {
    "maxDeliveryCount": 10,
    "defaultTtlAsIso8601": "PT1H",
    "feedback": {
        "lockDurationAsIso8601": "PT1M",
        "ttlAsIso8601": "PT1H",
        "maxDeliveryCount": 10
    }
},
"sku": {
    "name": "[parameters('sku_name')]",
    "capacity": "[parameters('sku_units')]"
}
}
}

```

Resources: Service Bus queue authorization rules

The Service Bus queue authorization rule is used to retrieve the connection string for the Service Bus queue. It uses a `dependsOn` clause to ensure it is not created before the Service Bus namespace and the Service Bus queue.

```

{
    "type": "Microsoft.ServiceBus/namespaces/queues/authorizationRules",
    "name": "[concat(variables('service_bus_namespace'), '/', variables('service_bus_queue'), '/', parameters('AuthRules_sb_queue'))]",
    "apiVersion": "[variables('sbVersion')]",
    "location": "[parameters('location')]",
    "scale": null,
    "properties": {
        "rights": [
            "Send"
        ]
    },
    "dependsOn": [
        "[resourceId('Microsoft.ServiceBus/namespaces', variables('service_bus_namespace'))]",
        "[resourceId('Microsoft.ServiceBus/namespaces/queues', variables('service_bus_namespace'), variables('service_bus_queue'))]"
    ]
},

```

Resources: Consumer group

In this section, you create a Consumer Group for the IoT Hub data to be used by the Azure Stream Analytics in the second part of this tutorial.

```
{  
    "type": "Microsoft.Devices/IotHubs/eventHubEndpoints/ConsumerGroups",  
    "name": "[concat(variables('iotHubName'), '/events/', parameters('consumer_group'))]",  
    "apiVersion": "2018-04-01",  
    "dependsOn": [  
        "[concat('Microsoft.Devices/IotHubs/', variables('iotHubName'))]"  
    ]  
}
```

Resources: Outputs

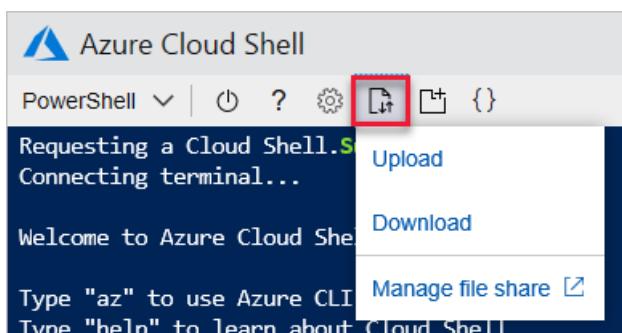
If you want to send a value back to the deployment script to be displayed, you use an output section. This part of the template returns the connection string for the Service Bus queue. Returning a value isn't required, it's included as an example of how to return results to the calling script.

```
"outputs": {  
    "sbq_connectionString": {  
        "type": "string",  
        "value": "  
[Concat('Endpoint=sb://',variables('service_bus_namespace'),'.servicebus.windows.net/;SharedAccessKeyName=',parameters('AuthRules_sb_queue'),';SharedAccessKey=',listkeys(variables('queueAuthorizationRuleResourceId')),variables('sbVersion')).primaryKey,';EntityPath=',variables('service_bus_queue'))]"  
    }  
}
```

Deploy the RM template

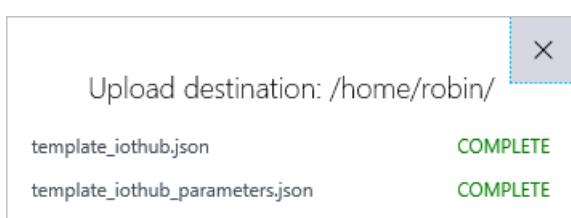
To deploy the template to Azure, upload the template and the parameters file to Azure Cloud Shell, and then execute a script to deploy the template. Open Azure Cloud Shell and sign in. This example uses PowerShell.

To upload the files, select the **Upload/Download files** icon in the menu bar, then choose Upload.



Use the File Explorer that pops up to find the files on your local disk and select them, then choose Open.

After the files are uploaded, a results dialog shows something like the following image.



The files are uploaded to the share used by your Cloud Shell instance.

Run the script to perform the deployment. The last line of this script retrieves the variable that was set up to be returned -- the Service Bus queue connection string.

The script sets and uses these variables:

\$RGName is the resource group name to which to deploy the template. This field is created before deploying the template.

\$location is the Azure location to be used for the template, such as "westus".

deploymentname is a name you assign to the deployment to retrieve the returning variable value.

Here's the PowerShell script. Copy this PowerShell script and paste it into the Cloud Shell window, then hit Enter to run it.

```
$RGName="ContosoResources"
$location = "westus"
$deploymentname="contoso-routing"

# Remove the resource group if it already exists.
#Remove-AzResourceGroup -name $RGName
# Create the resource group.
New-AzResourceGroup -name $RGName -Location $location

# Set a path to the parameter file.
$parameterFile = "$HOME/template_iothub_parameters.json"
$templateFile = "$HOME/template_iothub.json"

# Deploy the template.
New-AzResourceGroupDeployment ` 
    -Name $deploymentname ` 
    -ResourceGroupName $RGName ` 
    -TemplateParameterFile $parameterFile ` 
    -TemplateFile $templateFile ` 
    -verbose

# Get the returning value of the connection string.
(Get-AzResourceGroupDeployment -ResourceGroupName $RGName -Name
$deploymentname).Outputs.sq_connectionString.value
```

If you have script errors, you can edit the script locally, upload it again to the Cloud Shell, and run the script again. After the script finishes running successfully, continue to the next step.

Create simulated device

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 or when using an Azure Resource Manager template. The following steps tell you how to create the simulated device using the [Azure portal](#).

1. Open the [Azure portal](#) and log into your Azure account.
2. Select **Resource groups** and then choose your resource group. This tutorial uses **ContosoResources**.
3. In the list of resources, select your IoT hub. This tutorial uses **ContosoTestHub**. Select **IoT Devices** from the Hub pane.
4. Select **+ 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. Select **Save**.

Create a device

Find Certified for Azure IoT devices in the Device Catalog

* Device ID ⓘ
Contoso-Test-Device ✓

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

5. Now that it's been created, select the device to see the generated keys. Select the Copy icon on the Primary key and save it somewhere such as Notepad for the testing phase of this tutorial.

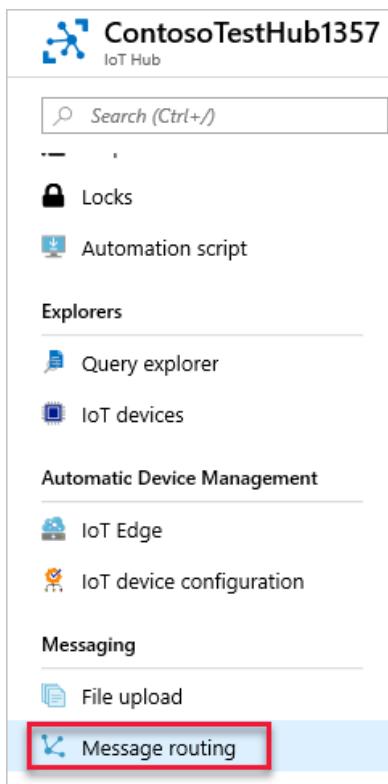
Device details
Contoso-Test-Device

Save Message to device Direct method Device twin Add module identity Regenerate keys Refresh

| | | |
|-------------------------------------|--|--|
| Device Id ⓘ | Contoso-Test-Device | |
| Primary key ⓘ | 72ChD3RLC0crdWRwgYCjtKPyQ8fECETukWYVdN97w= | |
| Secondary key ⓘ | 4Ty5h0d1Ju6icQkrhcmw3ysj5hzf7zsSdTzoMwf+Vbo= | |
| Connection string (primary key) ⓘ | HostName=ContosoTestHub3183.azure-devices.net;DeviceId=Contoso-Test-Device;SharedAccess... | |
| Connection string (secondary key) ⓘ | HostName=ContosoTestHub3183.azure-devices.net;DeviceId=Contoso-Test-Device;SharedAccess... | |
| Connect this device to an IoT hub ⓘ | <input checked="" type="radio"/> Enable <input type="radio"/> Disable | |
| Parent device (preview) ⓘ | No parent device Set a parent device | |

View message routing in the portal

Now that your endpoints and message routes are set up, you can view their configuration in the portal. Sign in to the [Azure portal](#) and go to **Resource Groups**. Next, select your resource group, then select your hub (the hub name starts with `ContosoTestHub` in this tutorial). You see the IoT Hub pane.



In the options for the IoT Hub, select **Message Routing**. The routes you have set up successfully are displayed.

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 |
|--------------------------|---------------------|----------------|------------------|------------------------|---------|
| <input type="checkbox"/> | ContosoStorageRoute | DeviceMessages | level="storage" | ContosoStorageEndpoint | true |
| <input type="checkbox"/> | ContosoSBQueueRoute | DeviceMessages | level="critical" | ContosoSBQueueEndpoint | true |

On the **Message routing** screen, select **Custom Endpoints** to see the endpoints you have defined for the routes.

Send data from your devices to endpoints that you choose.

Routes Custom endpoints

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 |
|--------------------------|------------------------|--------------------|--------------------|---|
| | ContosoSBQueueEndpoint | contososbnamespace | ContosoSBQueue2468 | Unknown |

▼ Service Bus topic

^ Blob storage

Recommended for storage.

| <input type="checkbox"/> | NAME | CON... | ENCODING FO... | BATCH FREQUE... | FILENAME FOR... | STATUS |
|--------------------------|------------------------|---------|----------------|-----------------|------------------|---|
| | ContosoStorageEndpo... | cont... | AVRO | 100 | {iothub}/{par... | Unknown |

Next steps

Now that you have all of the resources set up and the message routes are configured, advance to the next tutorial to learn how to process and display the information about the routed messages.

[Part 2 - View the message routing results](#)

Tutorial: Use the Azure CLI to configure IoT Hub message routing

3/6/2020 • 14 minutes to read • [Edit Online](#)

[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 Queues, Service Bus Topics, and Event Hubs. To configure custom message routing, you 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. If a message doesn't match any of the defined routing queries, it is routed to the default endpoint.

In this 2-part tutorial, you learn how to set up and use these custom routing queries with IoT Hub. You route messages from an IoT device to one of multiple endpoints, including blob storage and a Service Bus queue. Messages to the Service Bus queue are picked up by a Logic App and sent via e-mail. Messages that do not have custom message routing defined are sent to the default endpoint, then picked up by Azure Stream Analytics and viewed in a Power BI visualization.

To complete Parts 1 and 2 of this tutorial, you perform the following tasks:

Part I: Create resources, set up message routing

- Create the resources -- an IoT hub, a storage account, a Service Bus queue, and a simulated device. This can be done using the Azure portal, an Azure Resource Manager template, the Azure CLI, or Azure PowerShell.
- Configure the endpoints and message routes in IoT Hub for the storage account and Service Bus queue.

Part II: Send messages to the hub, view routed results

- 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

- For Part 1 of this tutorial:
 - You must have an Azure subscription. If you don't have an Azure subscription, create a [free account](#) before you begin.
- For Part 2 of this tutorial:
 - You must have completed Part 1 of this tutorial, and have the resources still available.
 - Install [Visual Studio](#).
 - Have access to a Power BI account to analyze the default endpoint's stream analytics. ([Try Power BI for free.](#))
 - Have a work or school account for sending notification e-mails.
 - Make sure that port 8883 is open in your firewall. The sample in this tutorial uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network

environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|--|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal . |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

Create base resources

Before you can configure the message routing, you need to create an IoT hub, a storage account, and a Service Bus queue. These resources can be created using one of the four articles that are available for Part 1 of this tutorial: the Azure portal, an Azure Resource Manager template, the Azure CLI, or Azure PowerShell.

Use the same resource group and location for all of the resources. Then at the end, you can remove all of the resources in one step by deleting the resource group.

Below is a summary of the steps to be performed in the following sections:

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. (If creating a Resource Manager template, this is done after deploying the template.)

Download the script (optional)

For the second part of this tutorial, you download and run a Visual Studio application to send messages to the IoT Hub. There is a folder in the download that contains the Azure Resource Manager template and parameters file, as well as the Azure CLI and PowerShell scripts.

If you want to view the finished script, download the [Azure IoT C# Samples](#). Unzip the master.zip file. The Azure CLI script is in /iot-hub/Tutorials/Routing/SimulatedDevice/resources/ as `iothub_routing_cli.azcli`.

Use the Azure CLI to create your resources

Copy and paste the script below into Cloud Shell and press Enter. It runs the script one line at a time. This first section of the script will create the base resources for this tutorial, including the storage account, IoT Hub, Service Bus Namespace, and Service Bus queue. As you go through the rest of the tutorial, copy each block of script and paste it into Cloud Shell to run it.

TIP

A tip about debugging: this script uses the continuation symbol (the backslash \) to make the script more readable. If you have a problem running the script, make sure your Cloud Shell session is running `bash` and that there are no spaces after any of the backslashes.

There are several resource names that must be globally unique, such as the IoT Hub name and the storage account name. To make this easier, those resource names are appended with a random alphanumeric value called *randomValue*. The *randomValue* is generated once at the top of the script and appended to the resource names as needed throughout the script. If you don't want it to be random, you can set it to an empty string or to a specific value.

IMPORTANT

The variables set in the initial script are also used by the routing script, so run all of the script in the same Cloud Shell session. If you open a new session to run the script for setting up the routing, several of the variables will be missing values.

```
# This command retrieves the subscription id of the current Azure account.  
# This field is used when setting up the routing queries.  
subscriptionID=$(az account show --query id -o tsv)  
  
# Concatenate this number onto the resources that have to be globally unique.  
# You can set this to "" or to a specific value if you don't want it to be random.  
# This retrieves a random value.  
randomValue=$RANDOM  
  
# This command installs 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-iot  
  
# Set the values for the resource names that  
# don't have to be globally unique.  
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 value to the end.
iotHubName=ContosoTestHub$randomValue
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 for the 'events' endpoint.
az iot hub consumer-group create --hub-name $iotHubName \
    --name $iotHubConsumerGroup

# The storage account name must be globally unique,
#   so add a random value to the end.
storageAccountName=contosostorage$randomValue
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 "storage account key = " $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 value to the end.
sbNamespace=ContosoSBNamespace$randomValue
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 value to the end.
sbQueueName=ContosoSBQueue$randomValue
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

```

Now that the base resources are set up, you can configure the message routing.

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). In the next tutorial, you send messages to the IoT Hub and see them routed to the different destinations.

| 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. |

The first step is to set up the endpoint to which the data will be routed. The second step is to set up the message route that uses that endpoint. After setting up the routing, you can view the endpoints and message routes in the portal.

To create a routing endpoint, use [az iot hub routing-endpoint create](#). To create the message route for the endpoint, use [az iot hub route create](#).

Route to a storage account

NOTE

The data can be written to blob storage in either the [Apache Avro](#) format, which is the default, or JSON (preview).

The capability to encode JSON format is in preview in all regions in which IoT Hub is available, except East US, West US and West Europe. The encoding format can be only set at the time the blob storage endpoint is configured. The format cannot be changed for an endpoint that has already been set up. When using JSON encoding, you must set the `contentType` to JSON and the `contentEncoding` to UTF-8 in the message system properties.

For more detailed information about using a blob storage endpoint, please see [guidance on routing to storage](#).

First, set up the endpoint for the storage account, then set up the route.

These are the variables used by the script that must be set within your Cloud Shell session:

storageConnectionString: This value is retrieved from the storage account set up in the previous script. It is used by the message routing to access the storage account.

resourceGroup: There are two occurrences of resource group -- set them to your resource group.

endpoint subscriptionID: This field is set to the Azure subscriptionID for the endpoint.

endpointType: This field is the type of endpoint. This value must be set to `azurestoragecontainer`, `eventhub`,

`servicebusqueue`, or `servicebustopic`. For your purposes here, set it to `azurestoragecontainer`.

iotHubName: This field is the name of the hub that will do the routing.

containerName: This field is the name of the container in the storage account to which data will be written.

encoding: This field will be either `avro` or `json`. This denotes the format of the stored data.

routeName: This field is the name of the route you are setting up.

endpointName: This field is the name identifying the endpoint.

enabled: This field defaults to `true`, indicating that the message route should be enabled after being created.

condition: This field is the query used to filter for the messages sent to this endpoint. The query condition for the messages being routed to storage is `level="storage"`.

Copy this script and paste it into your Cloud Shell window and run it.

```
##### ROUTING FOR STORAGE #####
endpointName="ContosoStorageEndpoint"
endpointType="azurestoragecontainer"
routeName="ContosoStorageRoute"
condition='level="storage"'

# Get the connection string for the storage account.
# Adding the "-o tsv" makes it be returned without the default double quotes around it.
storageConnectionString=$(az storage account show-connection-string \
--name $storageAccountName --query connectionString -o tsv)
```

The next step is to create the routing endpoint for the storage account. You also specify the container in which the results will be stored. The container was created previously when the storage account was created.

```
# Create the routing endpoint for storage.
az iot hub routing-endpoint create \
--connection-string $storageConnectionString \
--endpoint-name $endpointName \
--endpoint-resource-group $resourceGroup \
--endpoint-subscription-id $subscriptionID \
--endpoint-type $endpointType \
--hub-name $iotHubName \
--container $containerName \
--resource-group $resourceGroup \
--encoding avro
```

Next, create the route for the storage endpoint. The message route designates where to send the messages that meet the query specification.

```
# Create the route for the storage endpoint.
az iot hub route create \
--name $routeName \
--hub-name $iotHubName \
--source devicemessages \
--resource-group $resourceGroup \
--endpoint-name $endpointName \
--enabled \
--condition $condition
```

Route to a Service Bus queue

Now set up the routing for the Service Bus queue. To retrieve the connection string for the Service Bus queue, you

must create an authorization rule that has the correct rights defined. The following script creates an authorization rule for the Service Bus queue called `sbauthrule`, and sets the rights to `Listen Manage Send`. Once this authorization rule is defined, you can use it to retrieve the connection string for the queue.

```
# Create the authorization rule for the Service Bus queue.  
az servicebus queue authorization-rule create \  
  --name "sbauthrule" \  
  --namespace-name $sbNamespace \  
  --queue-name $sbQueueName \  
  --resource-group $resourceGroup \  
  --rights Listen Manage Send \  
  --subscription $subscriptionID
```

Now use the authorization rule to retrieve the connection string to the Service Bus queue.

```
# Get the Service Bus queue connection string.  
# The "-o tsv" ensures it is returned without the default double-quotes.  
sbqConnectionString=$(az servicebus queue authorization-rule keys list \  
  --name "sbauthrule" \  
  --namespace-name $sbNamespace \  
  --queue-name $sbQueueName \  
  --resource-group $resourceGroup \  
  --subscription $subscriptionID \  
  --query primaryConnectionString -o tsv)  
  
# Show the Service Bus queue connection string.  
echo "service bus queue connection string = " $sbqConnectionString
```

Now set up the routing endpoint and the message route for the Service Bus queue. These are the variables used by the script that must be set within your Cloud Shell session:

endpointName: This field is the name identifying the endpoint.

endpointType: This field is the type of endpoint. This value must be set to `azurestoragecontainer`, `eventhub`, `servicebusqueue`, or `servicebustopic`. For your purposes here, set it to `servicebusqueue`.

routeName: This field is the name of the route you are setting up.

condition: This field is the query used to filter for the messages sent to this endpoint. The query condition for the messages being routed to the Service Bus queue is `level="critical"`.

Here is the Azure CLI for the routing endpoint and the message route for the Service Bus queue.

```

endpointName="ContosoSBQueueEndpoint"
endpointType="ServiceBusQueue"
routeName="ContosoSBQueueRoute"
condition='level="critical"'

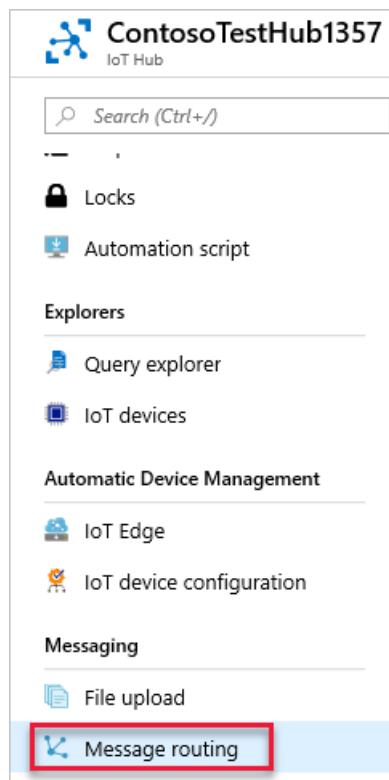
# Set up the routing endpoint for the Service Bus queue.
# This uses the Service Bus queue connection string.
az iot hub routing-endpoint create \
--connection-string $sbqConnectionString \
--endpoint-name $endpointName \
--endpoint-resource-group $resourceGroup \
--endpoint-subscription-id $subscriptionID \
--endpoint-type $endpointType \
--hub-name $iotHubName \
--resource-group $resourceGroup

# Set up the message route for the Service Bus queue endpoint.
az iot hub route create --name $routeName \
--hub-name $iotHubName \
--source-type devicemessages \
--resource-group $resourceGroup \
--endpoint-name $endpointName \
--enabled \
--condition $condition

```

View message routing in the portal

Now that your endpoints and message routes are set up, you can view their configuration in the portal. Sign in to the [Azure portal](#) and go to **Resource Groups**. Next, select your resource group, then select your hub (the hub name starts with `ContosoTestHub` in this tutorial). You see the IoT Hub pane.



In the options for the IoT Hub, select **Message Routing**. The routes you have set up successfully are displayed.

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 |
|--------------------------|---------------------|----------------|------------------|------------------------|---------|
| | ContosoStorageRoute | DeviceMessages | level="storage" | ContosoStorageEndpoint | true |
| | ContosoSBQueueRoute | DeviceMessages | level="critical" | ContosoSBQueueEndpoint | true |

On the Message routing screen, select **Custom Endpoints** to see the endpoints you have defined for the routes.

Send data from your devices to endpoints that you choose.

[Routes](#) [Custom endpoints](#)

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 |
|--------------------------|------------------------|---------------------|--------------------|---------|
| | ContosoSBQueueEndpoint | contososbnamespa... | ContosoSBQueue2468 | Unknown |

▼ Service Bus topic

^ Blob storage

Recommended for storage.

| <input type="checkbox"/> | NAME | CON... | ENCODING FO... | BATCH FREQUE... | FILENAME FOR... | STATUS |
|--------------------------|------------------------|---------|----------------|-----------------|------------------|---------|
| | ContosoStorageEndpo... | cont... | AVRO | 100 | {iothub}/{par... | Unknown |

Next steps

Now that you have the resources set up and the message routes configured, advance to the next tutorial to learn how to send messages to the IoT hub and see them be routed to the different destinations.

[Part 2 - View the message routing results](#)

Tutorial: Use Azure PowerShell to configure IoT Hub message routing

11/14/2019 • 14 minutes to read • [Edit Online](#)

[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 Queues, Service Bus Topics, and Event Hubs. To configure custom message routing, you 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. If a message doesn't match any of the defined routing queries, it is routed to the default endpoint.

In this 2-part tutorial, you learn how to set up and use these custom routing queries with IoT Hub. You route messages from an IoT device to one of multiple endpoints, including blob storage and a Service Bus queue. Messages to the Service Bus queue are picked up by a Logic App and sent via e-mail. Messages that do not have custom message routing defined are sent to the default endpoint, then picked up by Azure Stream Analytics and viewed in a Power BI visualization.

To complete Parts 1 and 2 of this tutorial, you perform the following tasks:

Part I: Create resources, set up message routing

- Create the resources -- an IoT hub, a storage account, a Service Bus queue, and a simulated device. This can be done using the Azure portal, an Azure Resource Manager template, the Azure CLI, or Azure PowerShell.
- Configure the endpoints and message routes in IoT Hub for the storage account and Service Bus queue.

Part II: Send messages to the hub, view routed results

- 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

- For Part 1 of this tutorial:
 - You must have an Azure subscription. If you don't have an Azure subscription, create a [free account](#) before you begin.
- For Part 2 of this tutorial:
 - You must have completed Part 1 of this tutorial, and have the resources still available.
 - Install [Visual Studio](#).
 - Have access to a Power BI account to analyze the default endpoint's stream analytics. ([Try Power BI for free.](#))
 - Have a work or school account for sending notification e-mails.
 - Make sure that port 8883 is open in your firewall. The sample in this tutorial uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network

environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|--|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal . |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

Create base resources

Before you can configure the message routing, you need to create an IoT hub, a storage account, and a Service Bus queue. These resources can be created using one of the four articles that are available for Part 1 of this tutorial: the Azure portal, an Azure Resource Manager template, the Azure CLI, or Azure PowerShell.

Use the same resource group and location for all of the resources. Then at the end, you can remove all of the resources in one step by deleting the resource group.

Below is a summary of the steps to be performed in the following sections:

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. (If creating a Resource Manager template, this is done after deploying the template.)

Download the script (optional)

For the second part of this tutorial, you download and run a Visual Studio application to send messages to the IoT Hub. There is a folder in the download that contains the Azure Resource Manager template and parameters file, as well as the Azure CLI and PowerShell scripts.

If you want to view the finished script, download the [Azure IoT C# Samples](#). Unzip the master.zip file. The Azure CLI script is in /iot-hub/Tutorials/Routing/SimulatedDevice/resources/ as `iothub_routing_psh.ps1`.

Create your resources

Start by creating the resources with PowerShell.

Use PowerShell to create your base resources

Copy and paste the script below into Cloud Shell and press Enter. It runs the script one line at a time. This first section of the script will create the base resources for this tutorial, including the storage account, IoT Hub, Service Bus Namespace, and Service Bus queue. As you go through the tutorial, copy each block of script and paste it into Cloud Shell to run it.

There are several resource names that must be globally unique, such as the IoT Hub name and the storage account name. To make this easier, those resource names are appended with a random alphanumeric value called *randomValue*. The *randomValue* is generated once at the top of the script and appended to the resource names as needed throughout the script. If you don't want it to be random, you can set it to an empty string or to a specific value.

IMPORTANT

The variables set in the initial script are also used by the routing script, so run all of the script in the same Cloud Shell session. If you open a new session to run the script for setting up the routing, several of the variables will be missing values.

```
# This command retrieves the subscription id of the current Azure account.  
# This field is used when setting up the routing queries.  
$subscriptionID = (Get-AzContext).Subscription.Id  
  
# Concatenate this number onto the resources that have to be globally unique.  
# You can set this to "" or to a specific value if you don't want it to be random.  
# This retrieves the first 6 digits of a random value.  
$randomValue = "$(Get-Random)".Substring(0,6)  
  
# Set the values for the resource names that don't have to be globally unique.  
$location = "West US"  
$resourceGroup = "ContosoResources"  
$iotHubConsumerGroup = "ContosoConsumers"  
$containerName = "contosoresults"  
  
# 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 value to the end.  
$iotHubName = "ContosoTestHub" + $randomValue  
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 value to the end.
$storageAccountName = "contosostorage" + $randomValue
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
# Retrieve the connection string from the context.
$storageConnectionString = $storageAccount.Context.ConnectionString
Write-Host "storage connection string = " $storageConnectionString

# Create the container in the storage account.
New-AzStorageContainer -Name $containerName ` 
    -Context $storageAccount.Context

# The Service Bus namespace must be globally unique,
#   so add a random value to the end.
$serviceBusNamespace = "ContosoSBNamespace" + $randomValue
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 value to the end.
$serviceBusQueueName = "ContosoSBQueue" + $randomValue
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 ` 
    -EnablePartitioning $False

```

Create a simulated device

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 or when using an Azure Resource Manager template. The following steps tell you how to create the simulated device using the [Azure portal](#).

1. Open the [Azure portal](#) and log into your Azure account.
2. Select **Resource groups** and then choose your resource group. This tutorial uses **ContosoResources**.
3. In the list of resources, select your IoT hub. This tutorial uses **ContosoTestHub**. Select **IoT Devices** from the Hub pane.

4. Select + 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. Select **Save**.

Create a device

Find Certified for Azure IoT devices in the Device Catalog

* Device ID: Contoso-Test-Device

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

5. Now that it's been created, select the device to see the generated keys. Select the Copy icon on the Primary key and save it somewhere such as Notepad for the testing phase of this tutorial.

Device details
Contoso-Test-Device

Save Message to device Direct method Device twin Add module identity Regenerate keys Refresh

Device Id: Contoso-Test-Device

Primary key: 72ChD3RLC0crdWRwgYCjtKPYQ8fECETukWVvdN97w=

Secondary key: 4Ty5h0d1Ju6icQkrhcmw3ysj5hzf7zsSdTzOMwF+Vbo=

Connection string (primary key): HostName=ContosoTestHub3183.azure-devices.net;DeviceId=Contoso-Test-Device;SharedAccess...

Connection string (secondary key): HostName=ContosoTestHub3183.azure-devices.net;DeviceId=Contoso-Test-Device;SharedAccess...

Connect this device to an IoT hub: Enable Disable

Parent device (preview): No parent device
Set a parent device

Now that the base resources are set up, you can configure the message routing.

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). In the next tutorial, you send messages to the IoT Hub and see them routed to the different destinations.

| 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. |

The first step is to set up the endpoint to which the data will be routed. The second step is to set up the message route that uses that endpoint. After setting up the routing, you can view the endpoints and message routes in the portal.

To create a routing endpoint, use [Add-AzIoTHubRoutingEndpoint](#). To create the messaging route for the endpoint, use [Add-AzIoTHubRoute](#).

Route to a storage account

First, set up the endpoint for the storage account, then create the message route.

NOTE

The data can be written to blob storage in either the [Apache Avro](#) format, which is the default, or JSON (preview).

The capability to encode JSON format is in preview in all regions in which IoT Hub is available, except East US, West US and West Europe. The encoding format can be only set at the time the blob storage endpoint is configured. The format cannot be changed for an endpoint that has already been set up. When using JSON encoding, you must set the `contentType` to `JSON` and the `contentEncoding` to `UTF-8` in the message system properties.

For more detailed information about using a blob storage endpoint, please see [guidance on routing to storage](#).

These are the variables used by the script that must be set within your Cloud Shell session:

resourceGroup: There are two occurrences of this field -- set both of them to your resource group.

name: This field is the name of the IoT Hub to which the routing will apply.

endpointName: This field is the name identifying the endpoint.

endpointType: This field is the type of endpoint. This value must be set to `azurestoragecontainer`, `eventhub`, `servicebusqueue`, or `servicebustopic`. For your purposes here, set it to `azurestoragecontainer`.

subscriptionID: This field is set to the `subscriptionID` for your Azure account.

storageConnectionString: This value is retrieved from the storage account set up in the previous script. It is used by the routing to access the storage account.

containerName: This field is the name of the container in the storage account to which data will be written.

Encoding: Set this field to either `AVRO` or `JSON`. This designates the format of the stored data. The default is AVRO.

routeName: This field is the name of the route you are setting up.

condition: This field is the query used to filter for the messages sent to this endpoint. The query condition for the messages being routed to storage is `level="storage"`.

enabled: This field defaults to `true`, indicating that the message route should be enabled after being created.

Copy this script and paste it into your Cloud Shell window.

```
##### ROUTING FOR STORAGE #####
$endpointName = "ContosoStorageEndpoint"
$endpointType = "azurorestoragecontainer"
$routeName = "ContosoStorageRoute"
$condition = 'level="storage"'
```

The next step is to create the routing endpoint for the storage account. You also specify the container in which the results will be stored. The container was created when the storage account was created.

```
# Create the routing endpoint for storage.
# Specify 'AVRO' or 'JSON' for the encoding of the data.
Add-AzIotHubRoutingEndpoint `

-ResourceGroupName $resourceGroup `
-Name $iotHubName `
-EndpointName $endpointName `
-EndpointType $endpointType `
-EndpointResourceGroup $resourceGroup `
-EndpointSubscriptionId $subscriptionId `
-ConnectionString $storageConnectionString `
-ContainerName $containerName `
-Encoding AVRO
```

Next, create the message route for the storage endpoint. The message route designates where to send the messages that meet the query specification.

```
# Create the route for the storage endpoint.
Add-AzIotHubRoute `

-ResourceGroupName $resourceGroup `
-Name $iotHubName `
-RouteName $routeName `
-Source DeviceMessages `
-EndpointName $endpointName `
-Condition $condition `
-Enabled
```

Route to a Service Bus queue

Now set up the routing for the Service Bus queue. To retrieve the connection string for the Service Bus queue, you must create an authorization rule that has the correct rights defined. The following script creates an authorization rule for the Service Bus queue called `sbauthrule`, and sets the rights to `Listen Manage Send`. Once this authorization rule is set up, you can use it to retrieve the connection string for the queue.

```
##### ROUTING FOR SERVICE BUS QUEUE #####
# Create the authorization rule for the Service Bus queue.
New-AzServiceBusAuthorizationRule `

-ResourceGroupName $resourceGroup `
-NamespaceName $serviceBusNamespace `
-Queue $serviceBusQueueName `
-Name "sbauthrule" `
-Rights @("Manage","Listen","Send")
```

Now use the authorization rule to retrieve the Service Bus queue key. This authorization rule will be used to retrieve the connection string later in the script.

```
$sbqkey = Get-AzServiceBusKey ` 
    -ResourceGroupName $resourceGroup ` 
    -NamespaceName $serviceBusNamespace ` 
    -Queue $servicebusQueueName ` 
    -Name "sbauthrule"
```

Now set up the routing endpoint and the message route for the Service Bus queue. These are the variables used by the script that must be set within your Cloud Shell session:

endpointName: This field is the name identifying the endpoint.

endpointType: This field is the type of endpoint. This value must be set to `azurestoragecontainer`, `eventhub`, `servicebusqueue`, or `servicebustopic`. For your purposes here, set it to `servicebusqueue`.

routeName: This field is the name of the route you are setting up.

condition: This field is the query used to filter for the messages sent to this endpoint. The query condition for the messages being routed to the Service Bus queue is `level="critical"`.

Here is the Azure PowerShell for the message routing for the Service Bus queue.

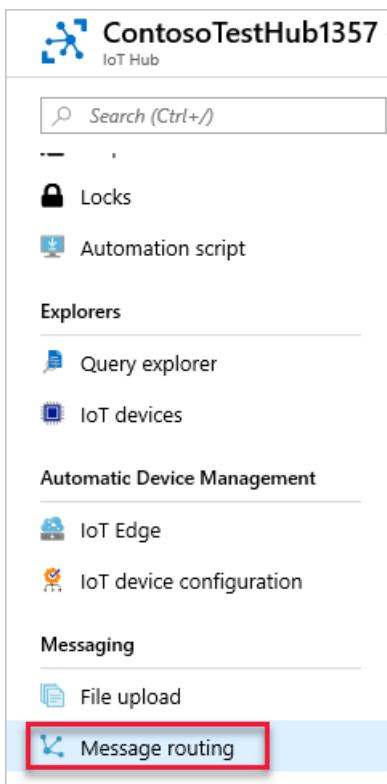
```
$endpointName = "ContosoSBQueueEndpoint"
$endpointType = "servicebusqueue"
$routeName = "ContosoSBQueueRoute"
$condition = 'level="critical"

# Add the routing endpoint, using the connection string property from the key.
Add-AzIotHubRoutingEndpoint ` 
    -ResourceGroupName $resourceGroup ` 
    -Name $iotHubName ` 
    -EndpointName $endpointName ` 
    -EndpointType $endpointType ` 
    -EndpointResourceGroup $resourceGroup ` 
    -EndpointSubscriptionId $subscriptionId ` 
    -ConnectionString $sbqkey.PrimaryConnectionString

# Set up the message route for the Service Bus queue endpoint.
Add-AzIotHubRoute ` 
    -ResourceGroupName $resourceGroup ` 
    -Name $iotHubName ` 
    -RouteName $routeName ` 
    -Source DeviceMessages ` 
    -EndpointName $endpointName ` 
    -Condition $condition ` 
    -Enabled
```

View message routing in the portal

Now that your endpoints and message routes are set up, you can view their configuration in the portal. Sign in to the [Azure portal](#) and go to **Resource Groups**. Next, select your resource group, then select your hub (the hub name starts with `ContosoTestHub` in this tutorial). You see the IoT Hub pane.



In the options for the IoT Hub, select **Message Routing**. The routes you have set up successfully are displayed.

This screenshot shows the 'Message routing' screen. It has tabs for 'Routes' (which is selected and highlighted with a red box) and 'Custom endpoints'. Below the tabs, it says '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.' There are buttons for 'Disable fallback route', 'Add', 'Test all routes', and 'Delete'. A table lists the routes:

| <input type="checkbox"/> | NAME | DATA SOURCE | ROUTING QUERY | ENDPOINT | ENABLED |
|--------------------------|---------------------|----------------|------------------|------------------------|---------|
| <input type="checkbox"/> | ContosoStorageRoute | DeviceMessages | level="storage" | ContosoStorageEndpoint | true |
| <input type="checkbox"/> | ContosoSBQueueRoute | DeviceMessages | level="critical" | ContosoSBQueueEndpoint | true |

On the **Message routing** screen, select **Custom Endpoints** to see the endpoints you have defined for the routes.

Send data from your devices to endpoints that you choose.

Routes **Custom endpoints**

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 |
|--------------------------|------------------------|--------------------|--------------------|---|
| | ContosoSBQueueEndpoint | contososbnamespace | ContosoSBQueue2468 |  Unknown |

▼ Service Bus topic

^ Blob storage

Recommended for storage.

| <input type="checkbox"/> | NAME | CON... | ENCODING FO... | BATCH FREQUE... | FILENAME FOR... | STATUS |
|--------------------------|------------------------|---------|----------------|-----------------|------------------|---|
| | ContosoStorageEndpo... | cont... | AVRO | 100 | (iothub)/(par... |  Unknown |

Next steps

Now that you have the resources set up and the message routes configured, advance to the next tutorial to learn how to send messages to the IoT hub and see them be routed to the different destinations.

[Part 2 - View the message routing results](#)

Tutorial: Part 2 – View the routed messages

11/8/2019 • 15 minutes to read • [Edit Online](#)

[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 Queues, Service Bus Topics, and Event Hubs. To configure custom message routing, you 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. If a message doesn't match any of the defined routing queries, it is routed to the default endpoint.

In this 2-part tutorial, you learn how to set up and use these custom routing queries with IoT Hub. You route messages from an IoT device to one of multiple endpoints, including blob storage and a Service Bus queue. Messages to the Service Bus queue are picked up by a Logic App and sent via e-mail. Messages that do not have custom message routing defined are sent to the default endpoint, then picked up by Azure Stream Analytics and viewed in a Power BI visualization.

To complete Parts 1 and 2 of this tutorial, you perform the following tasks:

Part I: Create resources, set up message routing

- Create the resources -- an IoT hub, a storage account, a Service Bus queue, and a simulated device. This can be done using the Azure portal, an Azure Resource Manager template, the Azure CLI, or Azure PowerShell.
- Configure the endpoints and message routes in IoT Hub for the storage account and Service Bus queue.

Part II: Send messages to the hub, view routed results

- 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.

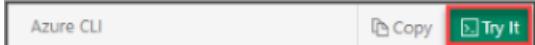
Prerequisites

- For Part 1 of this tutorial:
 - You must have an Azure subscription. If you don't have an Azure subscription, create a [free account](#) before you begin.
- For Part 2 of this tutorial:
 - You must have completed Part 1 of this tutorial, and have the resources still available.
 - Install [Visual Studio](#).
 - Have access to a Power BI account to analyze the default endpoint's stream analytics. ([Try Power BI for free.](#))
 - Have a work or school account for sending notification e-mails.
 - Make sure that port 8883 is open in your firewall. The sample in this tutorial uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|--|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal . |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

NOTE

This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module installation instructions, see [Install Azure PowerShell](#).

Rules for routing the messages

These are the rules for the message routing; these were set up in Part 1 of this tutorial, and you see them work in this second part.

| 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. |

Now you create the resources to which the messages will be routed, run an app to send messages to the hub, and

see the routing in action.

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](#), select **+ Create a resource**. Put **logic app** in the search box and click **Enter**. From the search results displayed, select **Logic App**, then select **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: Select **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.

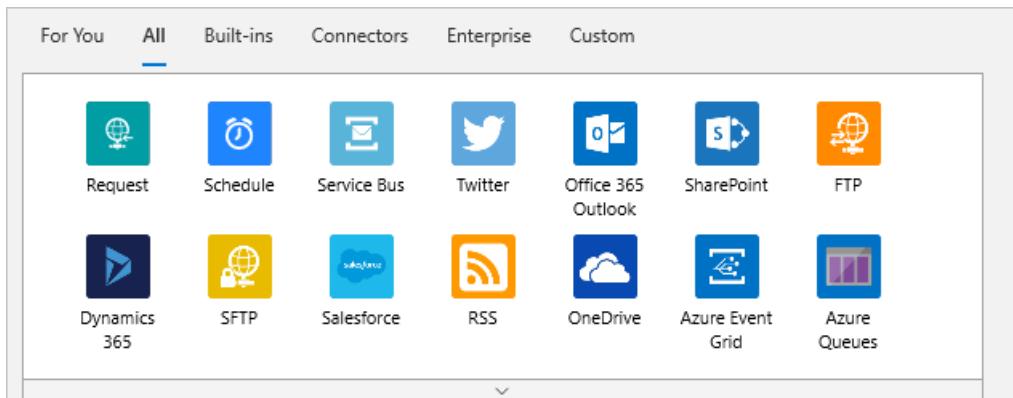
The screenshot shows the 'Create logic app' pane in the Azure portal. It includes fields for Name (ContosoLogicApp), Subscription (Microsoft Azure Internal Consumption (a42)), Resource group (ContosoResources), Location (West US), and Log Analytics (Off). A note at the bottom says 'You can add triggers and actions to your Logic App after creation.' At the bottom are 'Create' and 'Automation options' buttons.

Select **Create**. It may take a few minutes for the app to deploy.

2. Now go to the Logic App. The easiest way to get to the Logic App is to select **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 select it. The default tab is "For You". If this pane is blank, select **All** to see all of the connectors and triggers available.

3. Select **Service Bus** from the list of connectors.



4. A list of triggers is displayed. Select **When a message is received in a queue (auto-complete) / Service Bus**.

| |
|--|
| <input type="checkbox"/> When a message is received in a queue (auto-complete) Service Bus |
| <input type="checkbox"/> When a message is received in a queue (peek-lock) Service Bus |
| <input type="checkbox"/> When a message is received in a topic subscription (auto-complete) Service Bus |
| <input type="checkbox"/> When a message is received in a topic subscription (peek-lock) Service Bus |
| <input type="checkbox"/> When one or more messages arrive in a queue (auto-complete) Service Bus |
| <input type="checkbox"/> When one or more messages arrive in a queue (peek-lock) Service Bus |
| <input type="checkbox"/> When one or more messages arrive in a topic (auto-complete) Service Bus |
| <input type="checkbox"/> When one or more messages arrive in a topic (peek-lock) Service Bus |

5. On the next screen, fill in the Connection Name. This tutorial uses **ContosoConnection**.

| Name | Resource Group | Location |
|-------------------------|-------------------|-----------|
| ContosoSBNamespace5906 | ContosoResources | West US |
| ContosoSBNamespace32083 | ContosoResources2 | West US 2 |

Select 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 select **Create**.

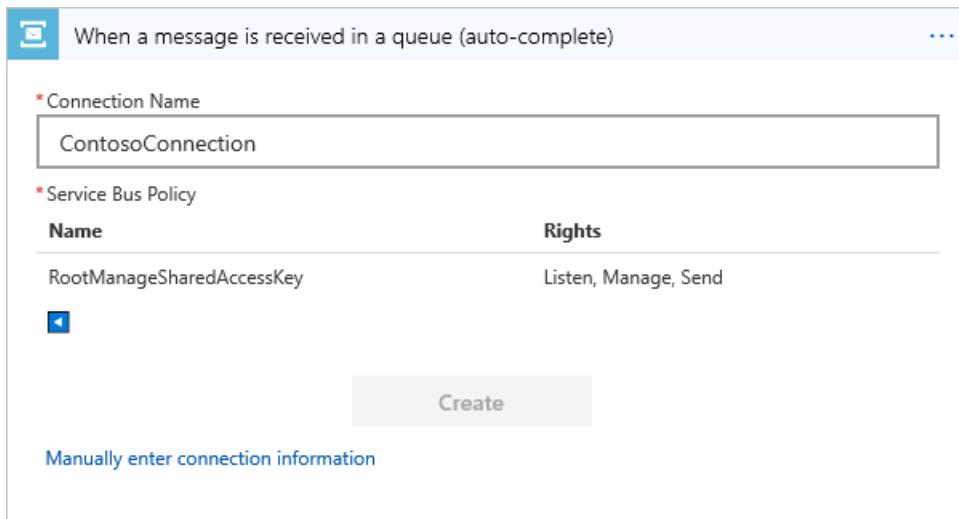
When a message is received in a queue (auto-complete) ...

* Connection Name
ContosoConnection

* Service Bus Policy

| Name | Rights |
|---------------------------|----------------------|
| RootManageSharedAccessKey | Listen, Manage, Send |

[Manually enter connection information](#)



6. 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.

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

* Queue name
contososbqueue5906

Queue type
Main

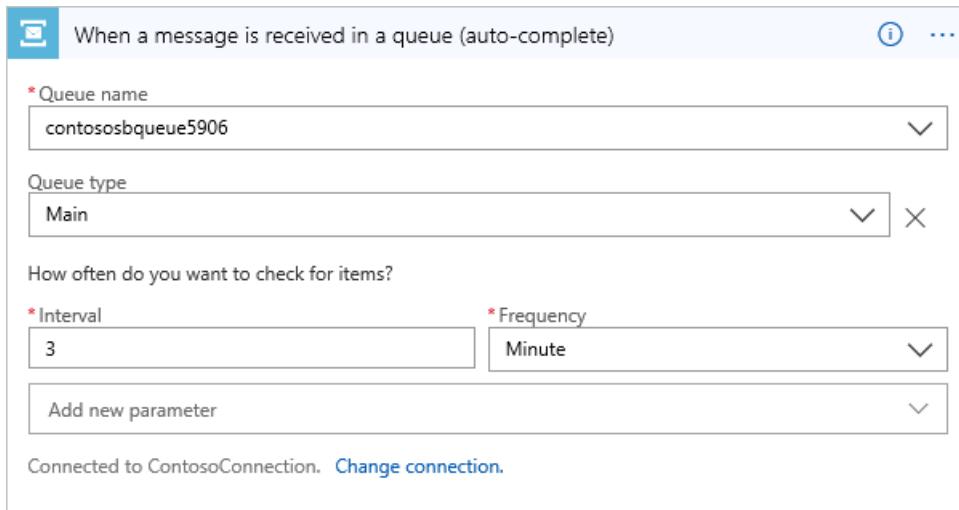
How often do you want to check for items?

* Interval
3

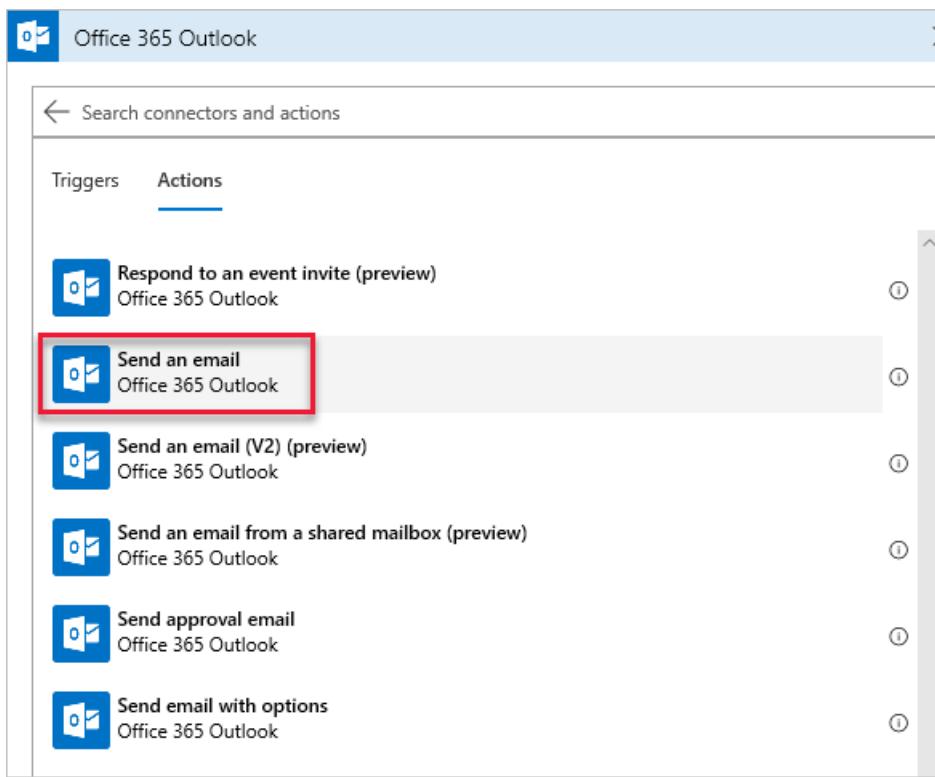
* Frequency
Minute

Add new parameter

Connected to ContosoConnection. [Change connection.](#)



7. Now set up the action to send an e-mail when a message is received in the queue. In the Logic Apps Designer, select **+ New step** to add a step, then select **All** to see all of the options available. In the **Choose an action** pane, find and select **Office 365 Outlook**. On the Actions screen, select **Send an e-mail / Office 365 Outlook**.



- Sign in to your Office 365 account to set up the connection. If this times out, just try again. 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.

Select **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 Logic App Designer with a 'Send an email' step selected. The 'Body' field contains dynamic content. A sidebar on the right shows the 'Dynamic content' section with various options like 'Content', 'Content Type', 'Lock Token', and 'Session Id'.

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

Send an email

*To: user@contoso.com

*Subject: Critical message from IoT Hub!

*Body: This is a critical message from an IoT device connected to your IoT Hub.
Content x

Add dynamic content

Show advanced options

Connected to user@contoso.com. Change connection.

+ New step

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

Dynamic content Expression

Search dynamic content

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

Content Content of the message

Content Type Content type of the message content

Lock Token The lock token of the message as a string.

Session Id Identifier of the session

- Select **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](#), select **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**.

Subscription: The Azure subscription you are using for the tutorial.

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 page. It includes fields for Job name (contosoJob), Subscription (Microsoft Azure Internal Consumption (a42f...)), Resource group (ContosoResources), Location (West US), Hosting environment (Edge selected), Streaming units (1 to 120) set to 3, and a 'Create' button at the bottom.

| | |
|----------------------------|--|
| Job name | contosoJob |
| Subscription | Microsoft Azure Internal Consumption (a42f...) |
| Resource group | ContosoResources |
| Location | West US |
| Hosting environment | Cloud Edge |
| Streaming units (1 to 120) | 3 |

3. Select **Create** to create the job. It may take a few minutes to deploy.

To get back to the job, select **Resource groups**. This tutorial uses **ContosoResources**. Select the resource group, then select the Stream Analytics job in the list of resources.

Add an input to the Stream Analytics job

1. Under **Job Topology**, select **Inputs**.

2. In the **Inputs** pane, select **Add stream input** and select IoT Hub. On the screen that comes up, fill in the following fields:

Input alias: This tutorial uses **contosoinputs**.

Select IoT Hub from your subscription: Select this radio button option.

Subscription: Select the Azure subscription you're using for this tutorial.

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 **service**. The portal fills in the Shared Access Policy Key for you.

Consumer group: Select the consumer group set up in Part 1 of this tutorial. 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 fields filled in are:

- * Input alias:** contosoinputs
- Subscription:** Internal use
- IoT Hub:** ContosoTestHub5906
- Endpoint:** Messaging
- Shared access policy name:** service
- Shared access policy key:** (redacted)
- Consumer group:** contosoconsumers
- * Event serialization format:** JSON
- Encoding:** UTF-8
- Event compression type:** None

A blue 'Save' button is at the bottom.

3. Select **Save**.

Add an output to the Stream Analytics job

1. Under **Job Topology**, select **Outputs**.
2. In the **Outputs** pane, select **Add**, and then select **Power BI**. On the screen that comes up, fill in the following fields:

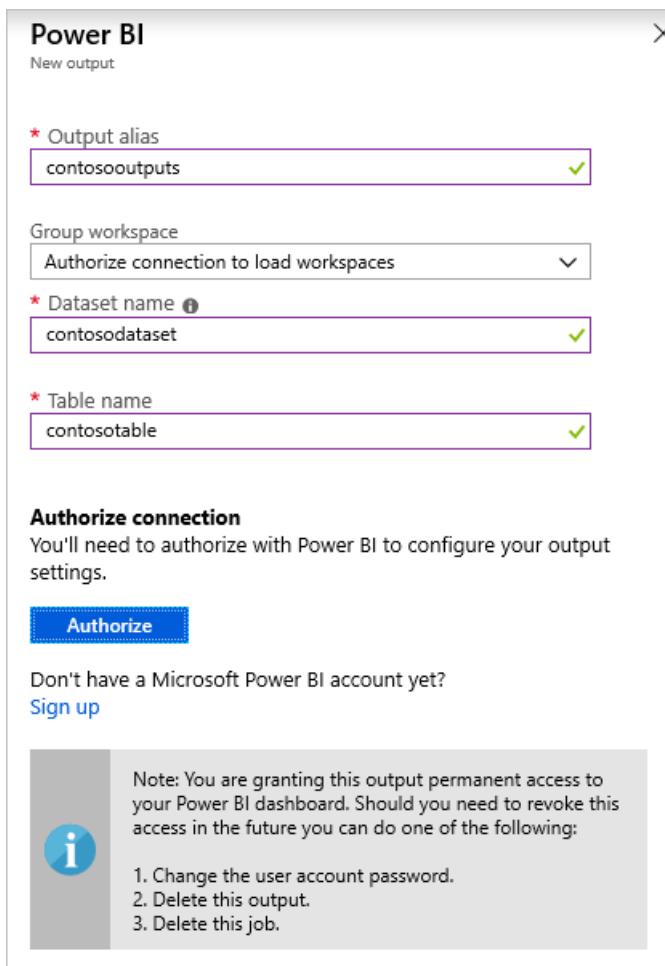
Output alias: The unique alias for the output. This tutorial uses **contosooutputs**.

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 **contosotable**.

Accept the defaults for the rest of the fields.

3. Select **Authorize**, and sign in to your Power BI account. (This may take more than one try).



4. Select **Save**.

Configure the query of the Stream Analytics job

1. Under **Job Topology**, select **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`.

4. Select **Save**.
5. Close the Query pane. You return to the view of the resources in the Resource Group. Select the Stream Analytics job. This tutorial calls it `contosoJob`.

Run the Stream Analytics job

In the Stream Analytics job, select **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

In Part 1 of this tutorial, you set up a device to simulate using an IoT device. In this section, you download the .NET

console app that simulates that device sending device-to-cloud messages to an IoT hub (assuming you didn't already download the app and resources in Part 1).

This application sends messages for each of the different message routing methods. There is also a folder in the download that contains the complete Azure Resource Manager template and parameters file, as well as the Azure CLI and PowerShell scripts.

If you didn't download the files from the repository in Part 1 of this tutorial, go ahead and download them now from [IoT Device Simulation](#). Selecting this link downloads a repository with several applications in it; the solution you are looking for is `iot-hub/Tutorials/Routing/IoT_SimulatedDevice.sln`.

Double-click on the solution file (`IoT_SimulatedDevice.sln`) to open the code in Visual Studio, then open `Program.cs`. Substitute `{your hub name}` 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 `{your device key}` with the device key you saved earlier when setting up the simulated device.

```
static string s_myDeviceId = "Contoso-Test-Device";
static string s_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 s_deviceKey = "{your device key}";
```

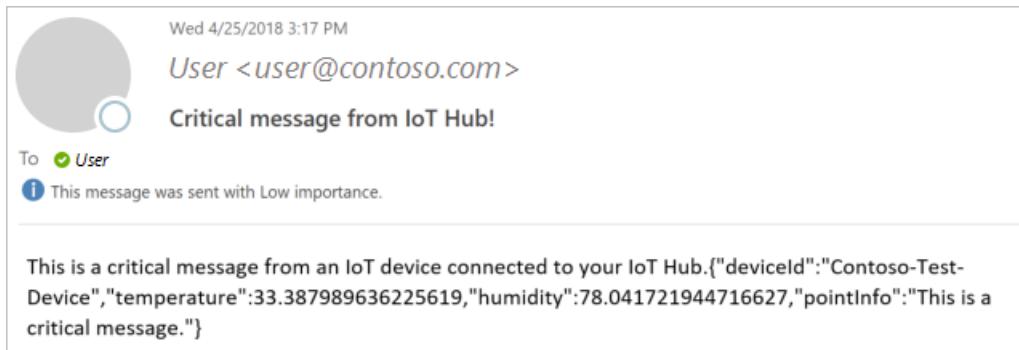
Run and test

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 result means the following statements are true.

- 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](#), select **Resource groups** and select your Resource Group. This tutorial uses **ContosoResources**.

Select the storage account, select **Containers**, 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 Explorer interface. On the left, there's a sidebar with options like Overview, Access Control (IAM), Settings, Access policy, Properties, Metadata, and Editor (preview). The main area displays a table of blobs. The table has columns for NAME, MODIFIED, ACCE..., BLOB TYPE, SIZE, and LEASE STATE. There are four rows: one for a folder named '[.]' and three for files named '26', '31', and '36'. Each row includes a '...' button. At the top of the main area, it says 'Authentication method: Access key (Switch to Azure AD User Account)' and 'Location: contosoresults / ContosoTestHub5906 / 00 / 2019 / 03 / 16 / 22'. A search bar at the top right says 'Search blobs by prefix (case-sensitive)' and a checkbox says 'Show deleted blobs'.

This result means the following statement is true.

- 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. Select **Datasets**. If you don't have any datasets, wait a few minutes and check again.

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**, select the first icon to create a report.

The screenshot shows the Power BI workspace. The left sidebar has links for Home (preview), Favorites, Recent, Apps, and Shared with me. The main area has tabs for Dashboards, Reports, Workbooks, and Datasets. Below these tabs, there's a table of datasets. The table has columns for NAME ↑ and ACTIONS. One dataset, 'contosodataset', is listed. The 'ACTIONS' column for this dataset has a red box around it. Other icons in the ACTIONS column include a bar chart, a pencil, a delete icon, and an ellipsis.

5. Create a line chart to show real-time temperature over time.

- On the report creation page, add a line chart by selecting 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 process for the first chart, placing **EventEnqueuedUtcTime** on the x-axis (**Axis**) and **humidity** on the y-axis (**Values**).



7. Select **Save** to save the report, entering a name for the report if prompted.

You should be able to see data on both charts. This result means the following statements are true:

- 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 selecting the Refresh button on the top of the Power BI window.

Clean up resources

If you want to remove all of the Azure resources you've created through both parts of this tutorial, 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. You can also remove the Power BI resources and clear the emails sent during the tutorial.

Clean up resources in the Power BI visualization

Sign in to your [Power BI](#) account. Go to your workspace. This tutorial uses **My Workspace**. To remove the Power BI visualization, go to DataSets and select the trash can icon to delete the dataset. This tutorial uses **contosodataset**. When you remove the dataset, the report is removed as well.

Use the Azure CLI to clean up resources

To remove the resource group, use the [az group delete](#) command. `$resourceGroup` was set to **ContosoResources** back at the beginning of this tutorial.

```
az group delete --name $resourceGroup
```

Use PowerShell to clean up resources

To remove the resource group, use the [Remove-AzResourceGroup](#) command. `$resourceGroup` was set to **ContosoResources** back at the beginning of this tutorial.

```
Remove-AzResourceGroup -Name $resourceGroup
```

Clean up test emails

You may also want to delete the quantity of emails in your inbox that were generated through the Logic App while the device application was running.

Next steps

In this 2-part tutorial, you learned how to use message routing to route IoT Hub messages to different destinations by performing the following tasks.

Part I: Create resources, set up message routing

- Create the resources -- an IoT hub, a storage account, a Service Bus queue, and a simulated device.
- Configure the endpoints and message routes in IoT Hub for the storage account and Service Bus queue.

Part II: Send messages to the hub, view routed results

- 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: Use Azure IoT Hub message enrichments

4/21/2020 • 17 minutes to read • [Edit Online](#)

Message enrichments describes the ability of Azure IoT Hub to *stamp* messages with additional information before the messages are sent to the designated endpoint. One reason to use message enrichments is to include data that can be used to simplify downstream processing. For example, enriching device telemetry messages with a device twin tag can reduce load on customers to make device twin API calls for this information. For more information, see [Overview of message enrichments](#).

In this tutorial, you see two ways to create and configure the resources that are needed to test the message enrichments for an IoT hub. The resources include one storage account with two storage containers. One container holds the enriched messages, and another container holds the original messages. Also included is an IoT hub to receive the messages and route them to the appropriate storage container based on whether they're enriched or not.

- The first method is to use the Azure CLI to create the resources and configure the message routing. Then you define the enrichments manually by using the [Azure portal](#).
- The second method is to use an Azure Resource Manager template to create both the resources *and* the configurations for the message routing and message enrichments.

After the configurations for the message routing and message enrichments are finished, you use an application to send messages to the IoT hub. The hub then routes them to both storage containers. Only the messages sent to the endpoint for the **enriched** storage container are enriched.

Here are the tasks you perform to complete this tutorial:

Use IoT Hub message enrichments

- First method: Create resources and configure message routing by using the Azure CLI. Configure the message enrichments manually by using the [Azure portal](#).
- Second method: Create resources and configure message routing and message enrichments by using a Resource Manager template.
- Run an app that simulates an IoT device sending messages to the hub.
- View the results, and verify that the message enrichments are working as expected.

Prerequisites

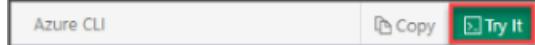
- You must have an Azure subscription. If you don't have an Azure subscription, create a [free account](#) before you begin.
- Install [Visual Studio](#).
- Make sure that port 8883 is open in your firewall. The device sample in this tutorial uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell

preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|--|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal . |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the Copy button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

Retrieve the IoT C# samples repository

Download the [IoT C# samples](#) from GitHub and unzip them. This repository has several applications, scripts, and Resource Manager templates in it. The ones to be used for this tutorial are as follows:

- For the manual method, there's a CLI script that's used to create the resources. This script is in /azure-iot-samples-csharp/iot-hub/Tutorials/Routing/SimulatedDevice/resources/iothub_msge enrichment_cli.azcli. This script creates the resources and configures the message routing. After you run this script, create the message enrichments manually by using the [Azure portal](#).
- For the automated method, there's an Azure Resource Manager template. The template is in /azure-iot-samples-csharp/iot-hub/Tutorials/Routing/SimulatedDevice/resources/template_msge enrichments.json. This template creates the resources, configures the message routing, and then configures the message enrichments.
- The third application you use is the Device Simulation app, which you use to send messages to the IoT hub and test the message enrichments.

Manually set up and configure by using the Azure CLI

In addition to creating the necessary resources, the Azure CLI script also configures the two routes to the endpoints that are separate storage containers. For more information on how to configure the message routing, see the [Routing tutorial](#). After the resources are set up, use the [Azure portal](#) to configure message enrichments for each endpoint. Then continue on to the testing step.

NOTE

All messages are routed to both endpoints, but only the messages going to the endpoint with configured message enrichments will be enriched.

You can use the script that follows, or you can open the script in the /resources folder of the downloaded

repository. The script performs the following steps:

- Create an IoT hub.
- Create a storage account.
- Create two containers in the storage account. One container is for the enriched messages, and another container is for messages that aren't enriched.
- Set up routing for the two different storage accounts:
 - Create an endpoint for each storage account container.
 - Create a route to each of the storage account container endpoints.

There are several resource names that must be globally unique, such as the IoT hub name and the storage account name. To make running the script easier, those resource names are appended with a random alphanumeric value called *randomValue*. The random value is generated once at the top of the script. It's appended to the resource names as needed throughout the script. If you don't want the value to be random, you can set it to an empty string or to a specific value.

If you haven't already done so, open an Azure [Cloud Shell window](#) and ensure that it's set to Bash. Open the script in the unzipped repository, select Ctrl+A to select all of it, and then select Ctrl+C to copy it. Alternatively, you can copy the following CLI script or open it directly in Cloud Shell. Paste the script in the Cloud Shell window by right-clicking the command line and selecting **Paste**. The script runs one statement at a time. After the script stops running, select **Enter** to make sure it runs the last command. The following code block shows the script that's used, with comments that explain what it's doing.

Here are the resources created by the script. *Enriched* means that the resource is for messages with enrichments. *Original* means that the resource is for messages that aren't enriched.

| NAME | VALUE |
|----------------------|--------------------------------|
| resourceGroup | ContosoResourcesMsgEn |
| container name | original |
| container name | enriched |
| IoT device name | Contoso-Test-Device |
| IoT Hub name | ContosoTestHubMsgEn |
| storage Account Name | contosostorage |
| endpoint Name 1 | ContosoStorageEndpointOriginal |
| endpoint Name 2 | ContosoStorageEndpointEnriched |
| route Name 1 | ContosoStorageRouteOriginal |
| route Name 2 | ContosoStorageRouteEnriched |

```
# This command retrieves the subscription id of the current Azure account.
# This field is used when setting up the routing queries.
subscriptionID=$(az account show --query id -o tsv)

# Concatenate this number onto the resources that have to be globally unique.
# You can set this to "" or to a specific value if you don't want it to be random.
# This retrieves a random value.
```

```
randomValue=$RANDOM

# This command installs 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-iot

# Set the values for the resource names that
# don't have to be globally unique.
location=westus2
resourceGroup=ContosoResourcesMsgEn
containerName1=original
containerName2=enriched
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 value to the end.
iotHubName=ContosoTestHubMsgEn$randomValue
echo "IoT hub name = " $iotHubName

# Create the IoT hub.
az iot hub create --name $iotHubName \
--resource-group $resourceGroup \
--sku S1 --location $location

# You need a storage account that will have two containers
# -- one for the original messages and
# one for the enriched messages.
# The storage account name must be globally unique,
# so add a random value to the end.
storageAccountName=contosostorage$randomValue
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 containers.
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 "storage account key = " $storageAccountKey

# Create the containers in the storage account.
az storage container create --name $containerName1 \
--account-name $storageAccountName \
--account-key $storageAccountKey \
--public-access off

az storage container create --name $containerName2 \
--account-name $storageAccountName \
--account-key $storageAccountKey \
--public-access off

# 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.
# If you are using Cloud Shell, you can scroll the window back up to retrieve this value.
az iot hub device-identity show --device-id $iotDeviceName \
    --hub-name $iotHubName

##### ROUTING FOR STORAGE #####
# You're going to have two routes and two endpoints.
# One route points to the first container ("original") in the storage account
# and includes the original messages.
# The other points to the second container ("enriched") in the same storage account
# and includes the enriched versions of the messages.

endpointType="azurestoragecontainer"
endpointName1="ContosoStorageEndpointOriginal"
endpointName2="ContosoStorageEndpointEnriched"
routeName1="ContosoStorageRouteOriginal"
routeName2="ContosoStorageRouteEnriched"

# for both endpoints, retrieve the messages going to storage
condition='level="storage"'

# Get the connection string for the storage account.
# Adding the "-o tsv" makes it be returned without the default double quotes around it.
storageConnectionString=$(az storage account show-connection-string \
    --name $storageAccountName --query connectionString -o tsv)

# Create the routing endpoints and routes.
# Set the encoding format to either avro or json.

# This is the endpoint for the first container, for endpoint messages that are not enriched.
az iot hub routing-endpoint create \
    --connection-string $storageConnectionString \
    --endpoint-name $endpointName1 \
    --endpoint-resource-group $resourceGroup \
    --endpoint-subscription-id $subscriptionID \
    --endpoint-type $endpointType \
    --hub-name $iotHubName \
    --container $containerName1 \
    --resource-group $resourceGroup \
    --encoding json

# This is the endpoint for the second container, for endpoint messages that are enriched.
az iot hub routing-endpoint create \
    --connection-string $storageConnectionString \
    --endpoint-name $endpointName2 \
    --endpoint-resource-group $resourceGroup \
    --endpoint-subscription-id $subscriptionID \
    --endpoint-type $endpointType \
    --hub-name $iotHubName \
    --container $containerName2 \
    --resource-group $resourceGroup \
    --encoding json

# This is the route for messages that are not enriched.
# Create the route for the first storage endpoint.
az iot hub route create \
    --name $routeName1 \
    --hub-name $iotHubName \
    --source devicemessages \
    --resource-group $resourceGroup \
    --endpoint-name $endpointName1 \
    --enabled \
    --condition $condition

# This is the route for messages that are enriched.
# Create the route for the second storage endpoint.
az iot hub route create \

```

```
az iot route create --name $routeName2 \
--hub-name $iotHubName \
--source devicemessages \
--resource-group $resourceGroup \
--endpoint-name $endpointName2 \
--enabled \
--condition $condition
```

At this point, the resources are all set up and the message routing is configured. You can view the message routing configuration in the portal and set up the message enrichments for messages going to the **enriched** storage container.

Manually configure the message enrichments by using the Azure portal

1. Go to your IoT hub by selecting **Resource groups**. Then select the resource group set up for this tutorial (**ContosoResourcesMsgEn**). Find the IoT hub in the list, and select it. Select **Message routing** for the IoT hub.

The screenshot shows the Azure IoT Hub configuration interface for the hub 'ContosoTestHubMsgEn3276'. The left sidebar contains navigation links for Shared access policies, Pricing and scale, IP Filter, Certificates, Built-in endpoints, Properties, Locks, Export template, Query explorer, IoT devices, IoT Edge, and IoT device configuration. The 'Message routing' link is highlighted with a red box. The main content area displays basic hub details: Resource group (ContosoResourcesMsgEn), Status (Active), Location (West US 2), Subscription (IoTHubLibrary_1), and Tags (Click here to add tags). There are also two promotional cards: one for device provisioning and another for learning more about IoT Hub documentation.

The message routing pane has three tabs labeled **Routes**, **Custom endpoints**, and **Enrich messages**. Browse the first two tabs to see the configuration set up by the script. Use the third tab to add message enrichments. Let's enrich messages going to the endpoint for the storage container called **enriched**. Fill in the name and value, and then select the endpoint **ContosoStorageEndpointEnriched** from the drop-down list. Here's an example of how to set up an enrichment that adds the IoT hub name to the message:

Send data from your devices to endpoints that you choose.

Routes Custom endpoints **Enrich messages**

Add up to 10 message enrichments per IoT Hub. These are added as application properties to messages sent to chosen endpoint(s). [Learn more](#)

Value can be any string. Additionally, you may use a value to stamp the IoT Hub name (for example, \$iothubname) or information from the device twin (for example, \$twin.tags.field or \$twin.properties.desired.value)

| NAME | VALUE | ENDPOINT(S) |
|----------|--------------|--------------------------------|
| myIoTHub | \$iothubname | ContosoStorageEndpointEnriched |

Apply

Built-in endpoints
 events

Azure Storage containers
 ContosoStorageEndpointEnriched
 ContosoStorageEndpointOriginal

Event Hubs
Service Bus queues
Service Bus topics

2. Add these values to the list for the ContosoStorageEndpointEnriched endpoint.

| KEY | VALUE | ENDPOINT (DROP-DOWN LIST) |
|----------------|--------------------------------------|---|
| myIoTHub | \$iothubname | AzureStorageContainers > ContosoStorageEndpointEnriched |
| DeviceLocation | \$twin.tags.location | AzureStorageContainers > ContosoStorageEndpointEnriched |
| customerID | 6ce345b8-1e4a-411e-9398-d34587459a3a | AzureStorageContainers > ContosoStorageEndpointEnriched |

NOTE

If your device doesn't have a twin, the value you put in here will be stamped as a string for the value in the message enrichments. To see the device twin information, go to your hub in the portal and select **IoT devices**. Select your device, and then select **Device twin** at the top of the page.

You can edit the twin information to add tags, such as location, and set it to a specific value. For more information, see [Understand and use device twins in IoT Hub](#).

3. When you're finished, your pane should look similar to this image:

Send data from your devices to endpoints that you choose.

Routes Custom endpoints Enrich messages

Add up to 10 message enrichments per IoT Hub. These are added as application properties to messages sent to chosen endpoint(s). [Learn more](#)

Value can be any string. Additionally, you may use a value to stamp the IoT Hub name (for example, \$iothubname) or information from the device twin (for example, \$twin.tags.location or \$twin.properties.desired.value)

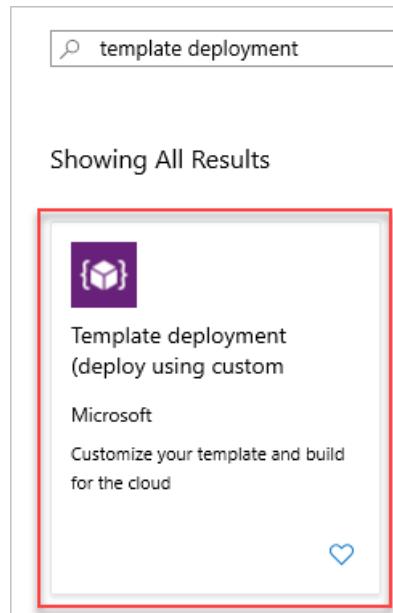
| NAME | VALUE | ENDPOINT(S) |
|--------------------------------------|--------------------------------------|---------------------------------|
| myIoTHub | \$iothubname | ContosoStorageEndpointEnrich... |
| device location | \$twin.tags.location | ContosoStorageEndpointEnrich... |
| customerID | 6ce345b8-1e4a-411e-9398-d34587459a3a | ContosoStorageEndpointEnrich... |
| | | |
| | | 0 selected |
| <input type="button" value="Apply"/> | | |

4. Select **Apply** to save the changes. Skip to the [Test message enrichments](#) section.

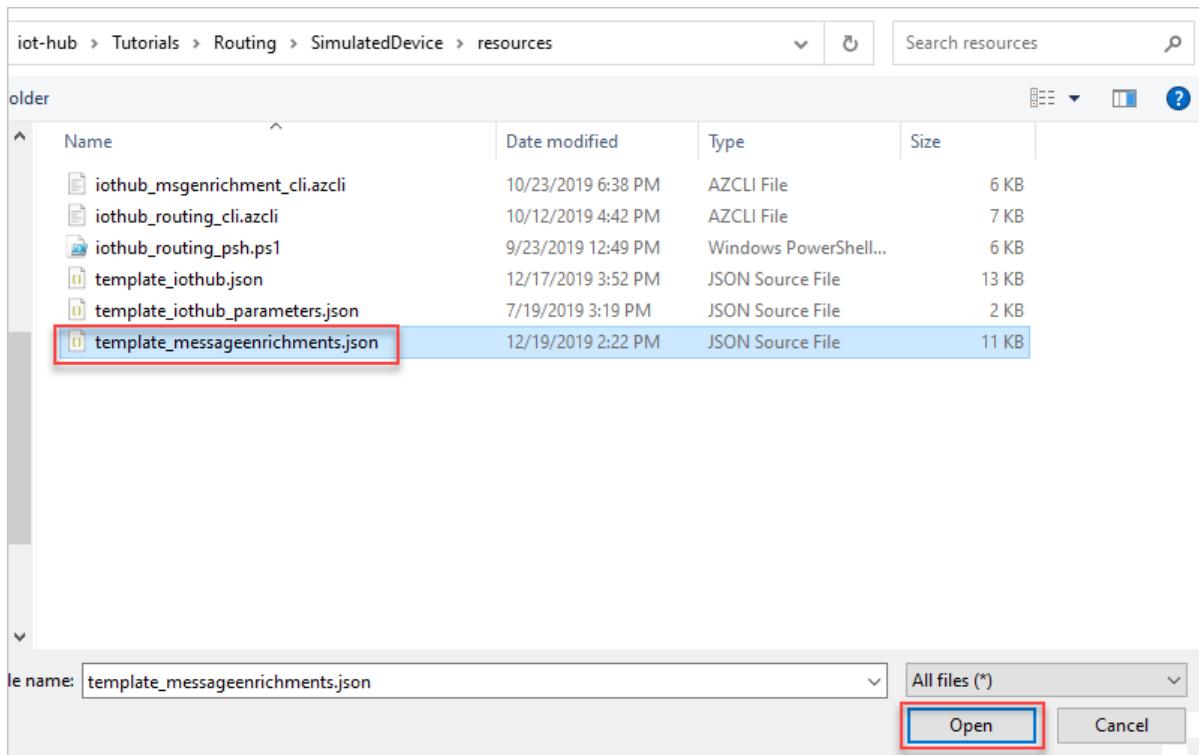
Create and configure by using a Resource Manager template

You can use a Resource Manager template to create and configure the resources, message routing, and message enrichments.

1. Sign in to the Azure portal. Select **+ Create a Resource** to bring up a search box. Enter *template deployment*, and search for it. In the results pane, select **Template deployment (deploy using custom template)**.



2. Select **Create** in the **Template deployment** pane.
3. In the **Custom deployment** pane, select **Build your own template in the editor**.
4. In the **Edit template** pane, select **Load file**. Windows Explorer appears. Locate the **template_messageenrichments.json** file in the unzipped repo file in **/iot-hub/Tutorials/Routing/SimulatedDevice/resources**.



5. Select **Open** to load the template file from the local machine. It loads and appears in the edit pane.

This template is set up to use a globally unique IoT hub name and storage account name by adding a random value to the end of the default names, so you can use the template without making any changes to it.

Here are the resources created by loading the template. **Enriched** means that the resource is for messages with enrichments. **Original** means that the resource is for messages that aren't enriched. These are the same values used in the Azure CLI script.

| NAME | VALUE |
|----------------------|--------------------------------|
| resourceGroup | ContosoResourcesMsgEn |
| container name | original |
| container name | enriched |
| IoT device name | Contoso-Test-Device |
| IoT Hub name | ContosoTestHubMsgEn |
| storage Account Name | contosostorage |
| endpoint Name 1 | ContosoStorageEndpointOriginal |
| endpoint Name 2 | ContosoStorageEndpointEnriched |
| route Name 1 | ContosoStorageRouteOriginal |
| route Name 2 | ContosoStorageRouteEnriched |

6. Select **Save**. The **Custom deployment** pane appears and shows all of the parameters used by the template. The only field you need to set is **Resource group**. Either create a new one or select one from the

drop-down list.

Here's the top half of the **Custom deployment** pane. You can see where you fill in the resource group.

The screenshot shows the 'Custom deployment' pane in the Azure portal. At the top, there's a breadcrumb navigation: Dashboard > New > Marketplace > Template deployment (deploy using custom templates) > Custom deployment. Below the breadcrumb, the title 'Custom deployment' is displayed with a subtitle 'Deploy from a custom template'. The 'TEMPLATE' section shows a 'Customized template' icon, '2 resources', and three buttons: 'Edit template', 'Edit paramet...', and 'Learn more'. The 'BASICS' section contains fields for 'Subscription *' (Content Developer testing (robinsh)), 'Resource group *' (TestNew, with a 'Create new' link), and 'Location' ((US) West US 2). The 'SETTINGS' section lists several parameters with their values: Random Value (substring(uniqueString(utcNow()),0,8)), Subscription Id ([subscription().subscriptionid]), Io T Hub Name_in (ContosoTestHubMsgEn), Location (westus2), Sku_name (S1), Sku_units (1), and D2c_partitions (4). The 'Resource group *' field is highlighted with a red box.

7. Here's the bottom half of the **Custom deployment** pane. You can see the rest of the parameters and the terms and conditions.

| | |
|------------------|--------------------------------|
| Route Name1 ⓘ | ContosoStorageRouteOriginal |
| Route Name2 ⓘ | ContosoStorageRouteEnriched |
| Endpoint Name1 ⓘ | ContosoStorageEndpointOriginal |
| Endpoint Name2 ⓘ | ContosoStorageEndpointEnriched |

TERMS AND CONDITIONS

Azure Marketplace Terms | Azure Marketplace

By clicking "Purchase," I (a) agree to the applicable legal terms associated with the offering; (b) authorize Microsoft to charge or bill my current payment method for the fees associated the offering(s), including applicable taxes, with the same billing frequency as my Azure subscription, until I discontinue use of the offering(s); and (c) agree that, if the deployment involves 3rd party offerings, Microsoft may share my contact information and other details of such deployment with the publisher of that offering.

I agree to the terms and conditions stated above

Purchase

8. Select the check box to agree to the terms and conditions. Then select **Purchase** to continue with the template deployment.
9. Wait for the template to be fully deployed. Select the bell icon at the top of the screen to check on the progress. When it's finished, continue to the [Test message enrichments](#) section.

Test message enrichments

To view the message enrichments, select **Resource groups**. Then select the resource group you're using for this tutorial. Select the IoT hub from the list of resources, and go to **Messaging**. The message routing configuration and the configured enrichments appear.

Now that the message enrichments are configured for the endpoint, run the Simulated Device application to send messages to the IoT hub. The hub was set up with settings that accomplish the following tasks:

- Messages routed to the storage endpoint ContosoStorageEndpointOriginal won't be enriched and will be stored in the storage container `original`.
- Messages routed to the storage endpoint ContosoStorageEndpointEnriched will be enriched and stored in the storage container `enriched`.

The Simulated Device application is one of the applications in the unzipped download. The application sends messages for each of the different message routing methods in the [Routing tutorial](#), which includes Azure Storage.

Double-click the solution file `IoT_SimulatedDevice.sln` to open the code in Visual Studio, and then open `Program.cs`. Substitute the IoT hub name for the marker `{your hub name}`. The format of the IoT hub host name is `{your hub name}.azure-devices.net`. For this tutorial, the hub host name is `ContosoTestHubMsgEn.azure-devices.net`. Next, substitute the device key you saved earlier when you ran the script to create the resources for the marker `{your device key}`.

If you don't have the device key, you can retrieve it from the portal. After you sign in, go to **Resource groups**, select your resource group, and then select your IoT hub. Look under **IoT Devices** for your test device, and select your device. Select the copy icon next to **Primary key** to copy it to the clipboard.

```

private readonly static string s_myDeviceId = "Contoso-Test-Device";
private readonly static string s_iotHubUri = "ContosoTestHubMsgEn.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.
private readonly static string s_deviceKey = "{your device key}";

```

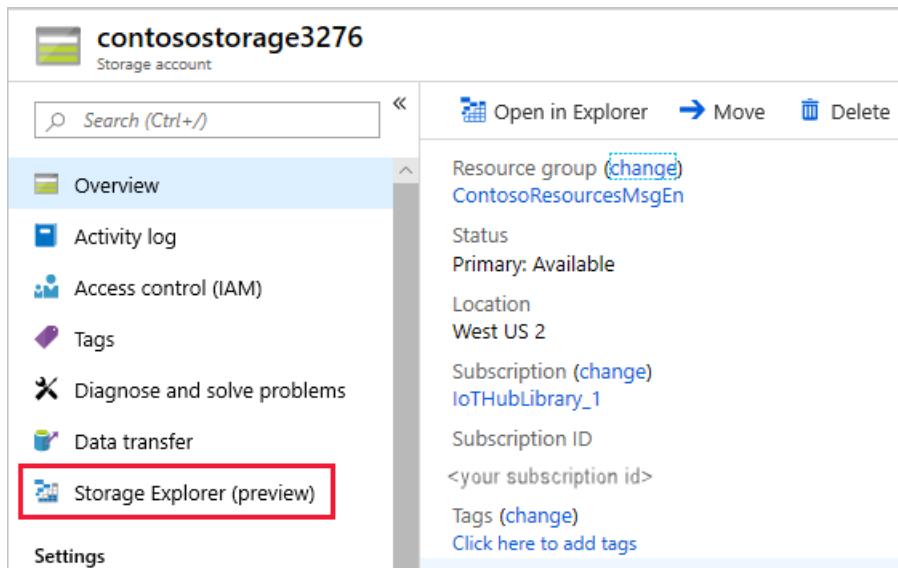
Run and test

Run the console application for a few minutes. The messages that are being sent are displayed on the console screen of the application.

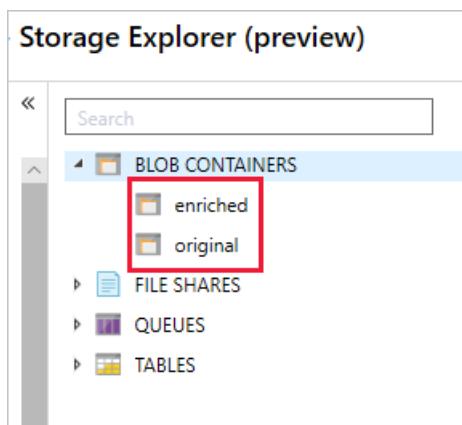
The app sends a new device-to-cloud message to the IoT hub every second. The message contains a JSON-serialized object with the device ID, temperature, humidity, and message level, which defaults to `normal`. It randomly assigns a level of `critical` or `storage`, which causes the message to be routed to the storage account or to the default endpoint. The messages sent to the **enriched** container in the storage account will be enriched.

After several storage messages are sent, view the data.

1. Select **Resource groups**. Find your resource group, **ContosoResourcesMsgEn**, and select it.
2. Select your storage account, which is **contosostorage**. Then select **Storage Explorer (preview)** in the left pane.



Select **BLOB CONTAINERS** to see the two containers that can be used.



The messages in the container called **enriched** have the message enrichments included in the messages. The messages in the container called **original** have the raw messages with no enrichments. Drill down into one of the containers until you get to the bottom, and open the most recent message file. Then do the same for the other container to verify that there are no enrichments added to messages in that container.

When you look at messages that have been enriched, you should see "my IoT Hub" with the hub name and the location and the customer ID, like this:

```
{"EnqueuedTimeUtc":"2019-05-10T06:06:32.722000Z", "Properties": {"level": "storage", "my IoT Hub": "contosotesthubmsgen3276", "devicelocation": "$twin.tags.location", "customerID": "6ce345b8-1e4a-411e-9398-d34587459a3a"}, "SystemProperties": {"connectionDeviceId": "Contoso-Test-Device", "connectionAuthMethod": {"\\"scope\\": \"device\", \\"type\\": \"sas\", \\"issuer\\": \"iothub\", \\"acceptingIpFilterRule\\": \"null\"}, "connectionDeviceGenerationId": "636930642531278483", "enqueuedTime": "2019-05-10T06:06:32.722000Z"}, "Body": "eyJkZXZpY2VJZCI6IkNvbnRvc28tVGVzdC1EZXZpY2UiLCJ0ZW1wZXJhdHVyZSI6MjkuMjMyMDE2ODQ4MDQyNjE1LCJodW1pZG10eSI6NjQuMzA1MzQ5NjkyODQ0NDg3LCJwb2ludEluZm8iOiJuaGlzIGlzIGEgc3RvcmlhZSBlIj9"}}
```

Here's an unenriched message. Notice that "my IoT Hub," "devicelocation," and "customerID" don't show up here because these fields are added by the enrichments. This endpoint has no enrichments.

```
{"EnqueuedTimeUtc": "2019-05-10T06:06:32.722000Z", "Properties": {"level": "storage"}, "SystemProperties": {"connectionDeviceId": "Contoso-Test-Device", "connectionAuthMethod": "", "\scope": "device", "\type": "sas", "\issuer": "iothub", "\acceptingIpFilterRule": null}, "connectionDeviceGenerationId": "636930642531278483", "enqueuedTime": "2019-05-10T06:06:32.722000Z"}, "Body": "eyJkZXZpY2VJZCI6IkNvbnRvc28tVGVzdC1EZXZpY2UiLCJ0ZW1wZXJhdHVyZSI6MjkuMjMyMDE2ODQ4MDQyNjE1LCJodW1pZG10eSI6NjQuMzA1MzQ5NjkyODQ0NDg3LCJwb2ludEluZm8iOiJuGaLzIGlzIGEgc3RvcmFnZSBtZXNzYWdlIj9"}
```

Clean up resources

To remove all of the resources you 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.

Use the Azure CLI to clean up resources

To remove the resource group, use the [az group delete](#) command. Recall that `$resourceGroup` was set to `ContosoResourcesMsqEn` at the beginning of this tutorial.

```
az group delete --name $resourceGroup
```

Next steps

In this tutorial, you configured and tested adding message enrichments to IoT Hub messages by using the following steps:

Use IoT Hub message enrichments

- First method: Create resources and configure message routing by using the Azure CLI. Configure the message enrichments manually by using the [Azure portal](#).
 - Second method: Create resources and configure message routing and message enrichments by using an Azure Resource Manager template.
 - Run an app that simulates an IoT device sending messages to the hub.
 - View the results, and verify that the message enrichments are working as expected.

For more information about message enrichments, see [Overview of message enrichments](#).

For more information about message routing, see these articles:

- Use IoT Hub message routing to send device-to-cloud messages to different endpoints
 - Tutorial: IoT Hub routing

Tutorial: Set up and use metrics and diagnostic logs with an IoT hub

4/21/2020 • 13 minutes to read • [Edit Online](#)

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.
- Make sure that port 8883 is open in your firewall. The device sample in this tutorial uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|--|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal. |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the Copy button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

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-iot

# 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-iot
```

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 the 'ContosoTestHub' IoT Hub. The left sidebar contains several sections: '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, Logs). The 'Diagnostic settings' option is highlighted with a red box.

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' (set to 'Microsoft Azure Internal Consumption'), 'Resource group' (set to 'ContosoResources'), 'Resource type' (set to 'IoT Hub'), and 'Resource' (set to 'ContosoTestHub'). Below these fields is a link to '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**

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: contosostoragemon

Stream to an event hub

Send to Log Analytics

LOG

| | |
|---|---|
| <input checked="" type="checkbox"/> Connections | Retention (days) <input type="text" value="7"/> |
| <input checked="" type="checkbox"/> DeviceTelemetry | Retention (days) <input type="text" value="7"/> |
| Retention (days) <input type="text"/> | |

- 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.

- In the settings pane for the IoT hub, click on the **Metrics** option in the **Monitoring** section.
- 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

Last 24 hours (Automatic)

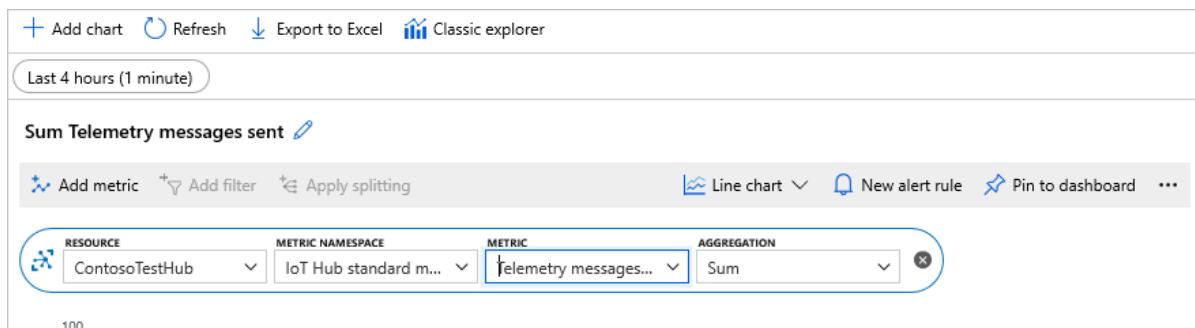
| | | |
|---|-------------------------------------|----------|
| Time range | Time granularity | |
| <input type="radio"/> Last 30 minutes | <input type="radio"/> Last 48 hours | 1 minute |
| <input type="radio"/> Last hour | <input type="radio"/> Last 3 days | |
| <input checked="" type="radio"/> Last 4 hours | <input type="radio"/> Last 7 days | |
| <input type="radio"/> Last 12 hours | <input type="radio"/> Last 30 days | |
| <input type="radio"/> Last 24 hours | <input type="radio"/> Custom | |

Show time as

UTC/GMT
 Local

Apply Cancel

- 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**.



4. 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*.



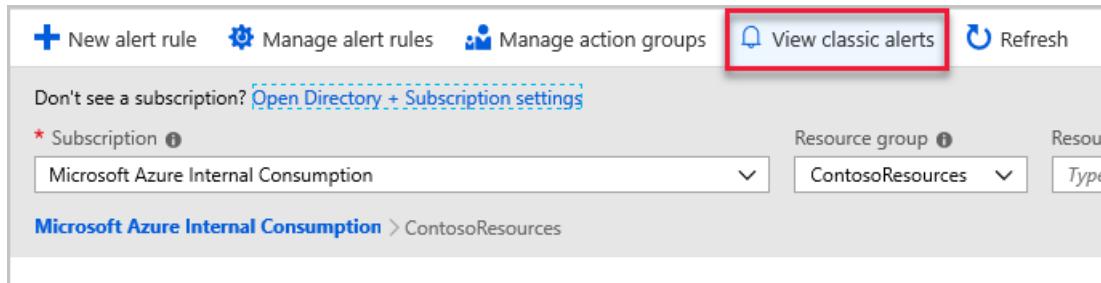
Click **Pin to dashboard**. It will pin it to the dashboard of your Azure portal so you can access it again. If you don't pin it to the dashboard, your settings are not retained.

Set up alerts

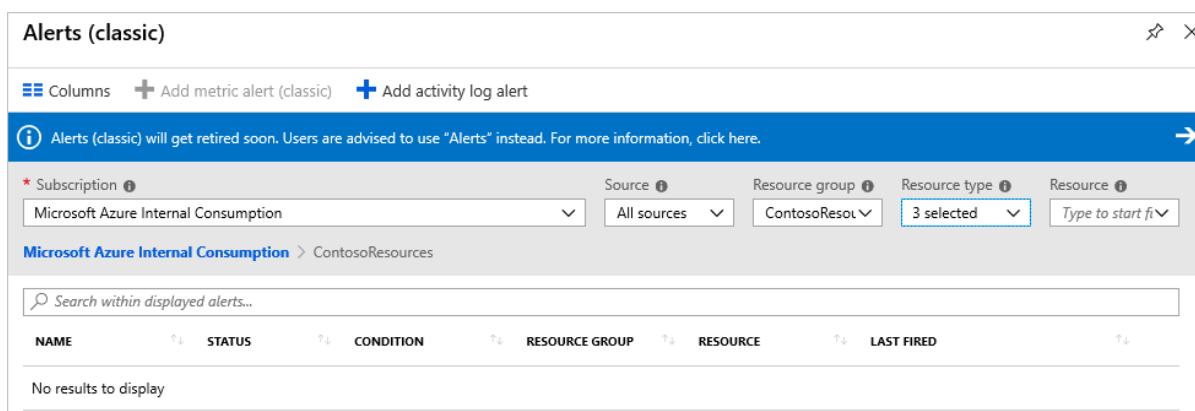
Go to the hub in the portal. Click **Resource Groups**, select *ContosoResources*, then select IoT Hub *ContosoTestHub*.

IoT Hub has not been migrated to the [metrics in Azure Monitor](#) yet; you have to use [classic alerts](#).

1. Under **Monitoring**, click **Alerts**. This shows the main alert screen.



2. 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 (count)

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

Learn more about configuring webhooks

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

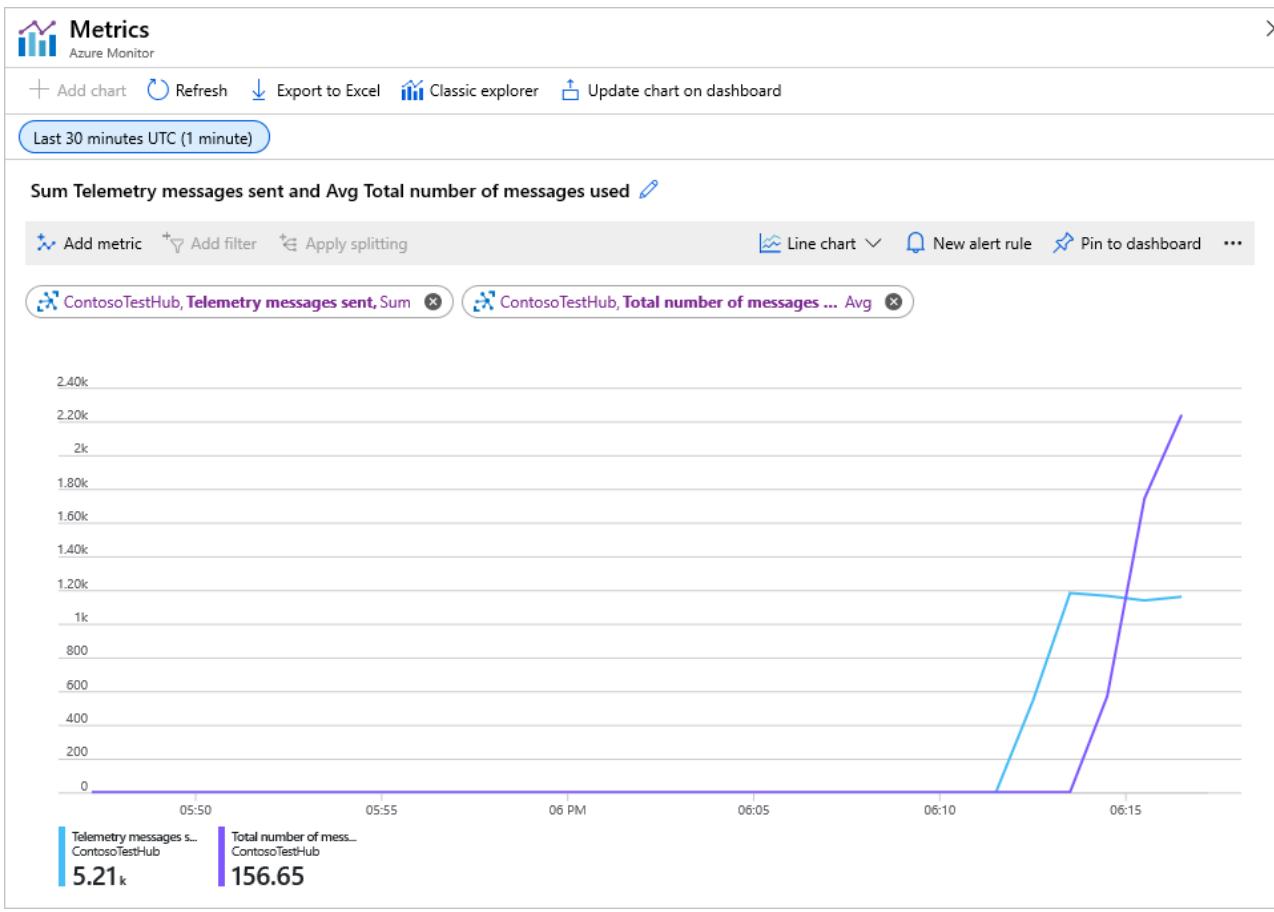
In `Program.cs`, change the `Task.Delay` from 1000 to 10, which reduces the amount of time between sending messages from 1 second to .01 seconds. Shortening this delay increases the number of messages sent.

```
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 with the following details:

- Alerts:**
 - number-messages-used-alert**: Warning, Total number of messages use..., ContosoResources, ContosoTestHub, 7 min ago
 - telemetry-messages**: Warning, Telemetry messages sent > 10..., ContosoResources, ContosoTestHub, 7 min ago
- Filtering:** Subscription: Microsoft Azure Internal Consumption, Source: All sources, Resource group: ContosoResources, Resource type: IoT Hub.
- Search:** Search within displayed alerts...

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 | Save | Discard | Refresh | Download | Acquire lease | ... 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 | | | | | | | | |
| <input type="text" value="Search blobs by prefix (case-sensitive)"/> <input type="checkbox"/> Show deleted blobs | | | | | | Overview | Snapshots | Edit blob |
| | | | | | | Properties | | |
| | | | | | | URL | https://contosostoragemon.blob.core.windows.net/insights-logs-connections/2018/12/17/18/00/CONTOSOTESTHUB/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 | - | |
| | | | | | | Undelete all snapshots | | |

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

7/29/2020 • 5 minutes to read • [Edit Online](#)

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 for IoT hubs created after May 18, 2017.

In this tutorial, you perform the following tasks:

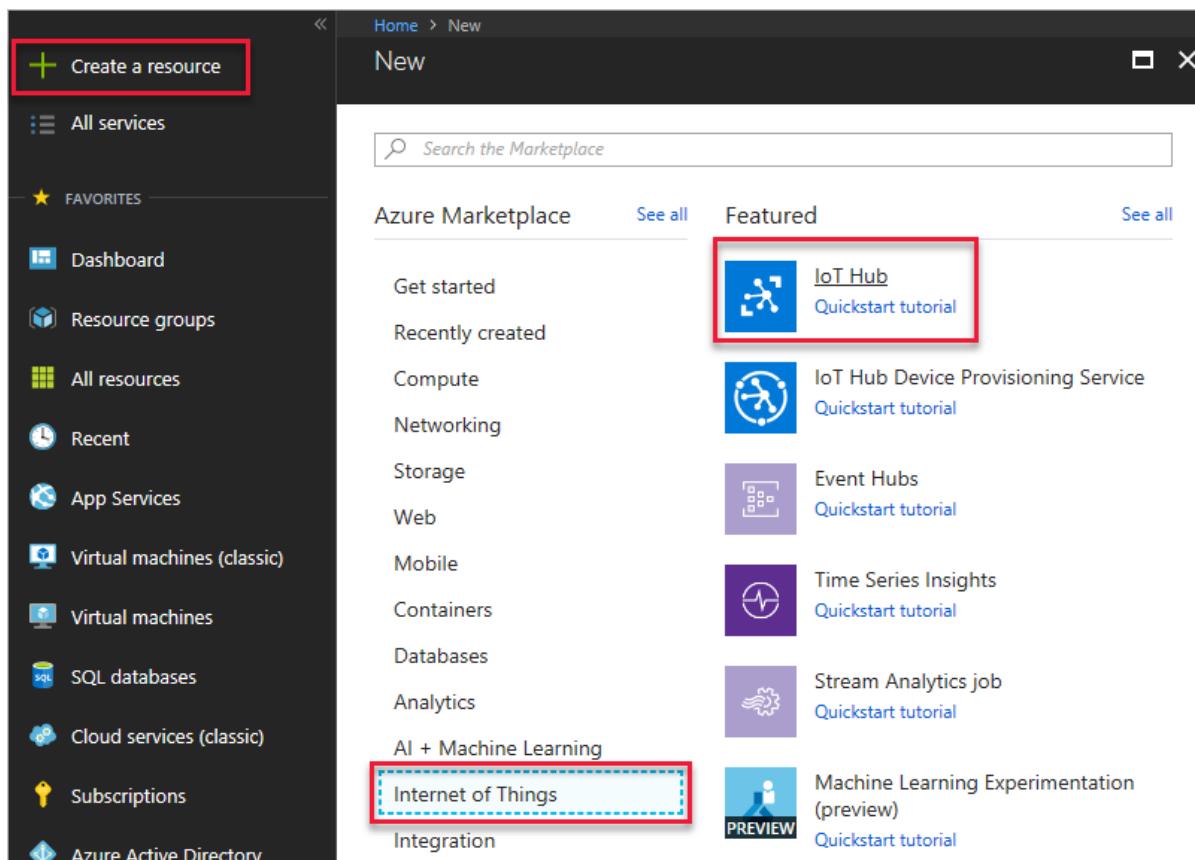
- Using the Azure portal, create an IoT hub.
- Perform a failover.
- See the hub running in the secondary location.
- Perform a fallback to return the IoT hub's operations to the primary location.
- Confirm the hub is running correctly in the right location.

Prerequisites

- An Azure subscription. If you don't have an Azure subscription, create a [free account](#) before you begin.
- Make sure that port 8883 is open in your firewall. The device sample in this tutorial uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

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. This tutorial uses **West US 2**. A failover can only be performed between Azure geo-paired regions. The region geo-paired with West US 2 is WestCentralUS.

IoT Hub Name: specify a name for your IoT hub. The hub name must be globally unique.

The screenshot shows the 'Create IoT hub' wizard on the 'Basics' step. The title bar says 'IoT hub' and 'Microsoft'. Below it, there are tabs for 'Basics', 'Size and scale', and 'Review + create'. The 'Basics' tab is selected. The form fields are as follows:

- * Subscription:** dropdown set to 'Azure Free Trial'.
- * Resource Group:** radio button selected for 'Create new', dropdown set to 'ManFailRG'.
- * Region:** dropdown set to 'West US 2'.
- * IoT Hub Name:** input field set to 'ContosoHubTestFailover2'.

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

Click **Review + create**. (It uses the defaults for size and scale.)

4. Review the information, then click **Create** to create the IoT hub.

The screenshot shows the 'Review + create' step of creating an IoT hub. It's divided into two main sections: 'BASICS' and 'SIZE AND SCALE'. In the 'BASICS' section, the subscription is DEVICEHUB_DEV1, the resource group is ManlFailRG, the region is West US 2, and the IoT Hub Name is ContosoHubTestFailover. In the 'SIZE AND SCALE' section, the pricing tier is S1, there is 1 S1 IoT Hub unit, 400,000 messages per day, and a monthly cost of 25.00 USD. At the bottom, there are three buttons: 'Create' (highlighted in blue), '< Previous: Size and scale', and 'Automation options'.

| BASICS | |
|----------------|------------------------|
| Subscription | DEVICEHUB_DEV1 |
| Resource Group | ManlFailRG |
| 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 |

Perform a manual failover

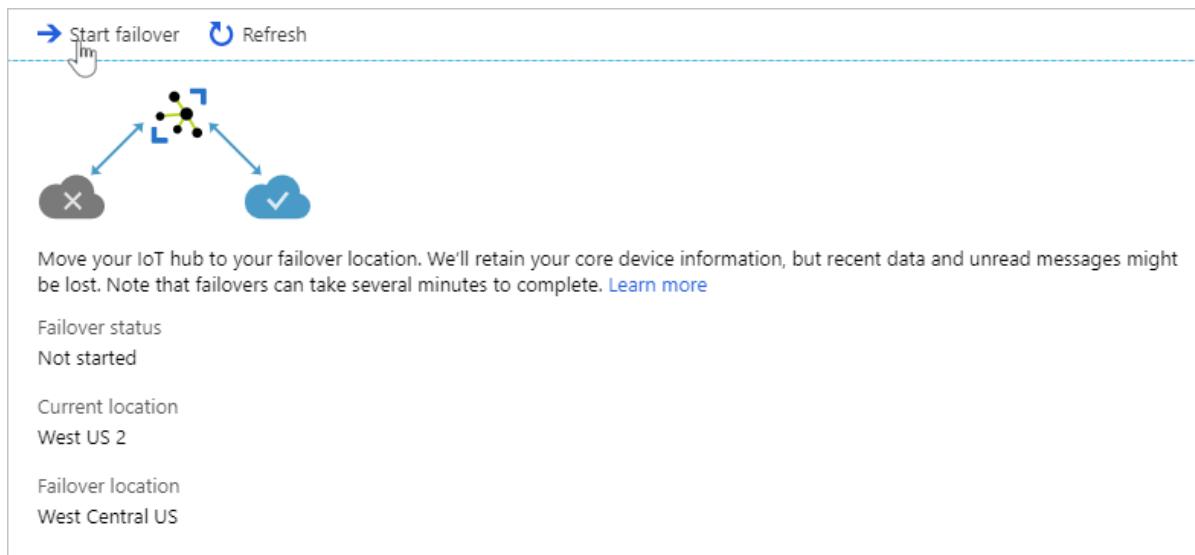
Note that there is a limit of two failovers and two failbacks per day for an IoT hub.

1. Click **Resource groups** and then select the resource group **ManlFailRG**. Click on your hub in the list of resources.
2. Under **Settings** on the IoT Hub pane, click **Failover**.

The screenshot shows the 'ContoHubTestFailover' IoT Hub settings pane. On the left, there's a sidebar with icons for Tags, Events, Settings (Shared access policies, Pricing and scale, IP Filter, Certificates, Built-in endpoints), Failover (which is highlighted with a mouse cursor), Properties, Locks, and Export template. The main area shows the hub's name and some basic configuration details.

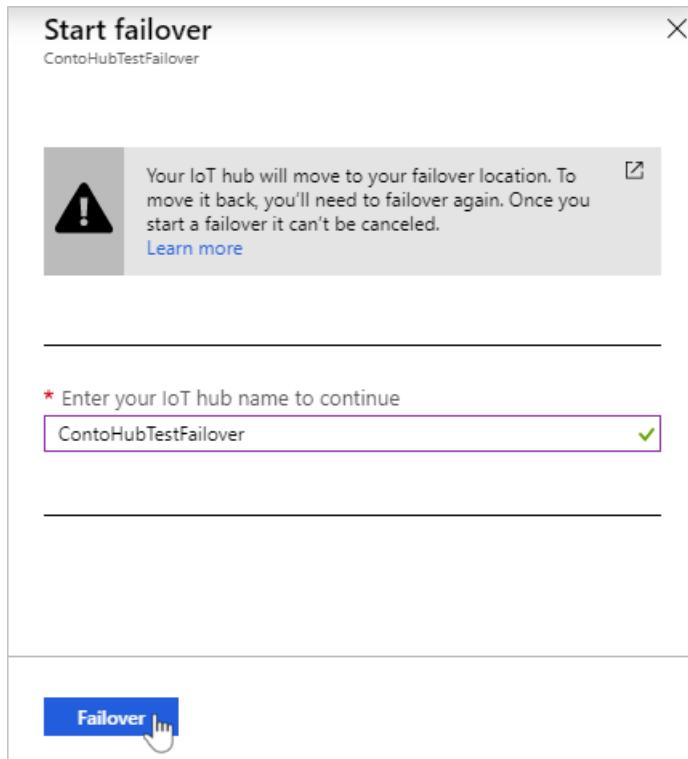
3. On the Manual failover pane, you see the **Current location** and the **Failover location**. The current location always indicates the location in which the hub is currently active. The failover location is the

standard Azure geo-paired region that is paired to the current location. You cannot change the location values. For this tutorial, the current location is **West US 2** and the failover location is **West Central US**.



4. At the top of the Manual failover pane, click **Start failover**.
5. In the confirmation 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 **Failover**.

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.



While the manual failover process is running, a banner appears to tell you a manual failover is in progress.

→ Start failover Refresh

Failover of ContoHubTestFailover 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 in the middle of a manual failover.

→ Move Delete Refresh

Manual failover from West US 2 to West Central US is in progress.

| | |
|---|--|
| Resource group (change) ManualFailRG | Hostname ContoHubTestFailover.azure-devices.net |
| Status Failover is in progress | Pricing and scale tier S1 - Standard |
| Current location West US 2 | Number of IoT Hub units 1 |

After it's finished, the current and failover regions on the Manual Failover page are flipped and the hub is active again. In this example, the current location is now [WestCentralUS](#) and the failover location is now [West US 2](#).

→ Start failover Refresh

Your Failover was successful. Your IoT Hub is running in West Central US.

Your failover was successful and your IoT hub is running in West Central US. To move it back, you'll need to failover again. [Learn more](#)

Failover status
Succeeded

Current location
West Central US

Failover location
West US 2

The overview page also shows a banner indicating that the failover complete and the IoT Hub is running in [West Central US](#).

Failover was successful. Your IoT Hub is running in West Central US.

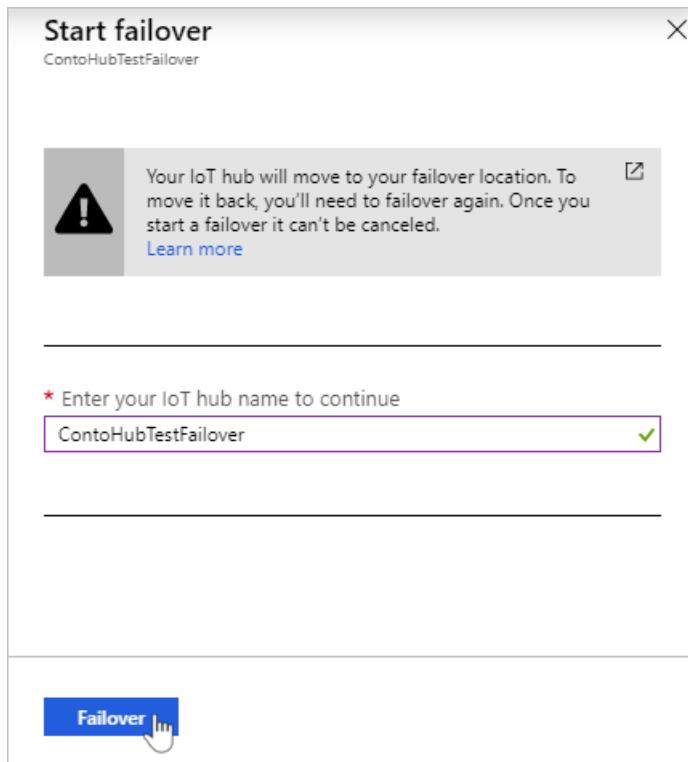
| | |
|--|--|
| Resource group (change) ManualFailRG | Hostname ContoHubTestFailover.azure-devices.net |
| Status Active | Pricing and scale tier S1 - Standard |
| Current location West Central US | Number of IoT Hub units 1 |
| Subscription (change) IoT_SubscriptionContainer_2 | |

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 **Settings** on the IoT Hub pane, click **Failover**.
3. At the top of the Manual failover pane, click **Start failover**.
4. In the confirmation 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 failback is complete, it again shows **West US 2** as the current location and **West Central US** as the failover 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 **ManIFailRG**. 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 failback 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.

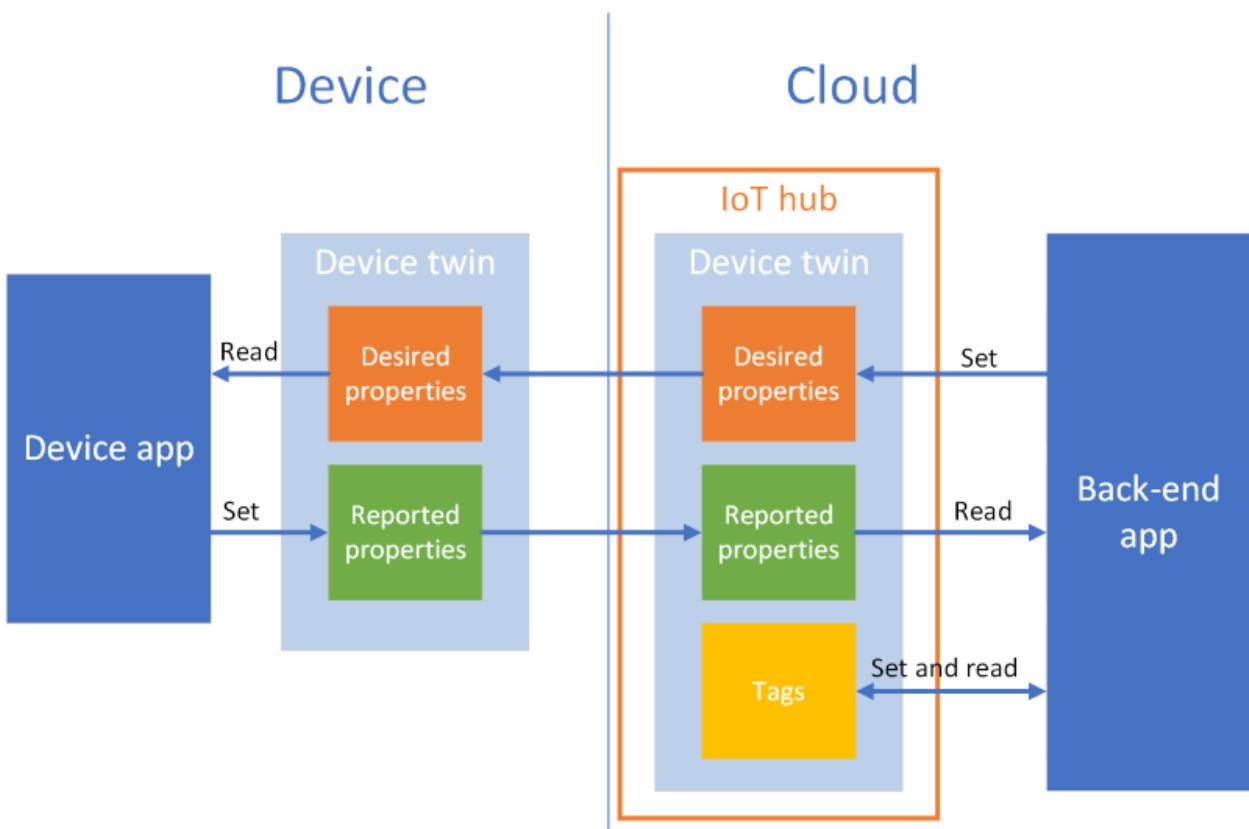
[Manage the state of an IoT device](#)

Tutorial: Configure your devices from a back-end service

7/29/2020 • 13 minutes to read • [Edit Online](#)

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.

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|--|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal . |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

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 v10.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.

Make sure that port 8883 is open in your firewall. The device sample in this tutorial uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

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-iot

# 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 --name $hubname --policy-name service -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:

```
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:

```
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 box.

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 box.

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.

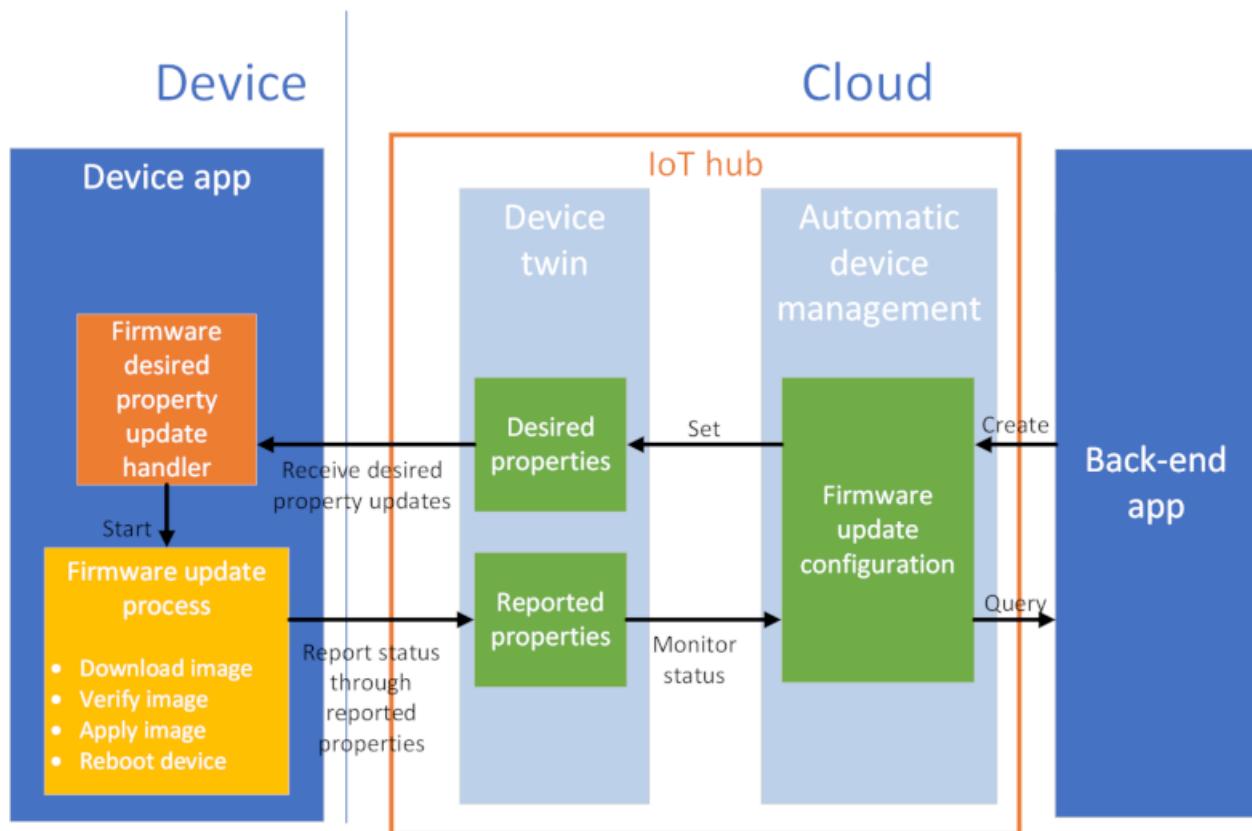
[Implement a device firmware update process](#)

Tutorial: Implement a device firmware update process

7/29/2020 • 12 minutes to read • [Edit Online](#)

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:

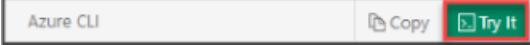
- Create an IoT hub and add a test device to the device identity registry.
- Create a configuration to define the firmware update.
- Simulate the firmware update process on a device.
- Receive status updates from the device as the firmware update progresses.

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can

use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|---|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal . |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

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 v10.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.

Make sure that port 8883 is open in your firewall. The device sample in this tutorial uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

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-iot

# 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 --name $hubname --policy-name service -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:

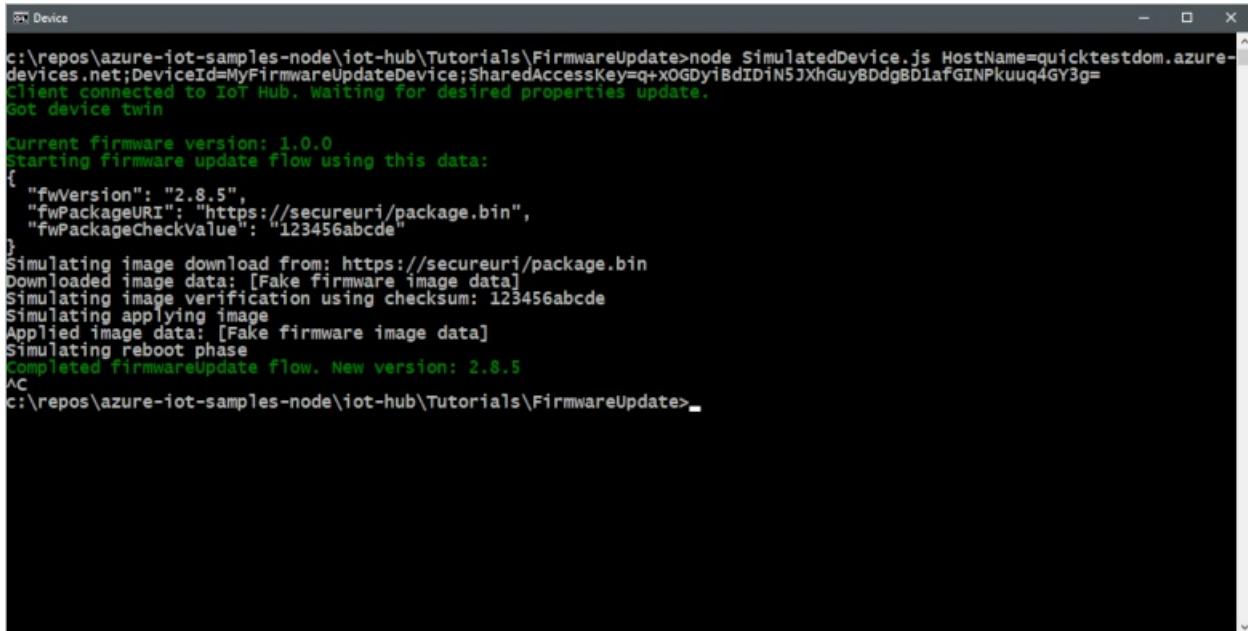
```
npm install
node SimulatedDevice.js "{your device connection string}"
```

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

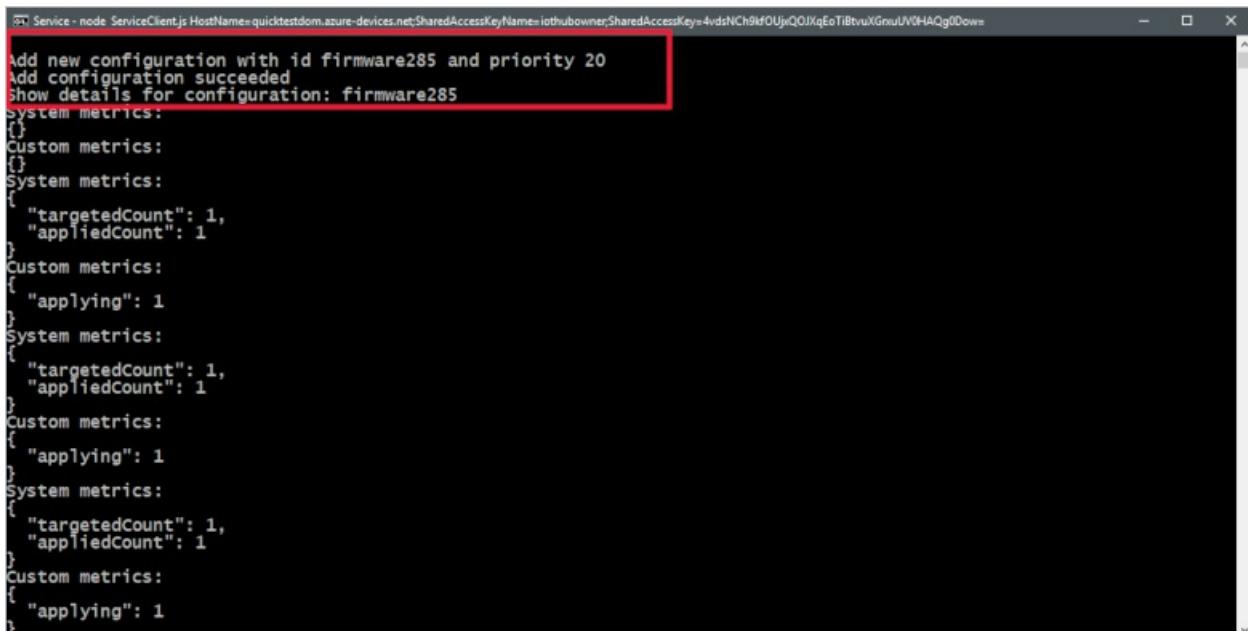
the firmware desired properties update from the back-end application:



```
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=4vdshCh9kfOUjrQOJxqEoTibvuXGnxUV3HAQg0Down
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=4vdshCh9kfOUjxQOJxqEoTbtvuXGnxUVlHAQg0Down
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 automatic device configurations run at creation time and then every five minutes, 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:

| 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 rollback |

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

7/29/2020 • 9 minutes to read • [Edit Online](#)

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

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|--|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal . |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

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-iot
```

NOTE

This article uses the newest version of the Azure IoT extension, called `azure-iot`. The legacy version is called `azure-cli-iot-ext`. You should only have one version installed at a time. You can use the command `az extension list` to validate the currently installed extensions.

Use `az extension remove --name azure-cli-iot-ext` to remove the legacy version of the extension.

Use `az extension add --name azure-iot` to add the new version of the extension.

To see what extensions you have installed, use `az extension list`.

The device simulator application you run in this tutorial is written using Node.js. You need Node.js v10.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.

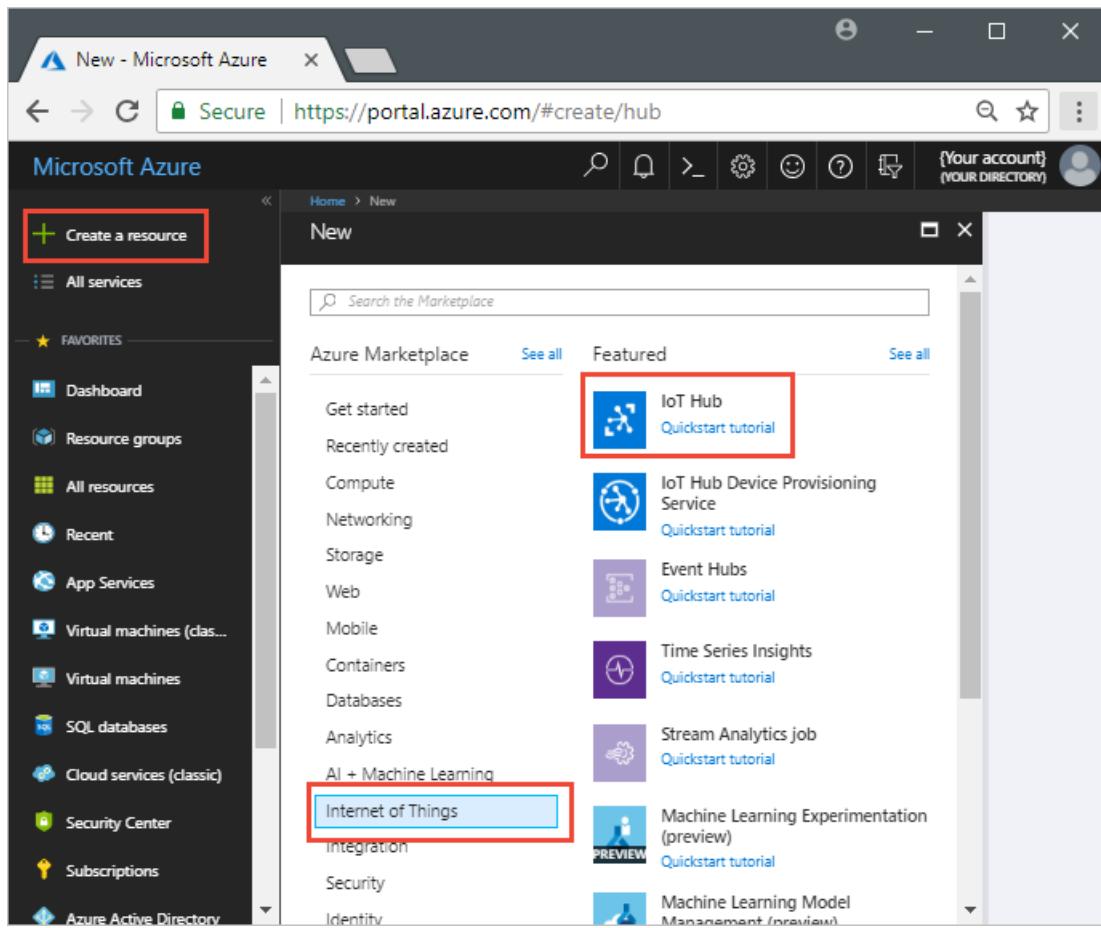
Make sure that port 8883 is open in your firewall. The device sample in this tutorial uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

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**.



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 Azure portal's IoT hub creation wizard. It displays two sections: 'BASICS' and 'SIZE AND SCALE'. In the 'BASICS' section, the IoT Hub Name is set to 'tutorials-iot-hub'. In the 'SIZE AND SCALE' section, the Pricing and scale tier is F1, with 1 unit and 8,000 messages per day. The cost per month is listed as 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.

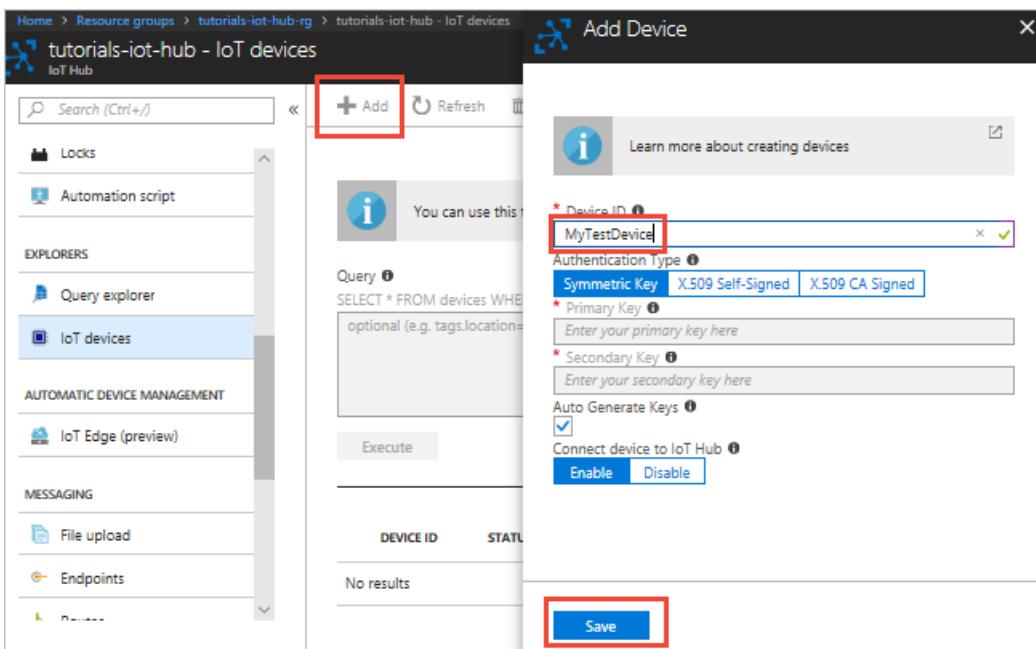
Check device authentication

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 devices' blade in the Azure portal. The left sidebar lists 'Locks', 'Automation script', 'Query explorer', and 'IoT devices' (which is highlighted with a red box). The main area displays the IoT hub configuration: Resource group (tutorials-iot-hub-rg), Status (Activating), Location (West US), Subscription (Pay-As-You-Go), and Subscription ID (1f09d235-d756-45bc-97b3-ffdca8ed7e5a). A callout box in the bottom right corner provides information about the IoT Hub Device Provisioning Service.

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.

| | |
|---------------------------------|---|
| Device Id: | MyTestDevice |
| Primary key | KuLjapcJZ8dQno3wCAkrUvsbBCM/exSATb5sRzBbKp8= |
| Secondary key | Kw8a3AcnQla9o4108cR0Lk+ynqHKwS7RGFurvFlyGIM= |
| Connection string—primary key | HostName=tutorials-iot-hub.azure-devices.net;DeviceId=MyTestDevice;SharedAccessKey=KuLjapcJZ8dQno3wCAkrUvsbBCM/exSATb5sRzBbKp8= |
| Connection string—secondary key | HostName=tutorials-iot-hub.azure-devices.net;DeviceId=MyTestDevice;SharedAccessKey=Kw8a3AcnQla9o4108cR0Lk+ynqHKwS7RGFurvFlyGIM= |

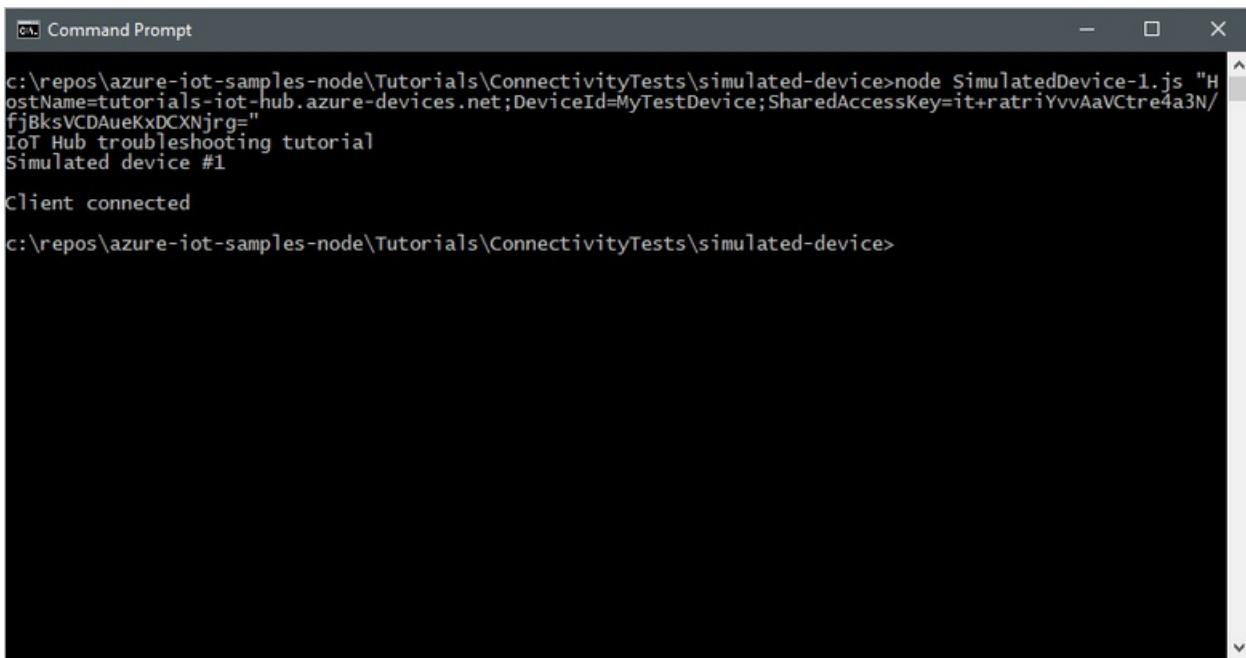
To simulate **MyTestDevice** sending telemetry to your IoT hub, run the Node.js simulated device application you downloaded previously.

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)

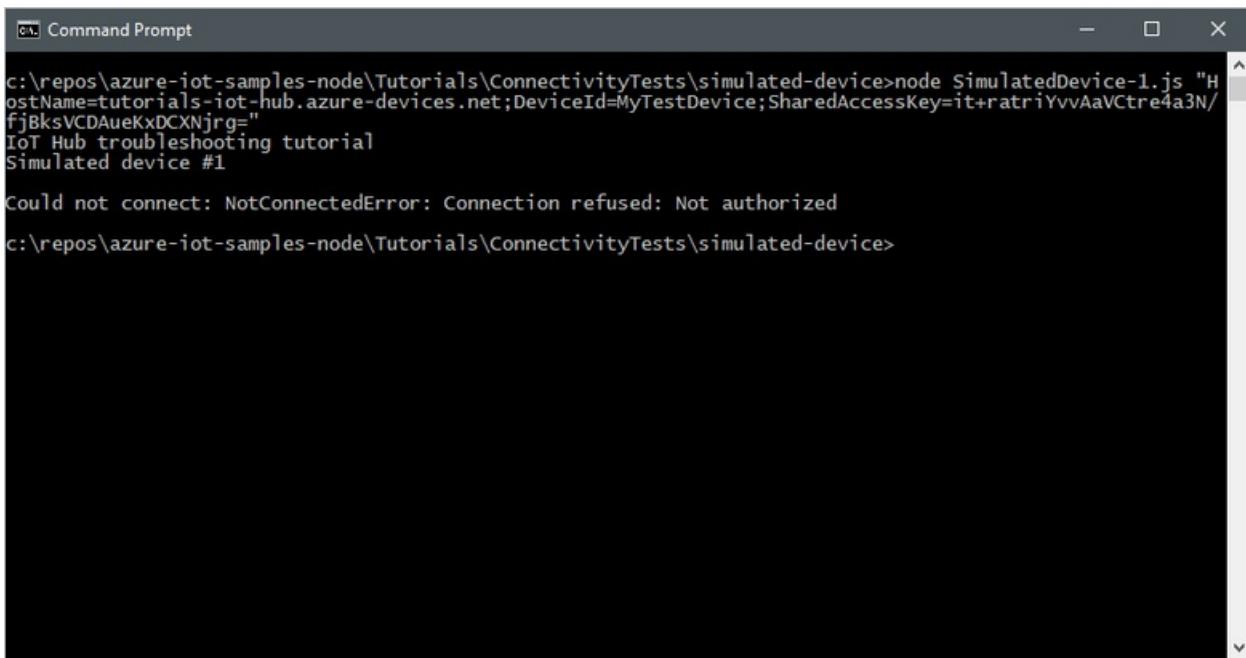
# Requires the IoT Extension for Azure CLI
# az extension add --name azure-iot

# 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 "H  
ostName=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:

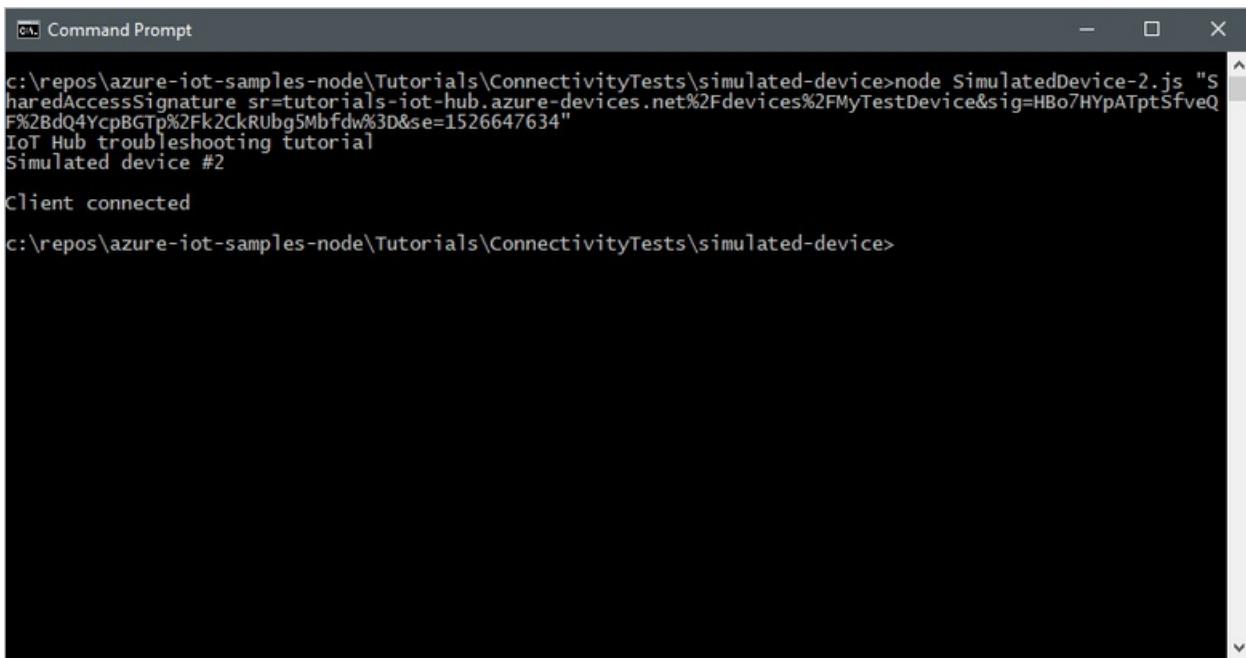
```
SharedAccessSignature sr=tutorials-iot-hub.azure-devices.net%2Fdevices%2FMyTestDevice&sig=....&se=1524155307
```

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** 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%2Fk2CkRUb95Mbfdw%3D&se=1526647634"
IoT Hub troubleshooting tutorial
Simulated device #2

Client connected

c:\repos\azure-iot-samples-node\Tutorials\ConnectivityTests\simulated-device>
```

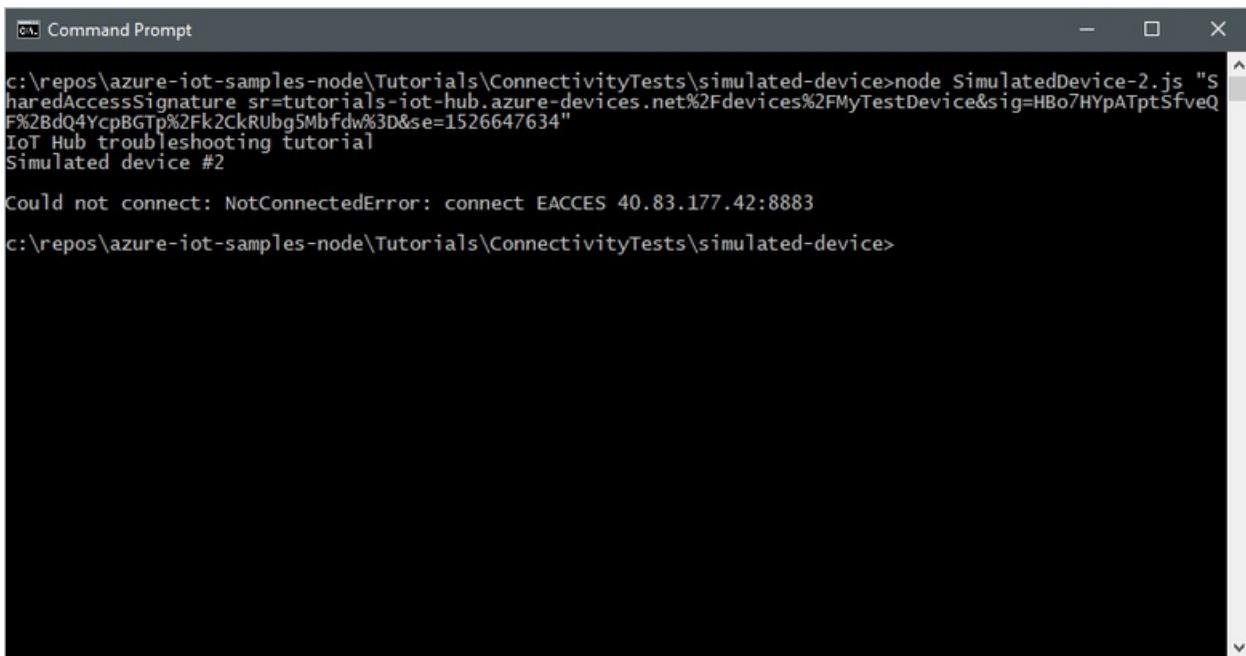
You've now successfully authenticated from a device using a test SAS token generated by a CLI command. The **SimulatedDevice-2.js** file includes sample code that shows you how to generate a SAS token in code.

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` 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:

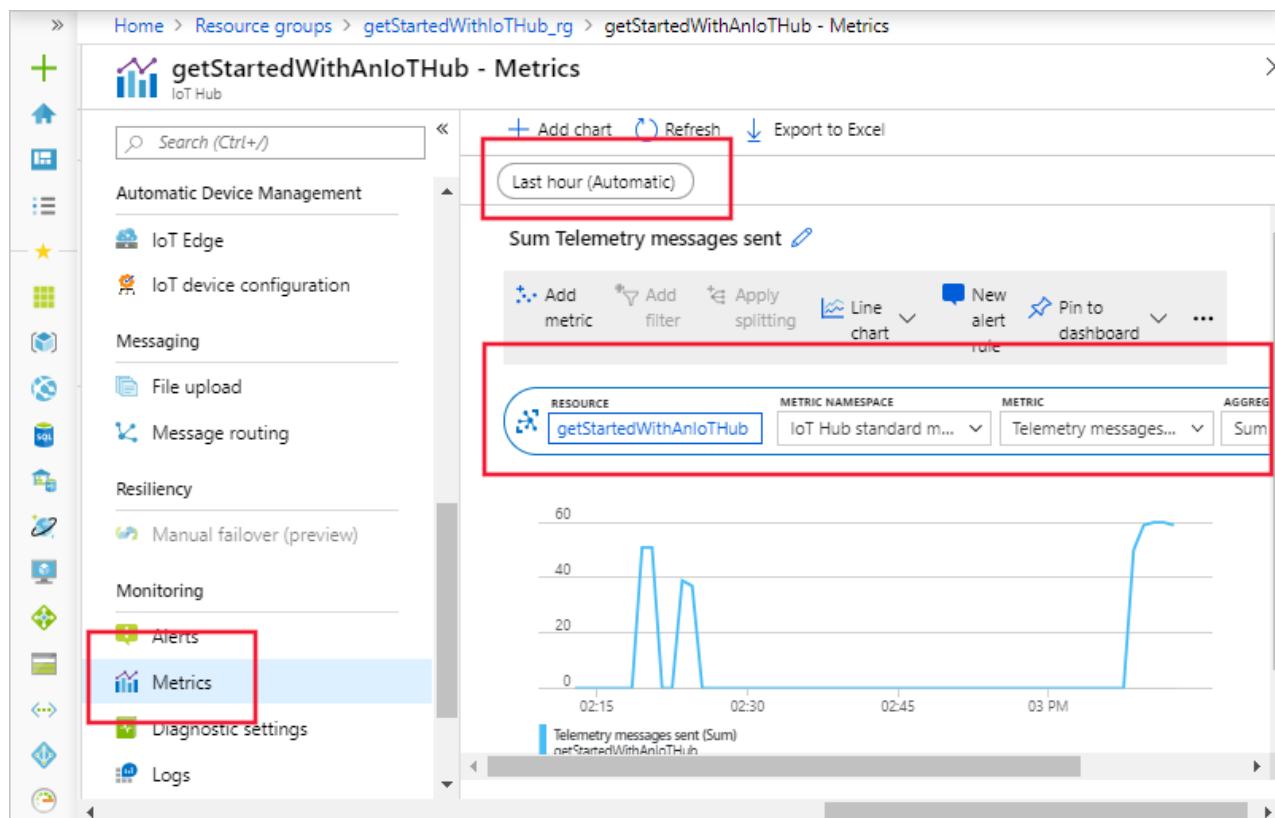
```

Command Prompt - node SimulatedDevice-3.js "HostName=tutorials-iot-hub.azure-devices.net;DeviceId=MyTestDevice;SharedAccessKey=MzNkZjFiOWNiMDZiMTYwMjFjNTAZZmuO"
c:\repos\azure-iot-samples-node\Tutorials\ConnectivityTests\simulated-device>node SimulatedDevice-3.js "HostName=tutorials-iot-hub.azure-devices.net;DeviceId=MyTestDevice;SharedAccessKey=MzNkZjFiOWNiMDZiMTYwMjFjNTAZZmuO"
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.

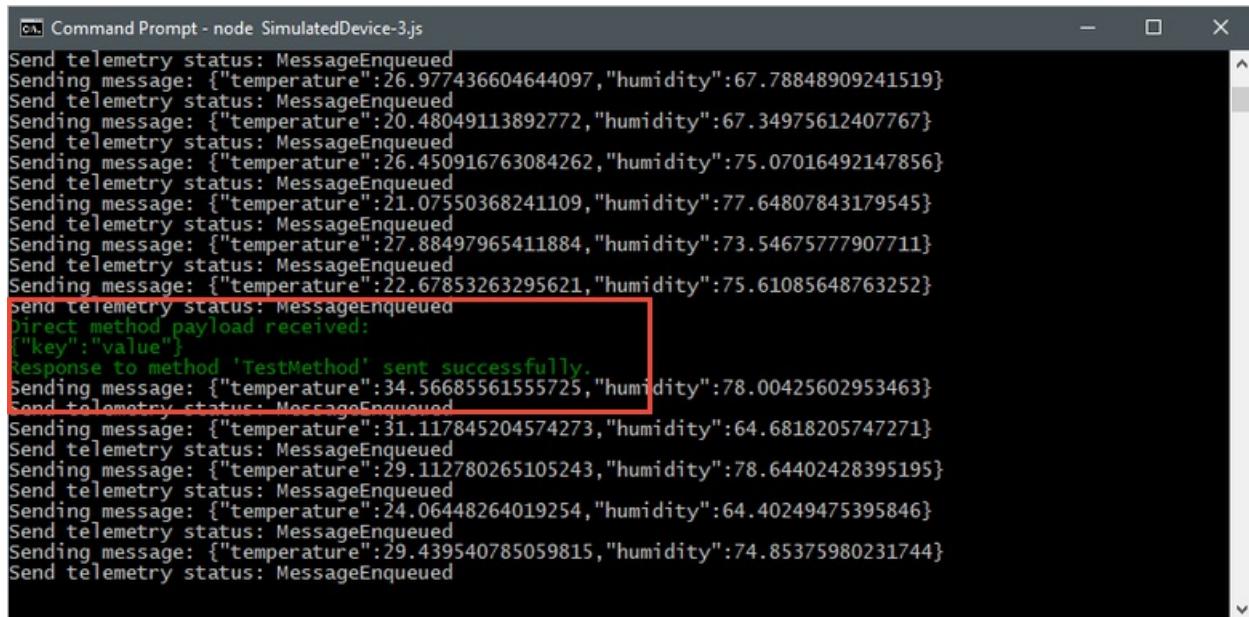
In a terminal window, use the following command to run the simulated device application:

```
node SimulatedDevice-3.js "{your device connection string}"
```

Use a CLI command to call a direct method on the device:

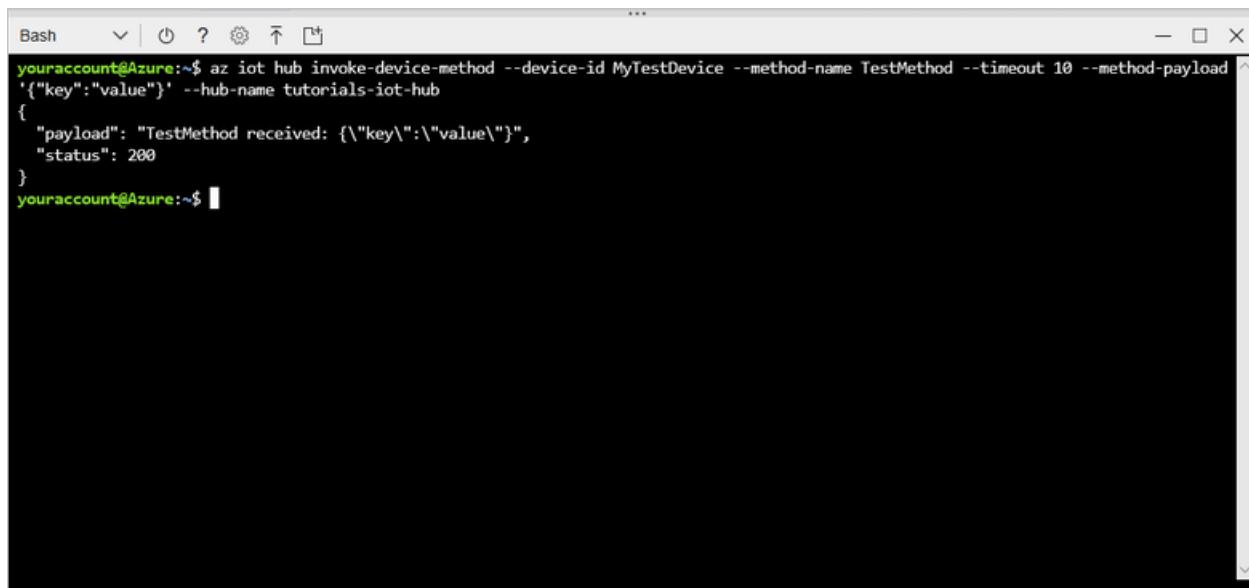
```
az iot hub invoke-device-method --device-id MyTestDevice --method-name TestMethod --timeout 10 --method-payload '{"key":"value"}' --hub-name {YourIoTHubName}
```

The simulated device prints a message to the console when it receives a direct method call:



```
Command Prompt - node SimulatedDevice-3.js
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:



```
Bash
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.

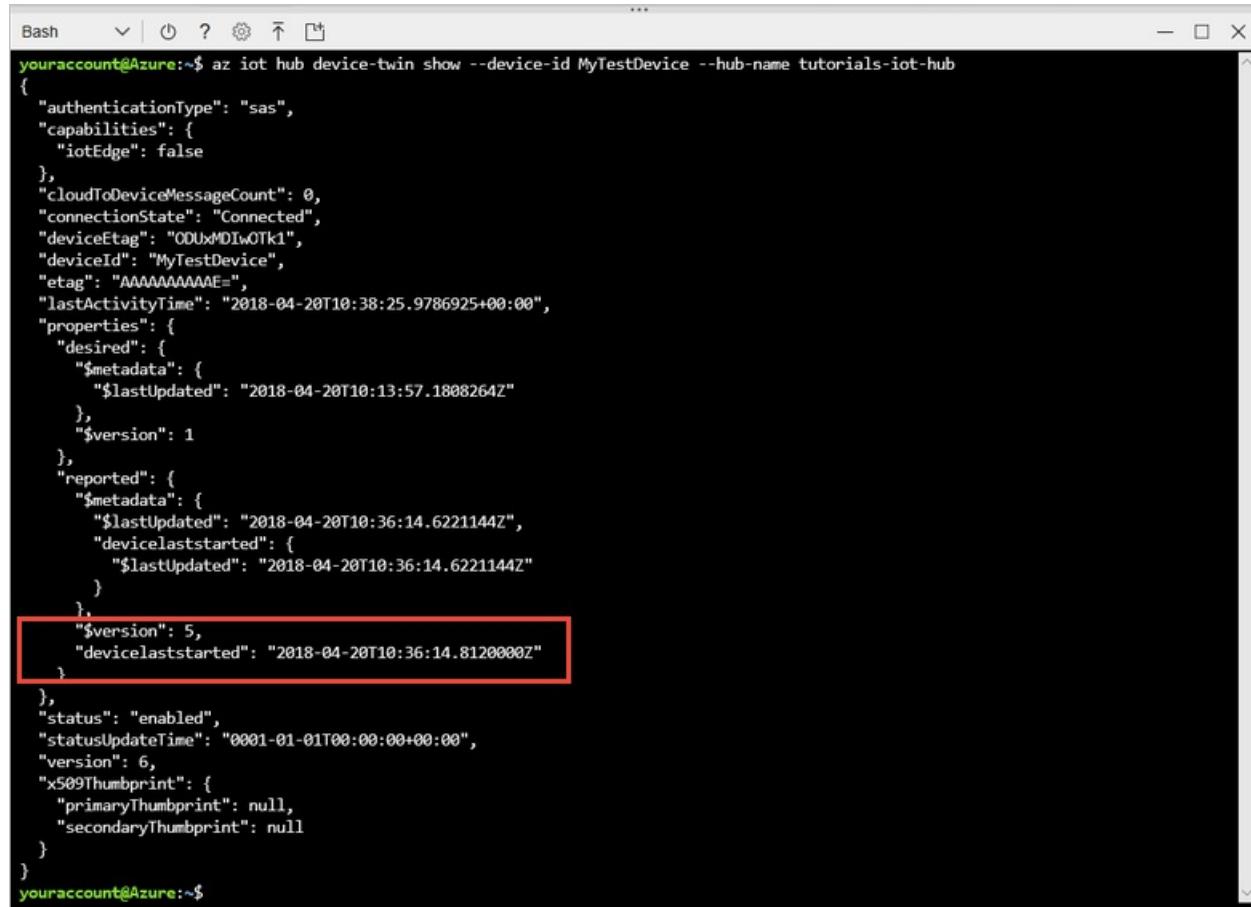
In a terminal window, use the following command to run the simulated device application:

```
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.

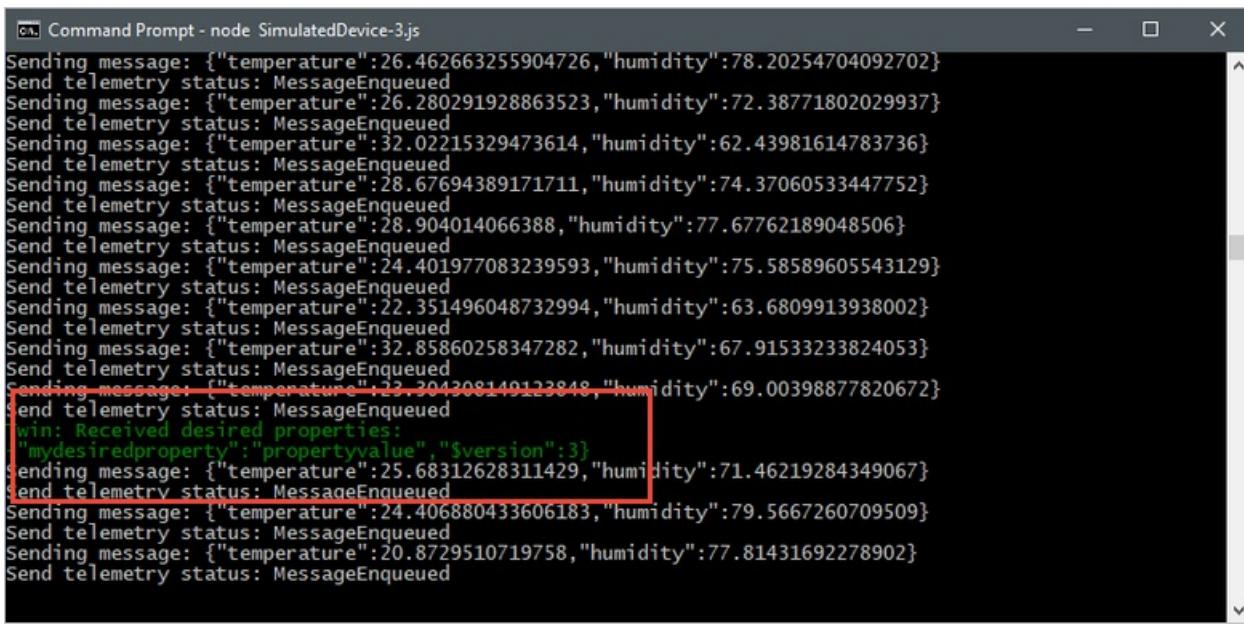


```
youraccount@Azure:~$ az iot hub device-twin show --device-id MyTestDevice --hub-name tutorials-iot-hub
{
  "authenticationType": "sas",
  "capabilities": {
    "iotEdge": false
  },
  "cloudToDeviceMessageCount": 0,
  "connectionState": "Connected",
  "deviceEtag": "ODUxMDIwOTk1",
  "deviceId": "MyTestDevice",
  "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"
        }
      },
      "$version": 5,
      "devicelaststarted": "2018-04-20T10:36:14.8120000Z"
    }
  },
  "status": "enabled",
  "statusUpdateTime": "2001-01-01T00:00:00+00:00",
  "version": 6,
  "x509Thumbprint": {
    "primaryThumbprint": null,
    "secondaryThumbprint": null
  }
}
youraccount@Azure:~$
```

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](#)

Azure Policy built-in definitions for Azure IoT Hub

7/29/2020 • 2 minutes to read • [Edit Online](#)

For IoT Hub sample code that shows how to implement common IoT scenarios, see the [IoT Hub quickstarts](#). There are quickstarts for multiple programming languages including C, Node.js, and Python.

This page is an index of [Azure Policy](#) built-in policy definitions for Azure IoT Hub. For additional Azure Policy built-ins for other services, see [Azure Policy built-in definitions](#).

The name of each built-in policy definition links to the policy definition in the Azure portal. Use the link in the **Version** column to view the source on the [Azure Policy GitHub repo](#).

Azure IoT Hub

| NAME (AZURE PORTAL) | DESCRIPTION | EFFECT(S) | VERSION (GITHUB) |
|--|--|----------------------------|-----------------------|
| Diagnostic logs in IoT Hub should be enabled | Audit enabling of diagnostic logs. This enables you to recreate activity trails to use for investigation purposes; when a security incident occurs or when your network is compromised | AuditIfNotExists, Disabled | 2.0.0 |

Next steps

- See the built-ins on the [Azure Policy GitHub repo](#).
- Review the [Azure Policy definition structure](#).
- Review [Understanding policy effects](#).

TLS support in IoT Hub

7/29/2020 • 2 minutes to read • [Edit Online](#)

IoT Hub uses Transport Layer Security (TLS) to secure connections from IoT devices and services. Three versions of the TLS protocol are currently supported, namely versions 1.0, 1.1, and 1.2.

TLS 1.0 and 1.1 are considered legacy and are planned for deprecation. For more information, see [Deprecating TLS 1.0 and 1.1 for IoT Hub](#). It is strongly recommended that you use TLS 1.2 as the preferred TLS version when connecting to IoT Hub.

TLS 1.2 enforcement available in select regions

For added security, configure your IoT Hubs to *only* allow client connections that use TLS version 1.2 and to enforce the use of [recommended ciphers](#). This feature is only supported in these regions:

- East US
- South Central US
- West US 2
- US Gov Arizona
- US Gov Virginia

For this purpose, provision a new IoT Hub in any of the supported regions and set the `minTlsVersion` property to `1.2` in your Azure Resource Manager template's IoT hub resource specification:

```
{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "resources": [
        {
            "type": "Microsoft.Devices/IotHubs",
            "apiVersion": "2020-01-01",
            "name": "<provide-a-valid-resource-name>",
            "location": "<any-of-supported-regions-below>",
            "properties": {
                "minTlsVersion": "1.2"
            },
            "sku": {
                "name": "<your-hubs-SKU-name>",
                "tier": "<your-hubs-SKU-tier>",
                "capacity": 1
            }
        }
    ]
}
```

The created IoT Hub resource using this configuration will refuse device and service clients that attempt to connect using TLS versions 1.0 and 1.1. Similarly, the TLS handshake will be refused if the client HELLO message does not list any of the [recommended ciphers](#).

NOTE

The `minTlsVersion` property is read-only and cannot be changed once your IoT Hub resource is created. It is therefore essential that you properly test and validate that *all* your IoT devices and services are compatible with TLS 1.2 and the [recommended ciphers](#) in advance.

Upon failovers, the `minTlsVersion` property of your IoT Hub will remain effective in the geo-paired region post-failover.

Recommended ciphers

IoT Hubs that are configured to accept only TLS 1.2 will also enforce the use of the following recommended ciphers:

- `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256`
- `TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384`
- `TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256`
- `TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384`

For IoT Hubs not configured for TLS 1.2 enforcement, TLS 1.2 still works with the following ciphers:

- `TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256`
- `TLS_DHE_RSA_WITH_AES_256_GCM_SHA384`
- `TLS_DHE_RSA_WITH_AES_128_GCM_SHA256`
- `TLS_RSA_WITH_AES_256_GCM_SHA384`
- `TLS_RSA_WITH_AES_128_GCM_SHA256`
- `TLS_RSA_WITH_AES_256_CBC_SHA256`
- `TLS_RSA_WITH_AES_128_CBC_SHA256`
- `TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA`
- `TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA`
- `TLS_RSA_WITH_AES_256_CBC_SHA`
- `TLS_RSA_WITH_AES_128_CBC_SHA`
- `TLS_RSA_WITH_3DES_EDE_CBC_SHA`

Use TLS 1.2 in your IoT Hub SDKs

Use the links below to configure TLS 1.2 and allowed ciphers in IoT Hub client SDKs.

| LANGUAGE | VERSIONS SUPPORTING TLS 1.2 | DOCUMENTATION |
|----------|-----------------------------|----------------------|
| C | Tag 2019-12-11 or newer | Link |
| Python | Version 2.0.0 or newer | Link |
| C# | Version 1.21.4 or newer | Link |
| Java | Version 1.19.0 or newer | Link |
| NodeJS | Version 1.12.2 or newer | Link |

Use TLS 1.2 in your IoT Edge setup

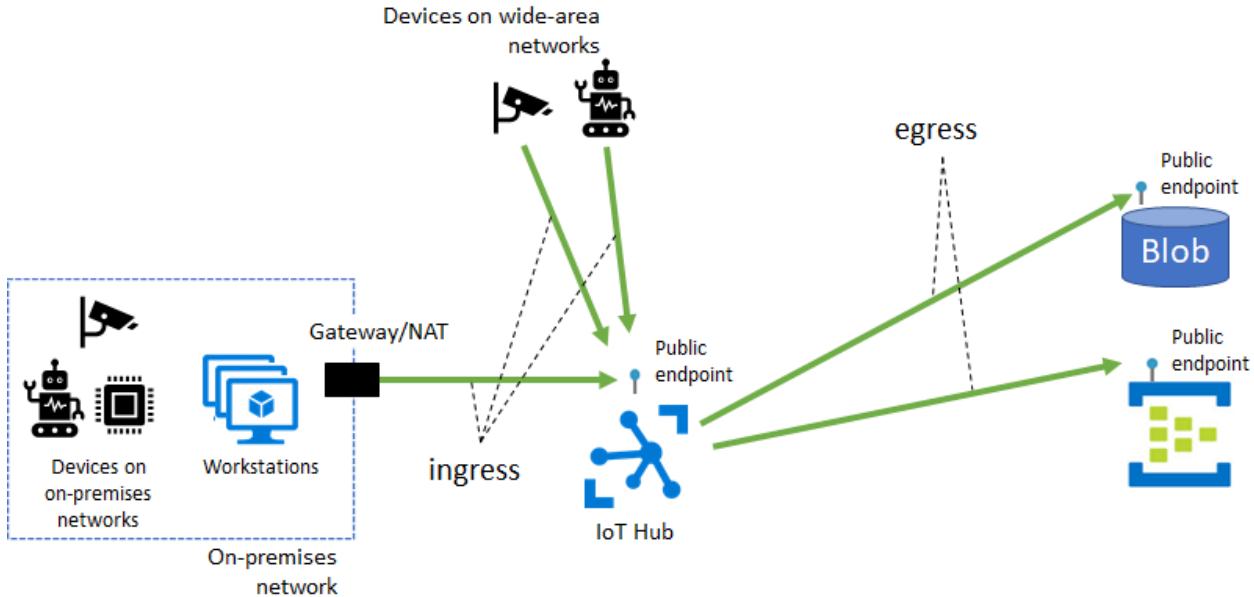
IoT Edge devices can be configured to use TLS 1.2 when communicating with IoT Hub. For this purpose, use the [IoT](#)

[Edge documentation page.](#)

IoT Hub support for virtual networks with Private Link and Managed Identity

7/29/2020 • 13 minutes to read • [Edit Online](#)

By default, IoT Hub's hostnames map to a public endpoint with a publicly routable IP address over the internet. Different customers share this IoT Hub public endpoint, and IoT devices in over wide-area networks and on-premises networks can all access it.



IoT Hub features including [message routing](#), [file upload](#), and [bulk device import/export](#) also require connectivity from IoT Hub to a customer-owned Azure resource over its public endpoint. These connectivity paths collectively make up the egress traffic from IoT Hub to customer resources.

You might want to restrict connectivity to your Azure resources (including IoT Hub) through a VNet that you own and operate. These reasons include:

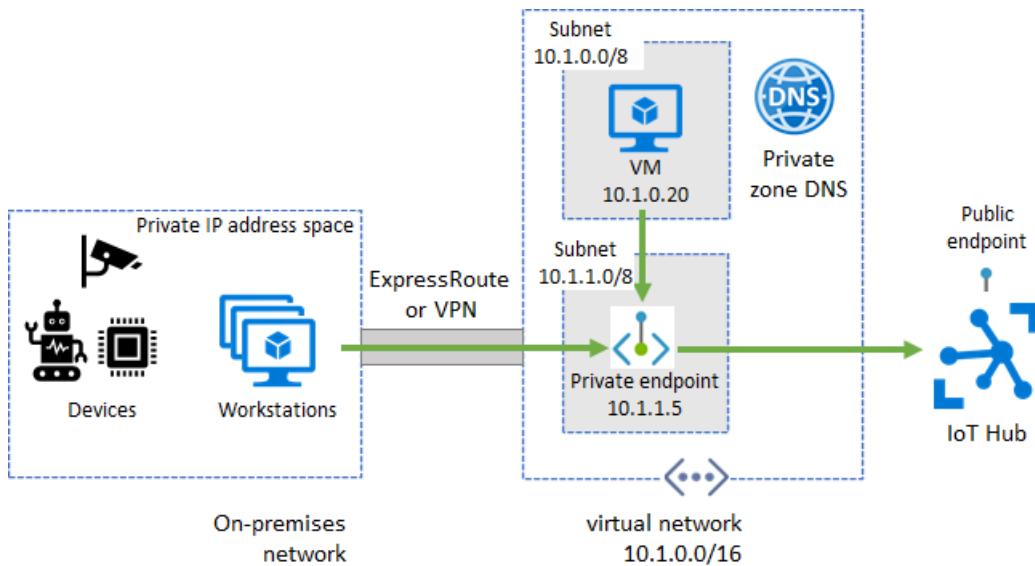
- Introducing network isolation for your IoT hub by preventing connectivity exposure to the public internet.
- Enabling a private connectivity experience from your on-premises network assets ensuring that your data and traffic is transmitted directly to Azure backbone network.
- Preventing exfiltration attacks from sensitive on-premises networks.
- Following established Azure-wide connectivity patterns using [private endpoints](#).

This article describes how to achieve these goals using [Azure Private Link](#) for ingress connectivity to IoT Hub and using trusted Microsoft services exception for egress connectivity from IoT Hub to other Azure resources.

Ingress connectivity to IoT Hub using Azure Private Link

A private endpoint is a private IP address allocated inside a customer-owned VNet via which an Azure resource is reachable. Through Azure Private Link, you can set up a private endpoint for your IoT hub to allow services inside your VNet to reach IoT Hub without requiring traffic to be sent to IoT Hub's public endpoint. Similarly, your on-premises devices can use [Virtual Private Network \(VPN\)](#) or [ExpressRoute](#) peering to gain connectivity to your VNet and your IoT Hub (via its private endpoint). As a result, you can restrict or completely block off connectivity to your

IoT hub's public endpoints by using [IoT Hub IP filter](#) and [configuring routing not to send any data to the built-in endpoint](#). This approach keeps connectivity to your Hub using the private endpoint for devices. The main focus of this setup is for devices inside an on-premises network. This setup isn't advised for devices deployed in a wide-area network.



Before proceeding ensure that the following prerequisites are met:

- You've [created an Azure VNet](#) with a subnet in which the private endpoint will be created.
- For devices that operate in on-premises networks, set up [Virtual Private Network \(VPN\)](#) or [ExpressRoute](#) private peering into your Azure VNet.

Set up a private endpoint for IoT Hub ingress

Private endpoint works for IoT Hub device APIs (like device-to-cloud messages) as well as service APIs (like creating and updating devices).

1. In Azure portal, select **Networking**, **Private endpoint connections**, and click the **+ Private endpoint**.

| CONNECTION NAME | CONNECTION STATE | PRIVATE ENDPOINT | DESCRIPTION |
|-------------------------------|------------------|--------------------|---------------|
| jlian-iothub-msi-again.9ee... | Approved | jlian-private-link | Auto-Approved |
| jlian-iothub-msi-again.c97... | Pending | manual | Manual |

2. Provide the subscription, resource group, name, and region to create the new private endpoint in. Ideally, private endpoint should be created in the same region as your hub.
3. Click **Next: Resource**, and provide the subscription for your IoT Hub resource, and select

"Microsoft.Devices/IotHubs" as resource type, your IoT Hub name as **resource**, and **iotHub** as target subresource.

4. Click **Next: Configuration** and provide your virtual network and subnet to create the private endpoint in. Select the option to integrate with Azure private DNS zone, if desired.
5. Click **Next: Tags**, and optionally provide any tags for your resource.
6. Click **Review + create** to create your private link resource.

Built-in Event Hub compatible endpoint doesn't support access over private endpoint

The [built-in Event Hub compatible endpoint](#) doesn't support access over private endpoint. When configured, an IoT hub's private endpoint is for ingress connectivity only. Consuming data from built-in Event Hub compatible endpoint can only be done over the public internet.

IoT Hub's [IP filter](#) also doesn't control public access to the built-in endpoint. To completely block public network access to your IoT hub, you must:

1. Configure private endpoint access for IoT Hub
2. [Turn off public network access](#) or use IP filter to block all IP
3. Stop using the built-in Event Hub endpoint by [setting up routing to not send data to it](#)
4. Turn off the [fallback route](#)
5. Configure egress to other Azure resources using [trusted Microsoft service](#)

Pricing for Private Link

For pricing details, see [Azure Private Link pricing](#).

Egress connectivity from IoT Hub to other Azure resources

IoT Hub can connect to your Azure blob storage, event hub, service bus resources for [message routing](#), [file upload](#), and [bulk device import/export](#) over the resources' public endpoint. Binding your resource to a VNet blocks connectivity to the resource by default. As a result, this configuration prevents IoT Hub's from working sending data to your resources. To fix this issue, enable connectivity from your IoT Hub resource to your storage account, event hub, or service bus resources via the [trusted Microsoft service](#) option.

Turn on managed identity for IoT Hub

To allow other services to find your IoT hub as a trusted Microsoft service, it must have a system-assigned managed identity.

1. Navigate to **Identity** in your IoT Hub portal
2. Under **Status**, select **On**, then click **Save**.

The screenshot shows the 'myIoTHub | Identity' page in the Azure portal. The left sidebar lists various settings like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Events, Shared access policies, Pricing and scale, IP Filter, Private endpoint connections, Certificates, Built-in endpoints, Failover, Properties, Locks, and Export template. The 'Identity' option is selected. The main content area is titled 'System Assigned' and contains information about system-assigned managed identities, a status switch (set to 'On'), an object ID input field (containing '97a114b7-fa48-4af6-9f94-3acb82e2672d'), and a link to 'Role Assignments'. A note at the top states: 'This resource is registered with Azure Active Directory. You can control its access to services like Azure Resource Manager, Azure Key Vault, etc.'

Assign managed identity to your IoT Hub at creation time using ARM template

To assign managed identity to your IoT hub at resource provisioning time, use the ARM template below:

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "resources": [
    {
      "type": "Microsoft.Devices/IotHubs",
      "apiVersion": "2020-03-01",
      "name": "<provide-a-valid-resource-name>",
      "location": "<any-of-supported-regions>",
      "identity": {
        "type": "SystemAssigned"
      },
      "sku": {
        "name": "<your-hubs-SKU-name>",
        "tier": "<your-hubs-SKU-tier>",
        "capacity": 1
      }
    },
    {
      "type": "Microsoft.Resources/deployments",
      "apiVersion": "2018-02-01",
      "name": "updateIotHubWithKeyEncryptionKey",
      "dependsOn": [
        "<provide-a-valid-resource-name>"
      ],
      "properties": {
        "mode": "Incremental",
        "template": {
          "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
          "contentVersion": "0.9.0.0",
          "resources": [
            {
              "type": "Microsoft.Devices/IotHubs",
              "apiVersion": "2020-03-01",
              "name": "<provide-a-valid-resource-name>",
              "location": "<any-of-supported-regions>",
              "identity": {
                "type": "SystemAssigned"
              },
              "sku": {
                "name": "<your-hubs-SKU-name>",
                "tier": "<your-hubs-SKU-tier>",
                "capacity": 1
              }
            }
          ]
        }
      }
    }
  ]
}
```

After substituting the values for your resource `name`, `location`, `SKU.name` and `SKU.tier`, you can use Azure CLI to deploy the resource in an existing resource group using:

```
az deployment group create --name <deployment-name> --resource-group <resource-group-name> --template-file <template-file.json>
```

After the resource is created, you can retrieve the managed service identity assigned to your hub using Azure CLI:

```
az resource show --resource-type Microsoft.Devices/IotHubs --name <iot-hub-resource-name> --resource-group <resource-group-name>
```

Pricing for managed identity

Trusted Microsoft first party services exception feature is free of charge. Charges for the provisioned storage accounts, event hubs, or service bus resources apply separately.

Egress connectivity to storage account endpoints for routing

IoT Hub can route messages to a customer-owned storage account. To allow the routing functionality to access a storage account while firewall restrictions are in place, your IoT Hub needs to have a [managed identity](#). Once a managed identity is provisioned, follow the steps below to give RBAC permission to your hub's resource identity to access your storage account.

1. In the Azure portal, navigate to your storage account's **Access control (IAM)** tab and click **Add** under the **Add a role assignment** section.
2. Select **Storage Blob Data Contributor** (*not Contributor or Storage Account Contributor*) as **role**, **Azure AD user, group, or service principal** as **Assigning access to** and select your IoT Hub's resource name in the drop-down list. Click the **Save** button.
3. Navigate to the **Firewalls and virtual networks** tab in your storage account and enable **Allow access from selected networks** option. Under the **Exceptions** list, check the box for **Allow trusted Microsoft services to access this storage account**. Click the **Save** button.
4. On your IoT Hub's resource page, navigate to **Message routing** tab.
5. Navigate to **Custom endpoints** section and click **Add**. Select **Storage** as the endpoint type.
6. On the page that shows up, provide a name for your endpoint, select the container that you intend to use in your blob storage, provide encoding, and file name format. Select **System Assigned** as the **Authentication type** to your storage endpoint. Click the **Create** button.

Now your custom storage endpoint is set up to use your hub's system assigned identity, and it has permission to access your storage resource despite its firewall restrictions. You can now use this endpoint to set up a routing rule.

Egress connectivity to event hubs endpoints for routing

IoT Hub can be configured to route messages to a customer-owned event hubs namespace. To allow the routing functionality to access an event hubs resource while firewall restrictions are in place, your IoT Hub needs to have a managed identity. Once a managed identity is created, follow the steps below to give RBAC permission to your hub's resource identity to access your event hubs.

1. In the Azure portal, navigate to your event hubs **Access control (IAM)** tab and click **Add** under the **Add a role assignment** section.
2. Select **Event Hubs Data Sender** as **role**, **Azure AD user, group, or service principal** as **Assigning access to** and select your IoT Hub's resource name in the drop-down list. Click the **Save** button.
3. Navigate to the **Firewalls and virtual networks** tab in your event hubs and enable **Allow access from selected networks** option. Under the **Exceptions** list, check the box for **Allow trusted Microsoft services to access event hubs**. Click the **Save** button.
4. On your IoT Hub's resource page, navigate to **Message routing** tab.
5. Navigate to **Custom endpoints** section and click **Add**. Select **Event hubs** as the endpoint type.
6. On the page that shows up, provide a name for your endpoint, select your event hubs namespace and instance and click the **Create** button.

Now your custom event hubs endpoint is set up to use your hub's system assigned identity, and it has permission to access your event hubs resource despite its firewall restrictions. You can now use this endpoint to set up a routing rule.

Egress connectivity to service bus endpoints for routing

IoT Hub can be configured to route messages to a customer-owned service bus namespace. To allow the routing functionality to access a service bus resource while firewall restrictions are in place, your IoT Hub needs to have a managed identity. Once a managed identity is provisioned, follow the steps below to give RBAC permission to your hub's resource identity to access your service bus.

1. In the Azure portal, navigate to your service bus' **Access control (IAM)** tab and click **Add** under the **Add a role assignment** section.
2. Select **Service bus Data Sender** as **role**, **Azure AD user, group, or service principal** as **Assigning access to** and select your IoT Hub's resource name in the drop-down list. Click the **Save** button.
3. Navigate to the **Firewalls and virtual networks** tab in your service bus and enable **Allow access from selected networks** option. Under the **Exceptions** list, check the box for **Allow trusted Microsoft services to access this service bus**. Click the **Save** button.
4. On your IoT Hub's resource page, navigate to **Message routing** tab.
5. Navigate to **Custom endpoints** section and click **Add**. Select **Service bus queue** or **Service Bus topic** (as applicable) as the endpoint type.
6. On the page that shows up, provide a name for your endpoint, select your service bus' namespace and queue or topic (as applicable). Click the **Create** button.

Now your custom service bus endpoint is set up to use your hub's system assigned identity, and it has permission to access your service bus resource despite its firewall restrictions. You can now use this endpoint to set up a routing rule.

Egress connectivity to storage accounts for file upload

IoT Hub's file upload feature allows devices to upload files to a customer-owned storage account. To allow the file upload to function, both devices and IoT Hub need to have connectivity to the storage account. If firewall restrictions are in place on the storage account, your devices need to use any of the supported storage account's mechanism (including [private endpoints](#), [service endpoints](#), or [direct firewall configuration](#)) to gain connectivity. Similarly, if firewall restrictions are in place on the storage account, IoT Hub needs to be configured to access the storage resource via the trusted Microsoft services exception. For this purpose, your IoT Hub must have a managed identity. Once a managed identity is provisioned, follow the steps below to give RBAC permission to your hub's resource identity to access your storage account.

1. In the Azure portal, navigate to your storage account's **Access control (IAM)** tab and click **Add** under the **Add a role assignment** section.
2. Select **Storage Blob Data Contributor** (*not* **Contributor** or **Storage Account Contributor**) as **role**, **Azure AD user, group, or service principal** as **Assigning access to** and select your IoT Hub's resource name in the drop-down list. Click the **Save** button.
3. Navigate to the **Firewalls and virtual networks** tab in your storage account and enable **Allow access from selected networks** option. Under the **Exceptions** list, check the box for **Allow trusted Microsoft services to access this storage account**. Click the **Save** button.
4. On your IoT Hub's resource page, navigate to **File upload** tab.
5. On the page that shows up, select the container that you intend to use in your blob storage, configure the **File notification settings**, **SAS TTL**, **Default TTL**, and **Maximum delivery count** as desired. Select **System Assigned** as the **Authentication type** to your storage endpoint. Click the **Create** button.

Now your storage endpoint for file upload is set up to use your hub's system assigned identity, and it has permission to access your storage resource despite its firewall restrictions.

Egress connectivity to storage accounts for bulk device import/export

IoT Hub supports the functionality to [import/export](#) devices' information in bulk from/to a customer-provided storage blob. To allow bulk import/export feature to function, both devices and IoT Hub need to have connectivity to the storage account.

This functionality requires connectivity from IoT Hub to the storage account. To access a service bus resource while firewall restrictions are in place, your IoT Hub needs to have a managed identity. Once a managed identity is provisioned, follow the steps below to give RBAC permission to your hub's resource identity to access your service bus.

1. In the Azure portal, navigate to your storage account's **Access control (IAM)** tab and click **Add** under the **Add a role assignment** section.
2. Select **Storage Blob Data Contributor** (*not Contributor or Storage Account Contributor*) as **role**, **Azure AD user, group, or service principal** as **Assigning access to** and select your IoT Hub's resource name in the drop-down list. Click the **Save** button.
3. Navigate to the **Firewalls and virtual networks** tab in your storage account and enable **Allow access from selected networks** option. Under the **Exceptions** list, check the box for **Allow trusted Microsoft services to access this storage account**. Click the **Save** button.

You can now use the Azure IoT REST APIs for [creating import export jobs](#) for information on how to use the bulk import/export functionality. You will need to provide the `storageAuthenticationType="identityBased"` in your request body and use `inputBlobContainerUri="https://..."` and `outputBlobContainerUri="https://..."` as the input and output URLs of your storage account, respectively.

Azure IoT Hub SDKs also support this functionality in the service client's registry manager. The following code snippet shows how to initiate an import job or export job in using the C# SDK.

```
// Call an import job on the IoT Hub
JobProperties importJob =
await registryManager.ImportDevicesAsync(
    JobProperties.CreateForImportJob(inputBlobContainerUri, outputBlobContainerUri, null,
    StorageAuthenticationType.IdentityBased),
    cancellationToken);

// Call an export job on the IoT Hub to retrieve all devices
JobProperties exportJob =
await registryManager.ExportDevicesAsync(
    JobProperties.CreateForExportJob(outputBlobContainerUri, true, null,
    StorageAuthenticationType.IdentityBased),
    cancellationToken);
```

To use this version of the Azure IoT SDKs with virtual network support for C#, Java, and Node.js:

1. Create an environment variable named `EnableStorageIdentity` and set its value to `1`.
2. Download the SDK: [Java](#) | [C#](#) | [Nodejs](#)

For Python, download our limited version from GitHub.

1. Navigate to the [GitHub release page](#).
 2. Download the following file, which you'll find at the bottom of the release page under the header named `assets`.
- `azure_iot_hub-2.2.0_limited-py2.py3-none-any.whl`
3. Open a terminal and navigate to the folder with the downloaded file.

4. Run the following command to install the Python Service SDK with support for virtual networks:

```
pip install ./azure_iot_hub-2.2.0_limited-py2.py3-none-any.whl
```

Next steps

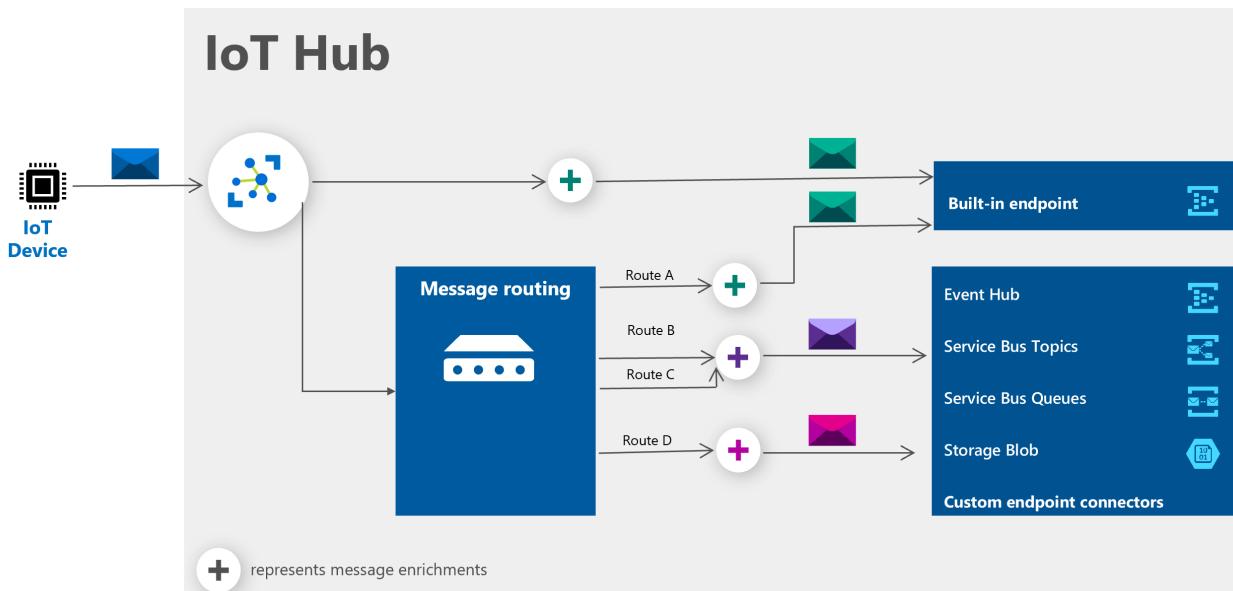
Use the links below to learn more about IoT Hub features:

- [Message routing](#)
- [File upload](#)
- [Bulk device import/export](#)

Message enrichments for device-to-cloud IoT Hub messages

7/29/2020 • 4 minutes to read • [Edit Online](#)

Message enrichments is the ability of the IoT Hub to *stamp* messages with additional information before the messages are sent to the designated endpoint. One reason to use message enrichments is to include data that can be used to simplify downstream processing. For example, enriching device telemetry messages with a device twin tag can reduce load on customers to make device twin API calls for this information.



A message enrichment has three key elements:

- Enrichment name or key
- A value
- One or more **endpoints** for which the enrichment should be applied.

The **key** is a string. A key can only contain alphanumeric characters or these special characters: hyphen (-), underscore (_), and period (.)

The **value** can be any of the following examples:

- Any static string. Dynamic values such as conditions, logic, operations, and functions are not allowed. For example, if you develop a SaaS application that is used by several customers, you can assign an identifier to each customer and make that identifier available in the application. When the application runs, IoT Hub will stamp the device telemetry messages with the customer's identifier, making it possible to process the messages differently for each customer.
- The name of the IoT hub sending the message. This value is `$iothubname`.
- Information from the device twin, such as its path. Examples would be `$twin.tags.field` and `$twin.tags.latitude`.

NOTE

At this time, only \$iothubname, \$twin.tags, \$twin.properties.desired, and \$twin.properties.reported are supported variables for message enrichment.

Message Enrichments are added as application properties to messages sent to chosen endpoint(s).

Applying enrichments

The messages can come from any data source supported by [IoT Hub message routing](#), including the following examples:

- device telemetry, such as temperature or pressure
- device twin change notifications -- changes in the device twin
- device life-cycle events, such as when the device is created or deleted

You can add enrichments to messages that are going to the built-in endpoint of an IoT Hub, or messages that are being routed to custom endpoints such as Azure Blob storage, a Service Bus queue, or a Service Bus topic.

You can add enrichments to messages that are being published to Event Grid by selecting the endpoint as Event Grid. We create a default route in IoT Hub to device telemetry, based on your Event Grid subscription. This single route can handle all of your Event Grid subscriptions. You can configure enrichments for the event grid endpoint after you have created the event grid subscription to device telemetry. For more information, see [IoT Hub and Event Grid](#).

Enrichments are applied per endpoint. If you specify five enrichments to be stamped for a specific endpoint, all messages going to that endpoint are stamped with the same five enrichments.

Enrichments can be configured using the the following methods:

| METHOD | COMMAND | |
|------------------|--|--|
| Portal | Azure portal | See the message enrichments tutorial |
| Azure CLI | <code>az iot hub message-enrichment</code> | |
| Azure PowerShell | <code>Add-AzIoTHubMessageEnrichment</code> | |

Adding message enrichments doesn't add latency to the message routing.

To try out message enrichments, see the [message enrichments tutorial](#)

Limitations

- You can add up to 10 enrichments per IoT Hub for those hubs in the standard or basic tier. For IoT Hubs in the free tier, you can add up to 2 enrichments.
- In some cases, if you are applying an enrichment with a value set to a tag or property in the device twin, the value will be stamped as a string value. For example, if an enrichment value is set to \$twin.tags.field, the messages will be stamped with the string "\$twin.tags.field" rather than the value of that field from the twin. This happens in the following cases:
 - Your IoT Hub is in the basic tier. Basic tier IoT hubs do not support device twins.
 - Your IoT Hub is in the standard tier, but the device sending the message has no device twin.

- Your IoT Hub is in the standard tier, but the device twin path used for the value of the enrichment does not exist. For example, if the enrichment value is set to `$twin.tags.location`, and the device twin does not have a location property under tags, the message is stamped with the string `"$twin.tags.location"`.
- Updates to a device twin can take up to five minutes to be reflected in the corresponding enrichment value.
- The total message size, including the enrichments, can't exceed 256 KB. If a message size exceeds 256 KB, the IoT Hub will drop the message. You can use [IoT Hub metrics](#) to identify and debug errors when messages are dropped. For example, you can monitor `d2c.telemetry.egress.invalid`.
- Message enrichments don't apply to digital twin change events (part of the [IoT Plug and Play public preview](#)).

Pricing

Message enrichments are available for no additional charge. Currently, you are charged when you send a message to an IoT Hub. You are only charged once for that message, even if the message goes to multiple endpoints.

Next steps

Check out these articles for more information about routing messages to an IoT Hub:

- [Message enrichments tutorial](#)
- [Use IoT Hub message routing to send device-to-cloud messages to different endpoints](#)
- [Tutorial: IoT Hub routing](#)

Overview of device management with IoT Hub

7/29/2020 • 5 minutes to read • [Edit Online](#)

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

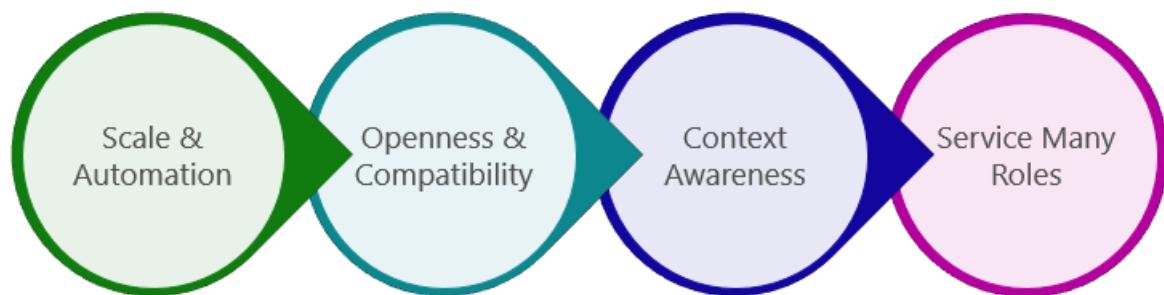
Some of the features mentioned in this article, like cloud-to-device messaging, device twins, and device management, are only available in the standard tier of IoT Hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

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:



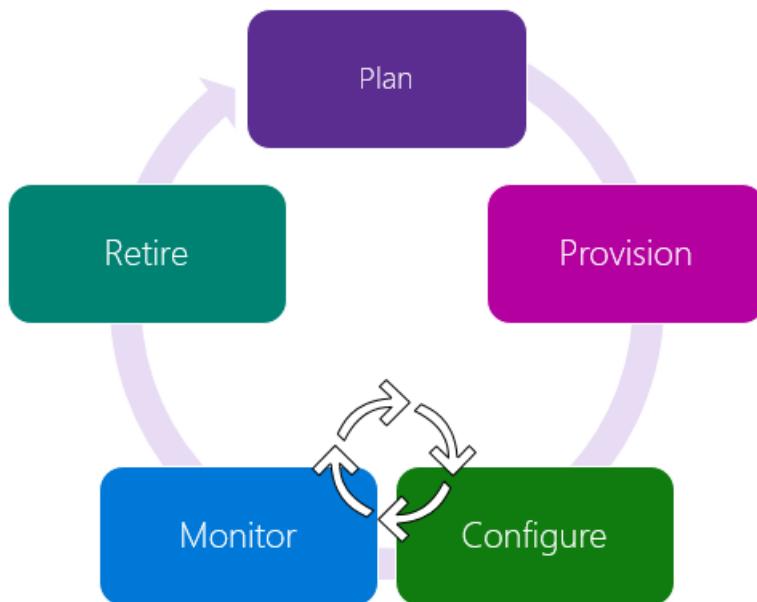
- **Scale and automation:** IoT solutions require simple tools that can automate routine tasks and enable a relatively small operations staff to manage millions of devices. Day-to-day, operators expect to handle device operations remotely, in bulk, and to only be alerted when issues arise that require 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.

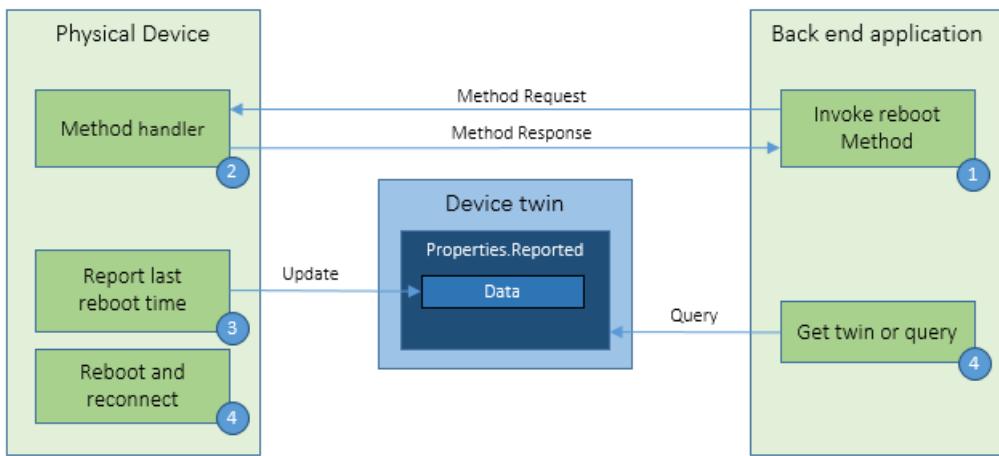
Further reading:

- [How to use device twin properties](#)
- [Manage device identities](#)

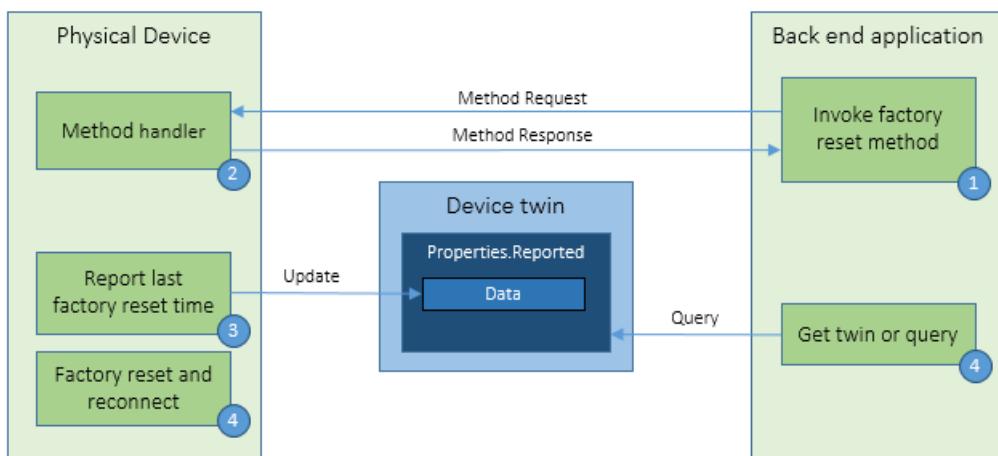
Device management patterns

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.

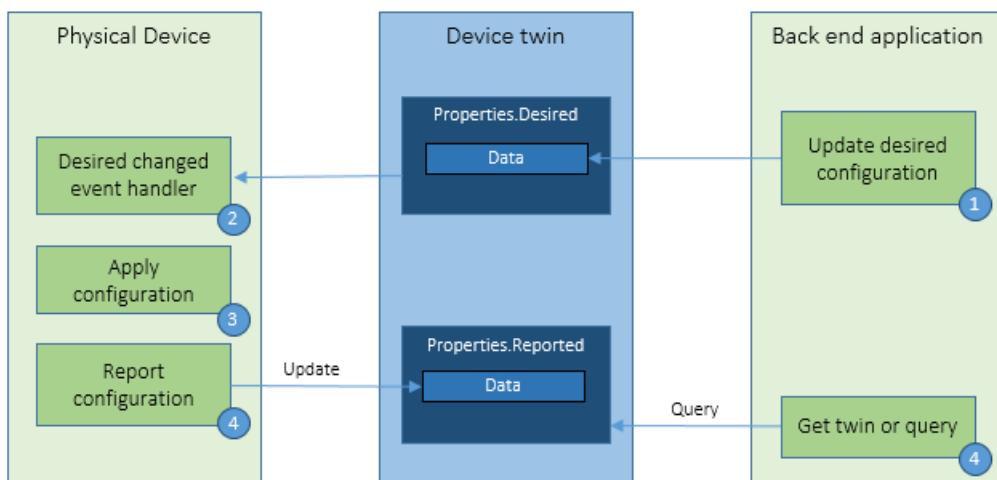
- **Reboot:** The back-end app informs the device through a direct method that it has initiated a reboot. The device uses the reported properties to update the reboot status of the device.



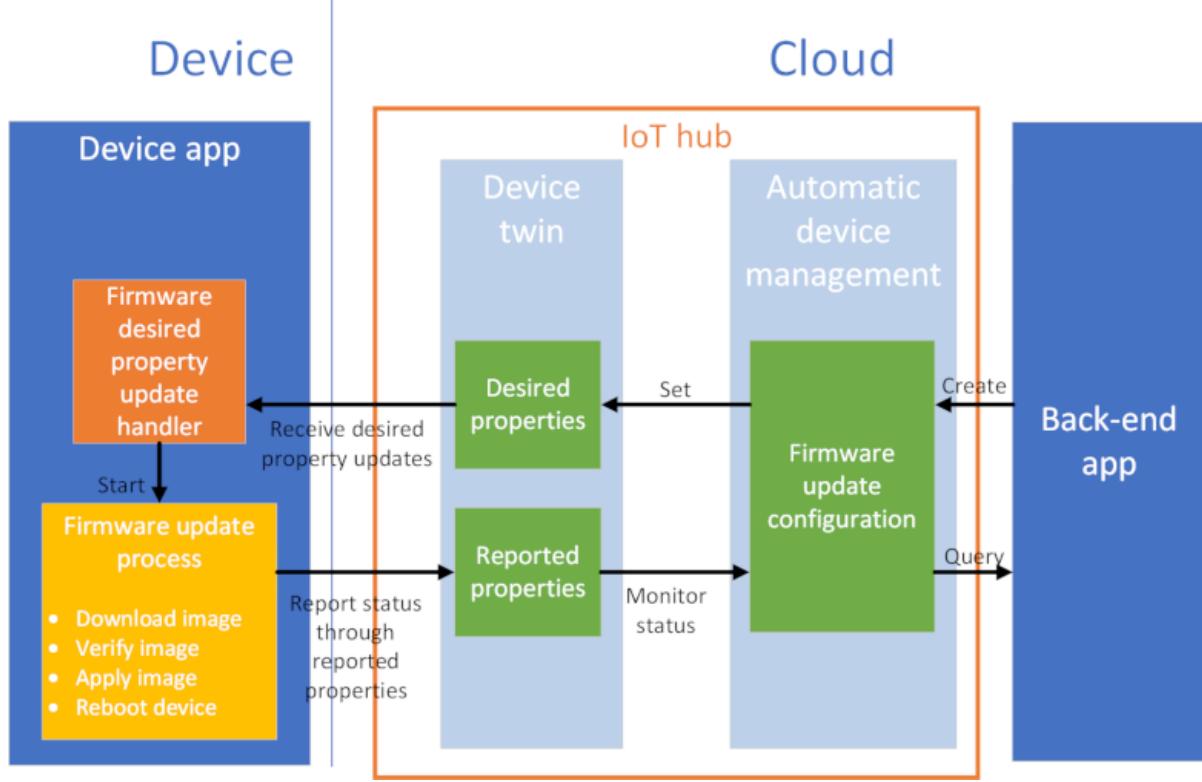
- **Factory Reset:** The back-end app informs the device through a direct method that it has initiated a factory reset. The device uses the reported properties to update the factory reset status of the device.



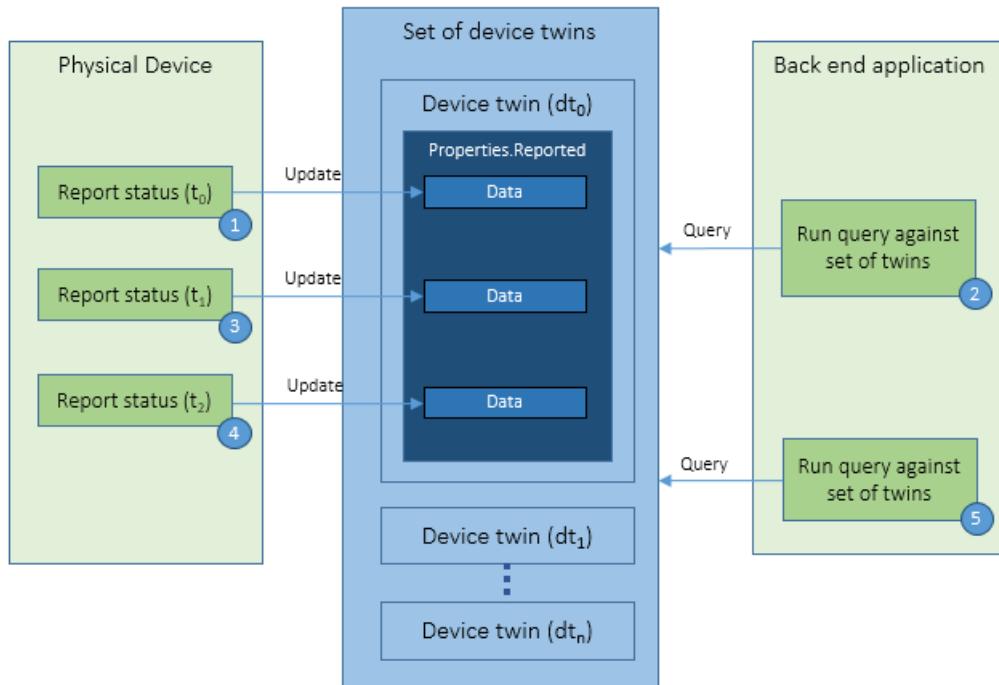
- **Configuration:** The back-end app uses the desired properties to configure software running on the device. The device uses the reported properties to update configuration status of the device.



- **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

7/29/2020 • 2 minutes to read • [Edit Online](#)

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

7/29/2020 • 5 minutes to read • [Edit Online](#)

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 S2.

| CAPABILITY | BASIC TIER | FREE/STANDARD TIER |
|--|------------|--------------------|
| Device-to-cloud telemetry | Yes | Yes |
| Per-device identity | Yes | Yes |
| Message routing, message enrichments, 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 Plug and Play Preview | | 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.

NOTE

The free tier does not support upgrading to basic or standard.

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 |

| API | BASIC TIER | FREE/STANDARD TIER |
|------------------------------------|---------------------------------------|--------------------|
| Get module | Yes | Yes |
| Get registry statistics | Yes | Yes |
| Get services statistics | Yes | Yes |
| Create or update device | Yes | Yes |
| Create or update module | Yes | Yes |
| Query IoT Hub | Yes | Yes |
| Create file upload SAS URI | Yes | Yes |
| Receive device bound notification | Yes | Yes |
| Send device event | Yes | Yes |
| Send module event | AMQP and MQTT only | AMQP and MQTT only |
| Update file upload status | Yes | Yes |
| Bulk device operation | Yes, except for IoT Edge capabilities | Yes |
| Cancel import export job | Yes | Yes |
| Create import export job | Yes | Yes |
| Get import export job | Yes | Yes |
| Get import export jobs | Yes | 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 |

| API | BASIC TIER | FREE/STANDARD TIER |
|------------|------------|--------------------|
| 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 for your IoT hub on a per-unit basis. When you create an IoT hub, you choose its tier and edition, and set the number of units available. You can purchase up to 200 units for the B1, B2, S1, or S2 edition, or up to 10 units for the B3 or S3 edition. After your IoT hub is created, you can change the number of units available within its edition, upgrade or downgrade between editions within its tier (B1 to B2), or upgrade from the basic to the standard tier (B1 to S1) 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 EDITION | 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) |

Device-to-cloud throughput is only one of the metrics you need to consider when designing an IoT solution. For more comprehensive information, see [IoT Hub quotas and throttles](#).

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.

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

7/29/2020 • 10 minutes to read • [Edit Online](#)

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-premises 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.

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. This means that your device-to-cloud messages should continue to work, and the entire device registry is intact. 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.

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 application directly, you will need to [fetch the new Event Hub-compatible endpoint](#) after failover to continue operations. If you use Azure Functions or Azure Stream Analytics to connect the built-in endpoint, you might need to perform a **Restart**.
- When routing to storage, we recommend listing the blobs or files and then iterating over them, to ensure all blobs or files are read without making any assumptions of partition. The partition range could potentially change during a Microsoft-initiated failover or manual failover. You can use the [List Blobs API](#) to enumerate the list of blobs or [List ADLS Gen2 API](#) for the list of files.

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

If your business uptime goals aren't satisfied by the RTO that Microsoft initiated failover provides, 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.

Manual failover is available at no additional cost for IoT hubs created after May 18, 2017

For step-by-step instructions, see [Tutorial: Perform manual failover for an IoT hub](#)

Running test drills

Test drills should not be performed on IoT hubs that are being used in your production environments.

Don't use manual failover to migrate IoT hub to a different region

Manual failover should *not* be used as a mechanism to permanently migrate your hub between the Azure geo paired regions. Doing so increases latency for the operations being performed against the IoT hub from devices homed in the old primary region.

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. You must 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 changes. 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

- [What is Azure IoT Hub?](#)
- [Get started with IoT Hubs \(Quickstart\)](#)
- [Tutorial: Perform manual failover for an IoT hub](#)

How to clone an Azure IoT hub to another region

12/16/2019 • 24 minutes to read • [Edit Online](#)

This article explores ways to clone an IoT Hub and provides some questions you need to answer before you start. Here are several reasons you might want to clone an IoT hub:

- You are moving your company from one region to another, such as from Europe to North America (or vice versa), and you want your resources and data to be geographically close to your new location, so you need to move your hub.
- You are setting up a hub for a development versus production environment.
- You want to do a custom implementation of multi-hub high availability. For more information, see the [How to achieve cross region HA section of IoT Hub high availability and disaster recovery](#).
- You want to increase the number of [partitions](#) configured for your hub. This is set when you first create your hub, and can't be changed. You can use the information in this article to clone your hub and when the clone is created, increase the number of partitions.

To clone a hub, you need a subscription with administrative access to the original hub. You can put the new hub in a new resource group and region, in the same subscription as the original hub, or even in a new subscription. You just can't use the same name because the hub name has to be globally unique.

NOTE

At this time, there's no feature available for cloning an IoT hub automatically. It's primarily a manual process, and thus is fairly error-prone. The complexity of cloning a hub is directly proportional to the complexity of the hub. For example, cloning an IoT hub with no message routing is fairly simple. If you add message routing as just one complexity, cloning the hub becomes at least an order of magnitude more complicated. If you also move the resources used for routing endpoints, it's another order of magnitude more complicated.

Things to consider

There are several things to consider before cloning an IoT hub.

- Make sure that all of the features available in the original location are also available in the new location. Some services are in preview, and not all features are available everywhere.
- Do not remove the original resources before creating and verifying the cloned version. Once you remove a hub, it's gone forever, and there is no way to recover it to check the settings or data to make sure the hub is replicated correctly.
- Many resources require globally unique names, so you must use different names for the cloned versions. You also should use a different name for the resource group to which the cloned hub belongs.
- Data for the original IoT hub is not migrated. This includes telemetry messages, cloud-to-device (C2D) commands, and job-related information such as schedules and history. Metrics and logging results are also not migrated.
- For data or messages routed to Azure Storage, you can leave the data in the original storage account, transfer that data to a new storage account in the new region, or leave the old data in place and create a new storage account in the new location for the new data. For more information on moving data in Blob storage, see [Get started with AzCopy](#).

- Data for Event Hubs and for Service Bus Topics and Queues can't be migrated. This is point-in-time data and is not stored after the messages are processed.
- You need to schedule downtime for the migration. Cloning the devices to the new hub takes time. If you are using the Import/Export method, benchmark testing has revealed that it could take around two hours to move 500,000 devices, and four hours to move a million devices.
- You can copy the devices to the new hub without shutting down or changing the devices.
 - If the devices were originally provisioned using DPS, re-provisioning them updates the connection information stored in each device.
 - Otherwise, you have to use the Import/Export method to move the devices, and then the devices have to be modified to use the new hub. For example, you can set up your device to consume the IoT Hub host name from the twin desired properties. The device will take that IoT Hub host name, disconnect the device from the old hub, and reconnect it to the new one.
- You need to update any certificates you are using so you can use them with the new resources. Also, you probably have the hub defined in a DNS table somewhere — you will need to update that DNS information.

Methodology

This is the general method we recommend for moving an IoT hub from one region to another. For message routing, this assumes the resources are not being moved to the new region. For more information, see the [section on Message Routing](#).

1. Export the hub and its settings to a Resource Manager template.
2. Make the necessary changes to the template, such as updating all occurrences of the name and the location for the cloned hub. For any resources in the template used for message routing endpoints, update the key in the template for that resource.
3. Import the template into a new resource group in the new location. This creates the clone.
4. Debug as needed.
5. Add anything that wasn't exported to the template.

For example, consumer groups are not exported to the template. You need to add the consumer groups to the template manually or use the [Azure portal](#) after the hub is created. There is an example of adding one consumer group to a template in the article [Use an Azure Resource Manager template to configure IoT Hub message routing](#).

6. Copy the devices from the original hub to the clone. This is covered in the section [Managing the devices registered to the IoT hub](#).

How to handle message routing

If your hub uses [custom routing](#), exporting the template for the hub includes the routing configuration, but it does not include the resources themselves. You must choose whether to move the routing resources to the new location or to leave them in place and continue to use them "as is".

For example, say you have a hub in West US that is routing messages to a storage account (also in West US), and you want to move the hub to East US. You can move the hub and have it still route messages to the storage account in West US, or you can move the hub and also move the storage account. There may be a small performance hit from routing messages to endpoint resources in a different region.

You can move a hub that uses message routing pretty easily if you do not also move the resources used for the

routing endpoints.

If the hub uses message routing, you have two choices.

1. Move the resources used for the routing endpoints to the new location.

- You must create the new resources yourself either manually in the [Azure portal](#) or through the use of Resource Manager templates.
- You must rename all of the resources when you create them in the new location, as they have globally unique names.
- You must update the resource names and the resource keys in the new hub's template, before creating the new hub. The resources should be present when the new hub is created.

2. Don't move the resources used for the routing endpoints. Use them "in place".

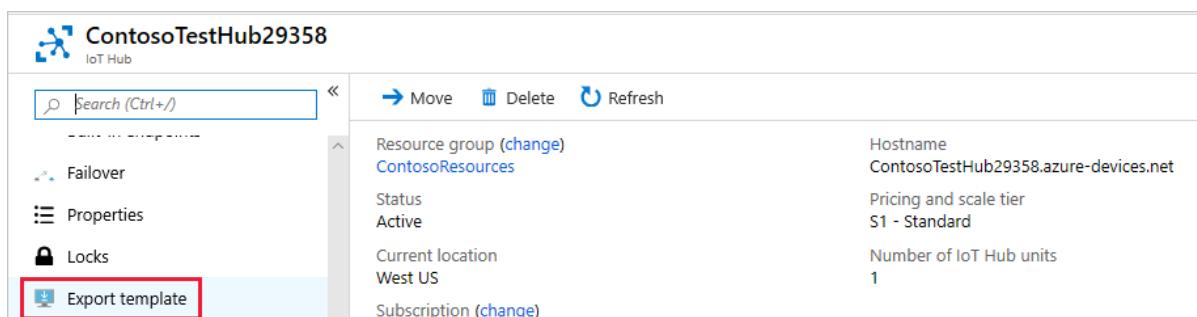
- In the step where you edit the template, you will need to retrieve the keys for each routing resource and put them in the template before you create the new hub.
- The hub still references the original routing resources and routes messages to them as configured.
- You will have a small performance hit because the hub and the routing endpoint resources are not in the same location.

Prepare to migrate the hub to another region

This section provides specific instructions for migrating the hub.

Find the original hub and export it to a resource template.

1. Sign into the [Azure portal](#).
2. Go to **Resource Groups** and select the resource group that contains the hub you want to move. You can also go to **Resources** and find the hub that way. Select the hub.
3. Select **Export template** from the list of properties and settings for the hub.



The screenshot shows the Azure portal interface for an IoT Hub named 'ContosoTestHub29358'. The left sidebar lists 'Failover', 'Properties', 'Locks', and 'Export template'. The main panel displays the following properties:

| | |
|---|---------------------------------------|
| Resource group (change) | Hostname |
| ContosoResources | ContosoTestHub29358.azure-devices.net |
| Status | Pricing and scale tier |
| Active | S1 - Standard |
| Current location | Number of IoT Hub units |
| West US | 1 |
| Subscription (change) | |

4. Select **Download** to download the template. Save the file somewhere you can find it again.

Export template

Document

[Download](#) [Add to library](#) [Deploy](#)

To export related resources, select the resources from the Resource Group view then select the "Export template" option from the toolbar.

Include parameters

Template [Parameters](#) [CLI](#) [PowerShell](#) [.NET](#) [Ruby](#)

► Parameters (1)
 Variables (0)
▼ Resources (1)
 [parameters('IoTHubs_ContosoT...')

```
1  {
2      "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
3      "contentVersion": "1.0.0.0",
4      "parameters": {
5          "IoTHubs_ContosoTestHub29358_name": {
6              "defaultValue": "ContosoTestHub29358",
7              "type": "String"
8          }
9      },
10     "variables": {},
11     "resources": [
```

View the template

1. Go to the Downloads folder (or to whichever folder you used when you exported the template) and find the zip file. Open the zip file and find the file called `template.json`. Select it, then select Ctrl+C to copy the template. Go to a different folder that's not in the zip file and paste the file (Ctrl+V). Now you can edit it.

The following example is for a generic hub with no routing configuration. It is an S1 tier hub (with 1 unit) called `ContosoTestHub29358` in region `westus`. Here is the exported template.

```
{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "IoTHubs_ContosoTestHub29358_name": {
            "defaultValue": "ContosoTestHub29358",
            "type": "String"
        }
    },
    "variables": {},
    "resources": [
        {
            "type": "Microsoft.Devices/IotHubs",
            "apiVersion": "2018-04-01",
            "name": "[parameters('IoTHubs_ContosoTestHub29358_name')]",
            "location": "westus",
            "sku": {
                "name": "S1",
                "tier": "Standard",
                "capacity": 1
            },
            "properties": {
                "operationsMonitoringProperties": {
                    "events": {
                        "None": "None",
                        "Connections": "None",
                        "DeviceTelemetry": "None",
                        "C2DCommands": "None",
                        "DeviceIdentityOperations": "None",
                        "FileUploadOperations": "None",
                        "Routes": "None"
                    }
                }
            }
        }
    ]
}
```

```
,  
    "ipFilterRules": [],  
    "eventHubEndpoints": {  
        "events": {  
            "retentionTimeInDays": 1,  
            "partitionCount": 2,  
            "partitionIds": [  
                "0",  
                "1"  
            ],  
            "path": "contosotesthub29358",  
            "endpoint": "sb://iothub-ns-contosotes-2227755-  
92aefc8b73.servicebus.windows.net/"  
        },  
        "operationsMonitoringEvents": {  
            "retentionTimeInDays": 1,  
            "partitionCount": 2,  
            "partitionIds": [  
                "0",  
                "1"  
            ],  
            "path": "contosotesthub29358-operationmonitoring",  
            "endpoint": "sb://iothub-ns-contosotes-2227755-  
92aefc8b73.servicebus.windows.net/"  
        }  
    },  
    "routing": {  
        "endpoints": {  
            "serviceBusQueues": [],  
            "serviceBusTopics": [],  
            "eventHubs": [],  
            "storageContainers": []  
        },  
        "routes": [],  
        "fallbackRoute": {  
            "name": "$fallback",  
            "source": "DeviceMessages",  
            "condition": "true",  
            "endpointNames": [  
                "events"  
            ],  
            "isEnabled": true  
        }  
    },  
    "storageEndpoints": {  
        "$default": {  
            "sasTtlAsIso8601": "PT1H",  
            "connectionString": "",  
            "containerName": ""  
        }  
    },  
    "messagingEndpoints": {  
        "fileNotifications": {  
            "lockDurationAsIso8601": "PT1M",  
            "ttlAsIso8601": "PT1H",  
            "maxDeliveryCount": 10  
        }  
    },  
    "enableFileUploadNotifications": false,  
    "cloudToDevice": {  
        "maxDeliveryCount": 10,  
        "defaultTtlAsIso8601": "PT1H",  
        "feedback": {  
            "lockDurationAsIso8601": "PT1M",  
            "ttlAsIso8601": "PT1H",  
            "maxDeliveryCount": 10  
        }  
    },  
    "features": "None"
```

```
        }  
    ]  
}
```

Edit the template

You have to make some changes before you can use the template to create the new hub in the new region. Use [VS Code](#) or a text editor to edit the template.

Edit the hub name and location

1. Remove the parameters section at the top -- it is much simpler to just use the hub name because we're not going to have multiple parameters.

```
"parameters": {  
    "IoTHubs_ContosoTestHub29358_name": {  
        "defaultValue": "ContosoTestHub29358",  
        "type": "String"  
    }  
},
```

2. Change the name to use the actual (new) name rather than retrieving it from a parameter (which you removed in the previous step).

For the new hub, use the name of the original hub plus the string *clone* to make up the new name. Start by cleaning up the hub name and location.

Old version:

```
"name": "[parameters('IoTHubs_ContosoTestHub29358_name')]",  
"location": "westus",
```

New version:

```
"name": "ContosoTestHub29358clone",  
"location": "eastus",
```

Next, you'll find that the values for **path** contain the old hub name. Change them to use the new one. These are the path values under **eventHubEndpoints** called **events** and **OperationsMonitoringEvents**.

When you're done, your event hub endpoints section should look like this:

```

"eventHubEndpoints": {
    "events": {
        "retentionTimeInDays": 1,
        "partitionCount": 2,
        "partitionIds": [
            "0",
            "1"
        ],
        "path": "contosotesthub29358clone",
        "endpoint": "sb://iothub-ns-contosotes-2227755-92aefc8b73.servicebus.windows.net/"
    },
    "operationsMonitoringEvents": {
        "retentionTimeInDays": 1,
        "partitionCount": 2,
        "partitionIds": [
            "0",
            "1"
        ],
        "path": "contosotesthub29358clone-operationmonitoring",
        "endpoint": "sb://iothub-ns-contosotes-2227755-92aefc8b73.servicebus.windows.net/"
    }
}

```

Update the keys for the routing resources that are not being moved

When you export the Resource Manager template for a hub that has routing configured, you will see that the keys for those resources are not provided in the exported template -- their placement is denoted by asterisks. You must fill them in by going to those resources in the portal and retrieving the keys **before** you import the new hub's template and create the hub.

1. Retrieve the keys required for any of the routing resources and put them in the template. You can retrieve the key(s) from the resource in the [Azure portal](#).

For example, if you are routing messages to a storage container, find the storage account in the portal.

Under the Settings section, select **Access keys**, then copy one of the keys. Here's what the key looks like when you first export the template:

```

"connectionString": "DefaultEndpointsProtocol=https;
AccountName=fabrikamstorage1234;AccountKey=*****",
"containerName": "fabrikamresults",

```

2. After you retrieve the account key for the storage account, put it in the template in the clause

`AccountKey=*****` in the place of the asterisks.

3. For service bus queues, get the Shared Access Key matching the SharedAccessKeyName. Here is the key and the `SharedAccessKeyName` in the json:

```

"connectionString": "Endpoint=sb://fabrikamsbnamespace1234.servicebus.windows.net:5671/;
SharedAccessKeyName=iothubroutes_FabrikamResources;
SharedAccessKey=*****;
EntityPath=fabrikamsbqueue1234",

```

4. The same applies for the Service Bus Topics and Event Hub connections.

Create the new routing resources in the new location

This section only applies if you are moving the resources used by the hub for the routing endpoints.

If you want to move the routing resources, you must manually set up the resources in the new location. You can create the routing resources using the [Azure portal](#), or by exporting the Resource Manager template for each of the resources used by the message routing, editing them, and importing them. After the resources are set up, you can

import the hub's template (which includes the routing configuration).

1. Create each resource used by the routing. You can do this manually using the [Azure portal](#), or create the resources using Resource Manager templates. If you want to use templates, these are the steps to follow:
 - a. For each resource used by the routing, export it to a Resource Manager template.
 - b. Update the name and location of the resource.
 - c. Update any cross-references between the resources. For example, if you create a template for a new storage account, you need to update the storage account name in that template and any other template that references it. In most cases, the routing section in the template for the hub is the only other template that references the resource.
 - d. Import each of the templates, which deploys each resource.

Once the resources used by the routing are set up and running, you can continue.

2. In the template for the IoT hub, change the name of each of the routing resources to its new name, and update the location if needed.

Now you have a template that will create a new hub that looks almost exactly like the old hub, depending on how you decided to handle the routing.

Move -- create the new hub in the new region by loading the template

Create the new hub in the new location using the template. If you have routing resources that are going to move, the resources should be set up in the new location and the references in the template updated to match. If you are not moving the routing resources, they should be in the template with the updated keys.

1. Sign into the [Azure portal](#).
2. Select **Create a resource**.
3. In the search box, put in "template deployment" and select Enter.
4. Select **template deployment (deploy using custom templates)**. This takes you to a screen for the Template deployment. Select **Create**. You see this screen:

Custom deployment

Deploy from a custom template

Learn about template deployment

[Read the docs](#)

[Build your own template in the editor](#)

Common templates

[Create a Linux virtual machine](#)

[Create a Windows virtual machine](#)

[Create a web app](#)

[Create a SQL database](#)

Load a GitHub quickstart template

Select a template (disclaimer)

Type to start filtering...

5. Select **Build your own template in the editor**, which enables you to upload your template from a file.
6. Select **Load file**.

Edit template

Edit your Azure Resource Manager template

[Add resource](#)

[Quickstart template](#)

[Load file](#)

[Download](#)

[Parameters \(0\)](#)

[Variables \(0\)](#)

[Resources \(0\)](#)

```
1  {
2      "$schema": "https://schema.management.azure.com/deploymentTemplate.json#",
3      "contentVersion": "1.0.0.0",
4      "parameters": {},
5      "resources": []
6 }
```

7. Browse for the new template you edited and select it, then select **Open**. It loads your template in the edit window. Select **Save**.

Custom deployment

Deploy from a custom template

TEMPLATE

Customized template
1 resource

Edit template Learn more

BASICS

* Subscription: Content Developer testing (robinsh)

* Resource group: ContosoResourcesClone (selected) | Create new

* Location: (US) East US

TERMS AND CONDITIONS

Azure Marketplace Terms | Azure Marketplace

By clicking "Purchase," I (a) agree to the applicable legal terms associated with the offering; (b) authorize Microsoft to charge or bill my current payment method for the fees associated the offering(s), including applicable taxes, with the same billing frequency as my Azure subscription, until I discontinue use of the offering(s); and (c) agree that, if the deployment involves 3rd party offerings, Microsoft may share my contact information and other details of such deployment with the publisher of that offering.

I agree to the terms and conditions stated above

Purchase

8. Fill in the following fields.

Subscription: select the subscription to use.

Resource group: create a new resource group in a new location. If you already have a new one set up, you can select it instead of creating a new one.

Location: If you selected an existing resource group, this is filled in for you to match the location of the resource group. If you created a new resource group, this will be its location.

I agree checkbox: this basically says that you agree to pay for the resource(s) you're creating.

9. Select the **Purchase** button.

The portal now validates your template and deploys your cloned hub. If you have routing configuration data, it will be included in the new hub, but will point at the resources in the prior location.

Managing the devices registered to the IoT hub

Now that you have your clone up and running, you need to copy all of the devices from the original hub to the clone.

There are multiple ways to accomplish this. You either originally used [Device Provisioning Service \(DPS\)](#) to provision the devices, or you didn't. If you did, this is not difficult. If you did not, this can be very complicated.

If you did not use DPS to provision your devices, you can skip the next section and start with [Using Import/Export to move the devices to the new hub](#).

Using DPS to re-provision the devices in the new hub

To use DPS to move the devices to the new location, see [How to re-provision devices](#). When you're finished, you can view the devices in the [Azure portal](#) and verify they are in the new location.

Go to the new hub using the [Azure portal](#). Select your hub, then select **IoT Devices**. You see the devices that were re-provisioned to the cloned hub. You can also view the properties for the cloned hub.

If you have implemented routing, test and make sure your messages are routed to the resources correctly.

Committing the changes after using DPS

This change has been committed by the DPS service.

Rolling back the changes after using DPS.

If you want to roll back the changes, re-provision the devices from the new hub to the old one.

You are now finished migrating your hub and its devices. You can skip to [Clean-up](#).

Using Import-Export to move the devices to the new hub

The application targets .NET Core, so you can run it on either Windows or Linux. You can download the sample, retrieve your connection strings, set the flags for which bits you want to run, and run it. You can do this without ever opening the code.

Downloading the sample

1. Use the IoT C# samples from this page: [Azure IoT Samples for C#](#). Download the zip file and unzip it on your computer.
2. The pertinent code is in `./iot-hub/Samples/service/ImportExportDevicesSample`. You don't need to view or edit the code in order to run the application.
3. To run the application, specify three connection strings and five options. You pass this data in as command-line arguments or use environment variables, or use a combination of the two. We're going to pass the options in as command line arguments, and the connection strings as environment variables.

The reason for this is because the connection strings are long and ungainly, and unlikely to change, but you might want to change the options and run the application more than once. To change the value of an environment variable, you have to close the command window and Visual Studio or VS Code, whichever you are using.

Options

Here are the five options you specify when you run the application. We'll put these on the command line in a minute.

- **addDevices** (argument 1) -- set this to true if you want to add virtual devices that are generated for you. These are added to the source hub. Also, set **numToAdd** (argument 2) to specify how many devices you want to add. The maximum number of devices you can register to a hub is one million. The purpose of this option is for testing -- you can generate a specific number of devices, and then copy them to another hub.
- **copyDevices** (argument 3) -- set this to true to copy the devices from one hub to another.
- **deleteSourceDevices** (argument 4) -- set this to true to delete all of the devices registered to the source hub. We recommend waiting until you are certain all of the devices have been transferred before you run this. Once you delete the devices, you can't get them back.
- **deleteDestDevices** (argument 5) -- set this to true to delete all of the devices registered to the destination hub (the clone). You might want to do this if you want to copy the devices more than once.

The basic command will be `dotnet run` -- this tells .NET to build the local csproj file and then run it. You add your command-line arguments to the end before you run it.

Your command-line will look like these examples:

```
// Format: dotnet run add-devices num-to-add copy-devices delete-source-devices delete-destination-devices

// Add 1000 devices, don't copy them to the other hub, or delete them.
// The first argument is true, numToAdd is 50, and the other arguments are false.
dotnet run true 1000 false false false

// Copy the devices you just added to the other hub; don't delete anything.
// The first argument is false, numToAdd is 0, copy-devices is true, and the delete arguments are both
false
dotnet run false 0 true false false
```

Using environment variables for the connection strings

1. To run the sample, you need the connection strings to the old and new IoT hubs, and to a storage account you can use for temporary work files. We will store the values for these in environment variables.
2. To get the connection string values, sign in to the [Azure portal](#).
3. Put the connection strings somewhere you can retrieve them, such as NotePad. If you copy the following, you can paste the connection strings in directly where they go. Don't add spaces around the equal sign, or it changes the variable name. Also, you do not need double-quotes around the connection strings. If you put quotes around the storage account connection string, it won't work.

For Windows, this is how you set the environment variables:

```
SET IOTHUB_CONN_STRING=<put connection string to original IoT Hub here>
SET DEST_IOTHUB_CONN_STRING=<put connection string to destination or clone IoT Hub here>
SET STORAGE_ACCT_CONN_STRING=<put connection string to the storage account here>
```

For Linux, this is how you define the environment variables:

```
export IOTHUB_CONN_STRING=<put connection string to original IoT Hub here>
export DEST_IOTHUB_CONN_STRING=<put connection string to destination or clone IoT Hub here>
export STORAGE_ACCT_CONN_STRING=<put connection string to the storage account here>
```

4. For the IoT hub connection strings, go to each hub in the portal. You can search in **Resources** for the hub. If you know the Resource Group, you can go to **Resource groups**, select your resource group, and then select the hub from the list of assets in that resource group.
5. Select **Shared access policies** from the Settings for the hub, then select **iothubowner** and copy one of the connection strings. Do the same for the destination hub. Add them to the appropriate SET commands.
6. For the storage account connection string, find the storage account in **Resources** or under its **Resource group** and open it.
7. Under the Settings section, select **Access keys** and copy one of the connection strings. Put the connection string in your text file for the appropriate SET command.

Now you have the environment variables in a file with the SET commands, and you know what your command-line arguments are. Let's run the sample.

Running the sample application and using command-line arguments

1. Open a command prompt window. Select Windows and type in `command prompt` to get the command

prompt window.

2. Copy the commands that set the environment variables, one at a time, and paste them into the command prompt window and select Enter. When you're finished, type `SET` in the command prompt window to see your environment variables and their values. Once you've copied these into the command prompt window, you don't have to copy them again, unless you open a new command prompt window.
3. In the command prompt window, change directories until you are in `./ImportExportDevicesSample` (where the `ImportExportDevicesSample.csproj` file exists). Then type the following, and include your command-line arguments.

```
// Format: dotnet run add-devices num-to-add copy-devices delete-source-devices delete-destination-devices  
dotnet run arg1 arg2 arg3 arg4 arg5
```

The `dotnet` command builds and runs the application. Because you are passing in the options when you run the application, you can change the values of them each time you run the application. For example, you may want to run it once and create new devices, then run it again and copy those devices to a new hub, and so on. You can also perform all the steps in the same run, although we recommend not deleting any devices until you are certain you are finished with the cloning. Here is an example that creates 1000 devices and then copies them to the other hub.

```
// Format: dotnet run add-devices num-to-add copy-devices delete-source-devices delete-destination-devices  
  
// Add 1000 devices, don't copy them to the other hub or delete them.  
dotnet run true 1000 false false false  
  
// Do not add any devices. Copy the ones you just created to the other hub; don't delete anything.  
dotnet run false 0 true false false
```

After you verify that the devices were copied successfully, you can remove the devices from the source hub like this:

```
// Format: dotnet run add-devices num-to-add copy-devices delete-source-devices delete-destination-devices  
// Delete the devices from the source hub.  
dotnet run false 0 false true false
```

Running the sample application using Visual Studio

1. If you want to run the application in Visual Studio, change your current directory to the folder where the `IoTHubServiceSamples.sln` file resides. Then run this command in the command prompt window to open the solution in Visual Studio. You must do this in the same command window where you set the environment variables, so those variables are known.

```
IoTHubServiceSamples.sln
```

2. Right-click on the project `ImportExportDevicesSample` and select **Set as startup project**.
3. Set the variables at the top of `Program.cs` in the `ImportExportDevicesSample` folder for the five options.

```

// Add randomly created devices to the source hub.
private static bool addDevices = true;
//If you ask to add devices, this will be the number added.
private static int numToAdd = 0;
// Copy the devices from the source hub to the destination hub.
private static bool copyDevices = false;
// Delete all of the devices from the source hub. (It uses the IoTHubConnectionString).
private static bool deleteSourceDevices = false;
// Delete all of the devices from the destination hub. (Uses the DestIoTHubConnectionString).
private static bool deleteDestDevices = false;

```

4. Select F5 to run the application. After it finishes running, you can view the results.

View the results

You can view the devices in the [Azure portal](#) and verify they are in the new location.

1. Go to the new hub using the [Azure portal](#). Select your hub, then select **IoT Devices**. You see the devices you just copied from the old hub to the cloned hub. You can also view the properties for the cloned hub.
2. Check for import/export errors by going to the Azure storage account in the [Azure portal](#) and looking in the `devicefiles` container for the `ImportErrors.log`. If this file is empty (the size is 0), there were no errors. If you try to import the same device more than once, it rejects the device the second time and adds an error message to the log file.

Committing the changes

At this point, you have copied your hub to the new location and migrated the devices to the new clone. Now you need to make changes so the devices work with the cloned hub.

To commit the changes, here are the steps you need to perform:

- Update each device to change the IoT Hub host name to point the IoT Hub host name to the new hub. You should do this using the same method you used when you first provisioned the device.
- Change any applications you have that refer to the old hub to point to the new hub.
- After you're finished, the new hub should be up and running. The old hub should have no active devices and be in a disconnected state.

Rolling back the changes

If you decide to roll back the changes, here are the steps to perform:

- Update each device to change the IoT Hub Hostname to point the IoT Hub Hostname for the old hub. You should do this using the same method you used when you first provisioned the device.
- Change any applications you have that refer to the new hub to point to the old hub. For example, if you are using Azure Analytics, you may need to reconfigure your [Azure Stream Analytics input](#).
- Delete the new hub.
- If you have routing resources, the configuration on the old hub should still point to the correct routing configuration, and should work with those resources after the hub is restarted.

Checking the results

To check the results, change your IoT solution to point to your hub in the new location and run it. In other words, perform the same actions with the new hub that you performed with the previous hub and make sure they work correctly.

If you have implemented routing, test and make sure your messages are routed to the resources correctly.

Clean-up

Don't clean up until you are really certain the new hub is up and running and the devices are working correctly. Also be sure to test the routing if you are using that feature. When you're ready, clean up the old resources by performing these steps:

- If you haven't already, delete the old hub. This removes all of the active devices from the hub.
- If you have routing resources that you moved to the new location, you can delete the old routing resources.

Next steps

You have cloned an IoT hub into a new hub in a new region, complete with the devices. For more information about performing bulk operations against the identity registry in an IoT Hub, see [Import and export IoT Hub device identities in bulk](#).

For more information about IoT Hub and development for the hub, please see the following articles.

- [IoT Hub developer's guide](#)
- [IoT Hub routing tutorial](#)
- [IoT Hub device management overview](#)
- If you want to deploy the sample application, please see [.NET Core application deployment](#).

IoT Hub IP addresses

7/29/2020 • 3 minutes to read • [Edit Online](#)

The IP address prefixes of IoT Hub public endpoints are published periodically under the [AzureIoTHub service tag](#).

NOTE

For devices that are deployed inside of on-premises networks, Azure IoT Hub supports VNET connectivity integration with private endpoints. See [IoT Hub support for VNet](#) for more information.

You may use these IP address prefixes to control connectivity between IoT Hub and your devices or network assets in order to implement a variety of network isolation goals:

| GOAL | APPLICABLE SCENARIOS | APPROACH |
|--|--|--|
| Ensure your devices and services communicate with IoT Hub endpoints only | Device-to-cloud , and cloud-to-device messaging, direct methods , device and module twins and device streams | Use AzureIoTHub and EventHub service tags to discover IoT Hub, and Event Hub IP address prefixes and configure ALLOW rules on your devices' and services' firewall setting for those IP address prefixes accordingly; drop traffic to other destination IP addresses you do not want the devices or services to communicate with. |
| Ensure your IoT Hub device endpoint receives connections only from your devices and network assets | Device-to-cloud , and cloud-to-device messaging, direct methods , device and module twins and device streams | Use IoT Hub IP filter feature to allow connections from your devices and network asset IP addresses (see limitations section). |
| Ensure your routes' custom endpoint resources (storage accounts, service bus and event hubs) are reachable from your network assets only | Message routing | Follow your resource's guidance on restrict connectivity (for example via firewall rules , private links , or service endpoints); use AzureIoTHub service tags to discover IoT Hub IP address prefixes and add ALLOW rules for those IP prefixes on your resource's firewall configuration (see limitations section). |

Best practices

- When adding ALLOW rules in your devices' firewall configuration, it is best to provide specific [ports used by applicable protocols](#).
- The IP address prefixes of IoT hub are subject to change. These changes are published periodically via service tags before taking effect. It is therefore important that you develop processes to regularly retrieve and use the latest service tags. This process can be automated via the [service tags discovery API](#). Note that Service tags discovery API is still in preview and in some cases may not produce the full list of tags and IP addresses. Until discovery API is generally available, consider using the [service tags in downloadable JSON format](#).
- Use the [AzureIoTHub.\[region name\]](#) tag to identify IP prefixes used by IoT hub endpoints in a specific region. To account for datacenter disaster recovery, or [regional failover](#) ensure connectivity to IP prefixes of your IoT

Hub's geo-pair region is also enabled.

- Setting up firewall rules in IoT Hub may block off connectivity needed to run Azure CLI and PowerShell commands against your IoT Hub. To avoid this, you can add ALLOW rules for your clients' IP address prefixes to re-enable CLI or PowerShell clients to communicate with your IoT Hub.

Limitations and workarounds

- IoT Hub IP filter feature has a limit of 10 rules. This limit can be raised via requests through Azure Customer Support.
- Your configured [IP filtering rules](#) are only applied on your IoT Hub IP endpoints and not on your IoT hub's built-in Event Hub endpoint. If you also require IP filtering to be applied on the Event Hub where your messages are stored, you may do so bringing your own Event Hub resource where you can configure your desired IP filtering rules directly. To do so, you need to provision your own Event Hub resource and set up [message routing](#) to send your messages to that resource instead of your IoT Hub's built-in Event Hub. Finally, as discussed in the table above, to enable message routing functionality you also need to allow connectivity from IoT Hub's IP address prefixes to your provisioned Event Hub resource.
- When routing to a storage account, allowing traffic from IoT Hub's IP address prefixes is only possible when the storage account is in a different region as your IoT Hub.

Support for IPv6

IPv6 is currently not supported on IoT Hub.

Support additional protocols for IoT Hub

7/29/2020 • 2 minutes to read • [Edit Online](#)

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

11/11/2019 • 5 minutes to read • [Edit Online](#)

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. IoT Hub publishes both [device events](#) and telemetry events.

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 and events | Yes, message routing can be used for telemetry data, report device twin changes, device lifecycle events, and digital twin change events (part of the IoT Plug and Play public preview). | Yes, Event Grid can be used for telemetry data but can also 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. |
| Filtering | Rich filtering on message application properties, message system properties, message body, device twin tags, and device twin properties. Filtering isn't applied to digital twin change events. For examples, see Message Routing Query Syntax . | Filtering based on event type, subject type and attributes in each event. For examples, see Understand filtering events in Event Grid Subscriptions . When subscribing to telemetry events, you can apply additional filters on the data to filter on message properties, message body and device twin in your IoT Hub, before publishing to Event Grid. See how to filter events . |

| 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 • Queue Storage • Microsoft Flow • Third-party services through WebHooks <p>500 endpoints per IoT Hub are supported. 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 |
|----------------------------|---|---|
| Maximum message size | 256 KB, device-to-cloud | 256 KB, device-to-cloud |
| 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 telemetry data, device created, deleted, connected, and disconnected. When subscribing to telemetry events, you can apply additional filters on the data to filter on message properties, message body and device twin in your IoT Hub, before publishing to Event Grid. See [how to filter events](#).

- **What endpoints need to receive this information?**

IoT Hub message routing supports limited number of unique endpoints and endpoint types, 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 500 endpoints per IoT Hub and a larger variety of endpoint types. It natively integrates with Azure Functions, Logic Apps, Storage and Service Bus queues, and also works with webhooks to extend sending data 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. For those cases in which absolute order of messages is significant and/or in which a consumer needs a trustworthy unique identifier for messages, we recommend using message routing.

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

4/22/2020 • 6 minutes to read • [Edit Online](#)

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](#) 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 run for the first time shortly after the configuration is created and then at five minute intervals. They also benefit from the 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 Device SDKs Platform Support

1/16/2020 • 5 minutes to read • [Edit Online](#)

Microsoft strives to continually expand the universe of Azure IoT Hub capable devices. Microsoft publishes open-source device SDKs on GitHub to help connect devices to Azure IoT Hub and the Device Provisioning Service. The device SDKs are available for C, .NET (C#), Java, Node.js, and Python. Microsoft tests each SDK to ensure that it runs on the supported configurations detailed for it in the [Microsoft SDKs and device platform support](#) section.

In addition to the device SDKs, Microsoft provides several other avenues to empower customers and developers to connect their devices to Azure IoT:

- Microsoft collaborates with several partner companies to help them publish development kits, based on the Azure IoT C SDK, for their hardware platforms.
- Microsoft works with Microsoft trusted partners to provide an ever-expanding set of devices that have been tested and certified for Azure IoT. For a current list of these devices, see the [Azure certified for IoT device catalog](#).
- Microsoft provides a platform abstraction layer (PAL) in the Azure IoT Hub Device C SDK that helps developers to easily port the SDK to their platform. To learn more, see the [C SDK porting guidance](#).

This topic provides information about the Microsoft SDKs and the platform configurations they support, as well as each of the other options listed above.

Microsoft SDKs and device platform support

Microsoft publishes open-source SDKs on GitHub for the following languages: C, .NET (C#), Node.js, Java, and Python. The SDKs and their dependencies are listed in this section. The SDKs are supported on any device platform that satisfies these dependencies.

For each of the listed SDKs, Microsoft:

- Continuously builds and runs end-to-end tests against the master branch of the relevant SDK in GitHub on several popular platforms. To provide test coverage across different compiler versions, we generally test against the latest LTS version and the most popular version.
- Provides installation guidance or installation packages if applicable.
- Fully supports the SDKs on GitHub with open-source code, a path for customer contributions, and product team engagement with GitHub issues.

C SDK

The [Azure IoT Hub C device SDK](#) is tested with and supports the following configurations.

| OS | TLS LIBRARY | ADDITIONAL REQUIREMENTS |
|-------------------|------------------------------|---|
| Linux | OpenSSL, WolfSSL, or BearSSL | Berkeley sockets Portable Operating System Interface (POSIX) |
| iOS 12.2 | OpenSSL | XCode emulated in OSX 10.13.4 |
| Windows 10 family | SChannel | |

| OS | TLS LIBRARY | ADDITIONAL REQUIREMENTS |
|-----------------|-------------|-------------------------------------|
| Mbed OS 5.4 | Mbed TLS 2 | MXChip IoT dev kit |
| Azure Sphere OS | WolfSSL | Azure Sphere MT3620 |
| Arduino | BearSSL | ESP32 or ESP8266 |

Python SDK

The [Azure IoT Hub Python device SDK](#) is tested with and supports the following configurations.

| OS | COMPILER |
|-------------------|----------------------------|
| Linux | Python 2.7.*, 3.5 or later |
| MacOS High Sierra | Python 2.7.*, 3.5 or later |
| Windows 10 family | Python 2.7.*, 3.5 or later |

Only Python version 3.5.3 or later support the asynchronous APIs, we recommend using version 3.7 or later.

.NET SDK

The [Azure IoT Hub .NET \(C#\) device SDK](#) is tested with and supports the following configurations.

| OS | STANDARD |
|------------------------------------|--|
| Linux | .NET Core 2.1 |
| Windows 10 Desktop and Server SKUs | .NET Core 2.1, .NET Framework 4.5.1, or .NET Framework 4.7 |

The .NET SDK can also be used with Windows IoT Core with the [Azure Device Agent](#) or a custom NTService that can use RPC to communicate with UWP applications.

Node.js SDK

The [Azure IoT Hub Node.js device SDK](#) is tested with and supports the following configurations.

| OS | NODE VERSION |
|-------------------|-----------------|
| Linux | LTS and Current |
| Windows 10 family | LTS and Current |

Java SDK

The [Azure IoT Hub Java device SDK](#) is tested with and supports the following configurations.

| OS | JAVA VERSION |
|-----------------------|--------------|
| Android API 28 | Java 8 |
| Linux x64 | Java 8 |
| Windows 10 family x64 | Java 8 |

Partner supported development kits

Microsoft works with various partners to provide development kits for several microprocessor architectures. These partners have ported the Azure IoT C SDK to their platform. Partners create and maintain the platform abstraction layer (PAL) of the SDK. Microsoft works with these partners to provide extended support.

| 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-AZURE P-NUCLEO-AZURE FP-CLD-AZURE | Support |
| Texas Instruments | CC3220SF LaunchPad CC3220S LaunchPad CC3235SF LaunchPad CC3235S LaunchPad MSP432E4 LaunchPad | Azure IoT Plugin for SimpleLink | TI E2E Forum TI E2E Forum for CC3220 TI E2E Forum for MSP432E4 |

Porting the Microsoft Azure IoT C SDK

If your device platform isn't covered by one of the previous sections, you can consider porting the Azure IoT C SDK. Porting the C SDK primarily involves implementing the platform abstraction layer (PAL) of the SDK. The PAL defines primitives that provide the glue between your device and higher-level functions in the SDK. For more information, see [Porting Guidance](#).

Microsoft partners and certified Azure IoT devices

Microsoft works with a number of partners to continually expand the Azure IoT universe with Azure IoT tested and certified devices.

- To browse Azure IoT certified devices, see [Microsoft Azure Certified for IoT Device Catalog](#).
- To learn more about the Azure Certified for IoT ecosystem, see [Join the Certified for IoT ecosystem](#).

Connecting to IoT Hub without an SDK

If you're not able to use one of the IoT Hub device SDKs, you can connect directly to IoT Hub using the [IoT Hub REST APIs](#) from any application capable of sending and receiving HTTPS requests and responses.

Support and other resources

If you experience problems while using the Azure IoT device SDKs, there are several ways to seek support. You can try one of the following channels:

Reporting bugs – Bugs in the device SDKs can be reported on the issues page of the relevant GitHub project. Fixes rapidly make their way from the project in to product updates.

- [Azure IoT Hub C SDK issues](#)

- [Azure IoT Hub .NET \(C#\) SDK issues](#)
- [Azure IoT Hub Java SDK issues](#)
- [Azure IoT Hub Node.js SDK issues](#)
- [Azure IoT Hub Python SDK issues](#)

Microsoft Customer Support team – Users who have a [support plan](#) can engage the Microsoft Customer Support team by creating a new support request directly from the [Azure portal](#).

Feature requests – Azure IoT feature requests are tracked via the product's [User Voice page](#).

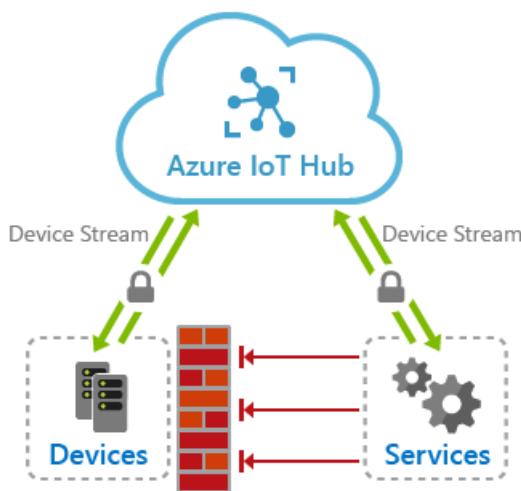
Next steps

- [Device and service SDKs](#)
- [Porting Guidance](#)

IoT Hub Device Streams (preview)

7/29/2020 • 9 minutes to read • [Edit Online](#)

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.

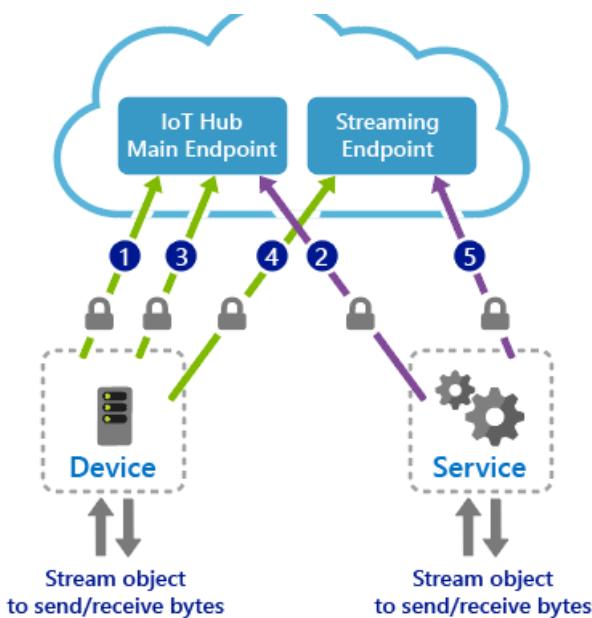
Device stream workflows

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:

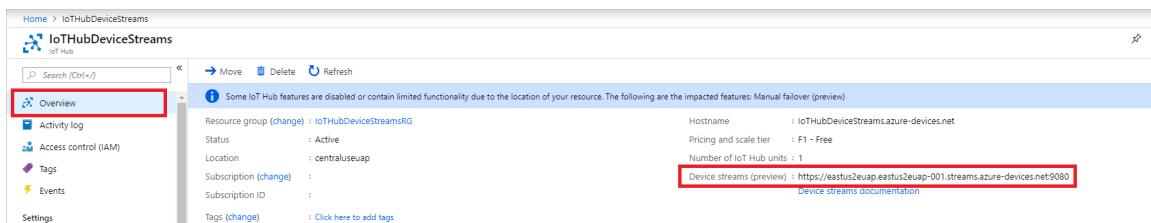
- The handshake process must complete within 60 seconds (step 2 through 5), otherwise the handshake would fail with a timeout and the service will be notified accordingly.
- After the stream creation flow above completes, the streaming endpoint will act as a proxy and will transfer traffic between the 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:



The screenshot shows the Azure IoT Hub Device Streams Overview page. The left sidebar has links for Home, IoTHubDeviceStreams, IoT Hub, Overview (which is selected and highlighted with a red box), Activity log, Access control (IAM), Tags, Events, and Settings. The main content area has a message: "Some IoT Hub features are disabled or contain limited functionality due to the location of your resource. The following are the impacted features: Manual failover (preview)". Below this, it shows resource details: Resource group (change) : IoTHubDeviceStreamsRG, Status : Active, Location : centraluseuap, Subscription (change) : , Subscription ID : . On the right, it shows Hostname : IoTHubDeviceStreams.azure-devices.net, Pricing and scale tier : F1 - Free, Number of IoT Hub units : 1, and Device streams (preview) : https://eastus2euap.eastus2euap-001.streams.azure-devices.net:9080. A link to "Device streams documentation" is also present.

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"
  ]
}
```

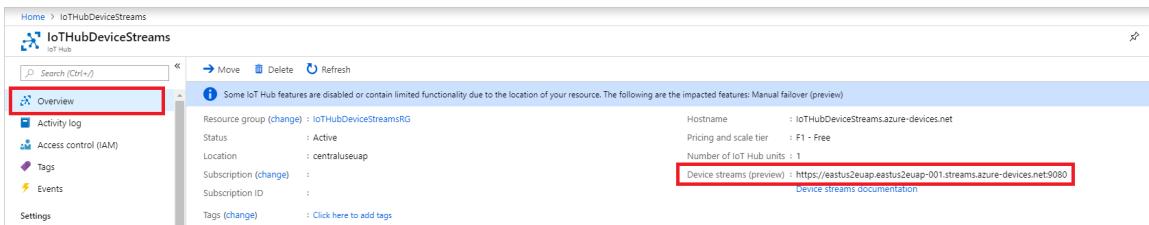
NOTE

Ensure you have installed Azure CLI version 2.0.57 or newer. You can download the latest version from the [Install Azure CLI](#) page.

Allow outbound connectivity to the device streaming endpoints

As mentioned at the beginning of this article, 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.



The screenshot shows the Azure IoT Hub Device Streams Overview page. The 'Overview' tab is selected. On the right, there's a summary of resource details: Resource group (IoTHubDeviceStreamsRG), Status (Active), Location (centraluseuap), Subscription (change), Subscription ID, and Tags (change). Below the summary, there's a note about IoT Hub features being disabled or limited due to location. At the bottom right, a red box highlights the 'Device streams (preview)' link, which is https://eastus2euap.eastus2euap-001.streams.azure-devices.net:9080. There's also a 'Device streams documentation' link.

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 from the [Install Azure CLI](#) page.

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 displays the 'Subscription' (OneDeploy Demo and Test Subscription) and 'Resource group' (IoTHubDeviceStreamsRG). A prominent red box highlights the text 'Turn on diagnostics to collect the following data.'

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.

Home > IoTHubDeviceStreams - Diagnostic settings > Diagnostics settings

Diagnostics settings

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 options like 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 option is highlighted with a red box.

In the main area, there's a search bar and a 'Run' button. Below it, a 'Filter (preview)' section has a 'Schema' dropdown set to 'IoTHubDeviceStreams' and a 'Filter by name or type...' input field containing 'AzureDiagnostics | take 10'. This filter is also highlighted with a red box.

The results table shows log entries from the last 24 hours. The columns include TenantId, SourceSystem, MG, ManagementGroupName, TimeGenerated [UTC], Computer, ResultSignature, CallerIP, Identity_s, DurationMs, ResultType, Resourced, OperationName, Category, Level, properties_s, SubscriptionId, ResourceGroup, and ResourceProvider. One specific log entry is highlighted with a red box:

| | |
|---------------|--------------------------|
| Identity_s | {"deviceId": "MyDevice"} |
| DurationMs | 0 |
| ResultType | success |
| Resourced | /SUBSCRIPTIONS/ |
| OperationName | StreamEstablished |

At the bottom of the table, there's a note: 'RESOURCEGROUPS/IOTHUBDEVICESTREAMSRG/PROVIDERS/MICROSOFT.DEVICES/IOTHUB'. The table has 10 records and a display time of 00:00:00.727.

Regional Availability

During public preview, IoT Hub device streams are available in the Central US, Central US EUAP, North Europe, and Southeast Asia 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

There are two [quickstart samples](#) available on the IoT Hub page. These 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. Note that you can use service and device programs in different languages. For example, you can use the C device program with the C# service program.

Here are the echo samples:

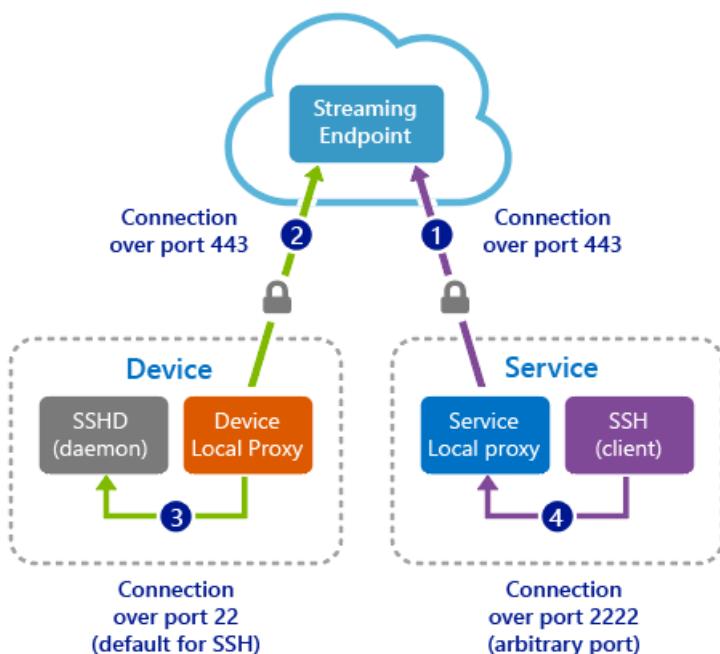
- [C# service and service program](#)
- [Node.js service program](#)
- [C device program](#)

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.

- [C# service and service program](#)
- [Node.js service program](#)
- [C device program](#)

Next steps

Use the links below to learn more about device streams.

[Device streams on IoT show \(Channel 9\)](#)

Azure IoT Hub developer guide

4/21/2020 • 3 minutes to read • [Edit Online](#)

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

7/29/2020 • 2 minutes to read • [Edit Online](#)

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.

| FACTOR | 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 32 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

7/29/2020 • 2 minutes to read • [Edit Online](#)

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 available only in the standard tier of IoT Hub. For more information about the basic and standard/free IoT Hub tiers, see [Choose the right IoT Hub tier](#).

Here is a detailed comparison of the various cloud-to-device communication options.

| CATEGORIES | 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 <code>deviceId</code> , or multiple devices using <code>jobs</code> . | Single device using <code>deviceId</code> , or multiple devices using <code>jobs</code> . | Single device by <code>deviceId</code> . |
| Size | Maximum direct method payload size is 128 KB. | Maximum desired properties size is 32 KB. | Up to 64 KB messages. |

| CATEGORIES | DIRECT METHODS | TWIN'S DESIRED PROPERTIES | CLOUD-TO-DEVICE 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 . |
| 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

7/29/2020 • 2 minutes to read • [Edit Online](#)

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

7/29/2020 • 8 minutes to read • [Edit Online](#)

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. For example, if you are using Event Hubs as a custom endpoint, you must configure the **throughput units** for that event hub so it can handle the ingress of events you plan to send via IoT Hub message routing. Similarly, when using a Service Bus Queue as an endpoint, you must configure the **maximum size** to ensure the queue can hold all the data ingressed, until it is egressed by consumers. 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.

Each message is routed to all endpoints whose routing queries it matches. In other words, a message can be routed to multiple endpoints.

If your custom endpoint has firewall configurations, consider using the Microsoft trusted first party exception, to give your IoT Hub access to the specific endpoint - [Azure Storage](#), [Azure Event Hubs](#) and [Azure Service Bus](#). This is available in select regions for IoT Hubs with [managed service identity](#).

IoT Hub currently supports the following endpoints:

- Built-in endpoint

- Azure Storage
- Service Bus Queues and Service Bus Topics
- Event Hubs

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 Storage

There are two storage services IoT Hub can route messages to -- [Azure Blob Storage](#) and [Azure Data Lake Storage Gen2](#) (ADLS Gen2) accounts. Azure Data Lake Storage accounts are [hierarchical namespace](#)-enabled storage accounts built on top of blob storage. Both of these use blobs for their storage.

IoT Hub supports writing data to Azure Storage in the [Apache Avro](#) format as well as in JSON format. 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 `application/json` and `contentEncoding` to `UTF-8` in the message [system properties](#). Both of these values are case-insensitive. If the content encoding is not set, then IoT Hub will write the messages in base 64 encoded format. 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.



IoT Hub batches messages and writes data to storage 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.

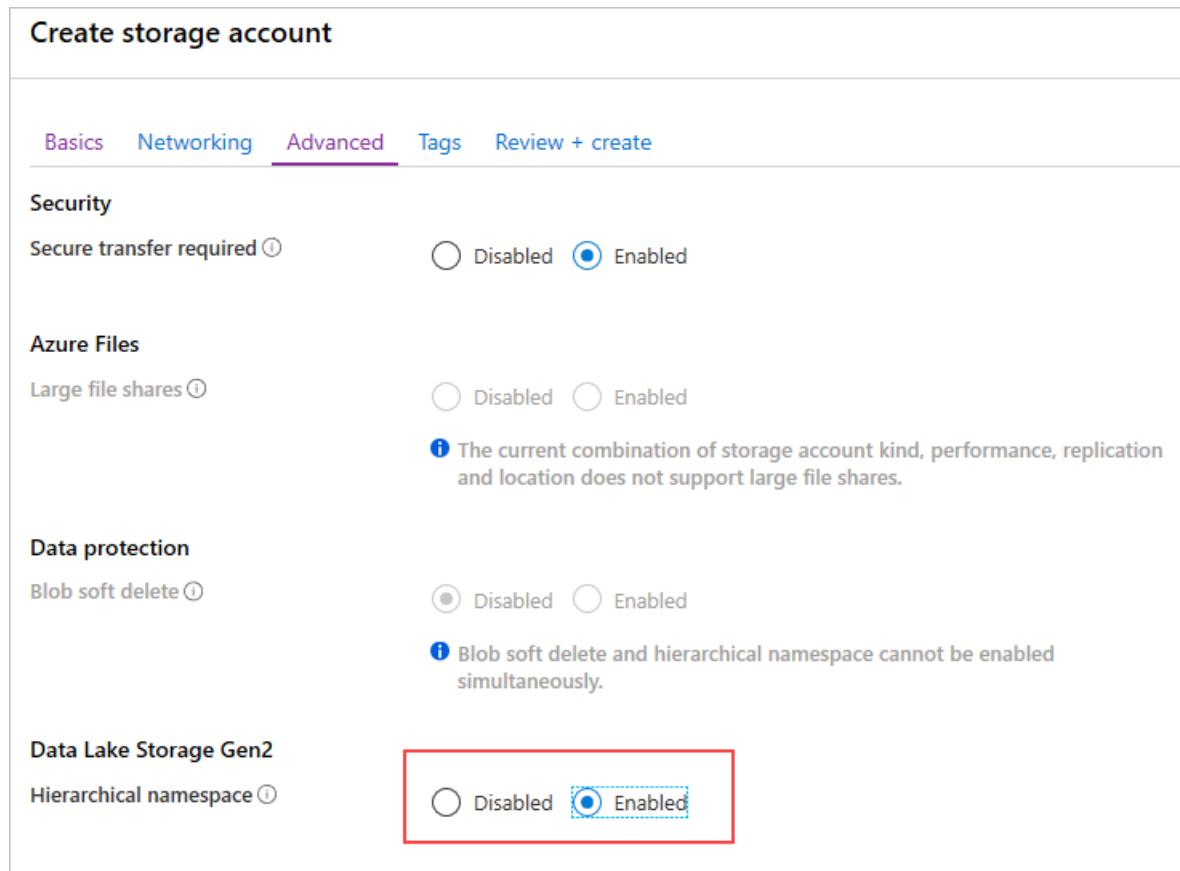
We recommend listing the blobs or files and then iterating over them, to ensure all blobs or files are read without making any assumptions of partition. The partition range could potentially change during a [Microsoft-initiated failover](#) or IoT Hub [manual failover](#). You can use the [List Blobs API](#) to enumerate the list of blobs or [List ADLS Gen2 API](#) for the list of files. 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);
        }
    }
}

```

To create an Azure Data Lake Gen2-compatible storage account, create a new V2 storage account and select *enabled* on the *Hierarchical namespace* field on the **Advanced** tab as shown in the following image:



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, device lifecycle events, and digital twin change events (in public preview). 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 sends a message indicating whether the device was deleted or created. Finally, as part of the [IoT Plug and Play public preview](#), a developer can create routes with data source set to **digital twin change events** and IoT Hub sends messages whenever a digital twin [property](#) is set or changed, a [digital twin](#) is replaced, or when a change event happens for the underlying device twin.

[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](#).

Ordering guarantees with at least once delivery

IoT Hub message routing guarantees ordered and at least once delivery of messages to the endpoints. This means that there can be duplicate messages and a series of messages can be retransmitted honoring the original message ordering. For example, if the original message order is [1,2,3,4], you could receive a message sequence like [1,2,1,2,3,1,2,3,4]. The ordering guarantee is that if you ever receive message [1], it would always be followed by [2,3,4].

For handling message duplicates, we recommend stamping a unique identifier in the application properties

of the message at the point of origin, which is usually a device or a module. The service consuming the messages can handle duplicate messages using this identifier.

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 metrics related to routing and endpoints to give you an overview of the health of your hub and messages sent. [IoT Hub metrics](#) lists all metrics that are enabled by default for your IoT Hub. Using the **routes** diagnostic logs in Azure Monitor [diagnostic settings](#), you can track errors that occur during evaluation of a routing query and endpoint health as perceived by IoT Hub. You can use the REST API [Get Endpoint Health](#) to get [health status](#) of the endpoints.

Use the [troubleshooting guide for routing](#) for more details and support for troubleshooting routing.

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

7/29/2020 • 5 minutes to read • [Edit Online](#)

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.
- You can stamp messages with information that goes into the application properties. For more information, please see [message enrichments](#).

For more information about how to encode and decode messages sent using different protocols, see [Azure IoT SDKs](#).

System Properties of D2C IoT Hub messages

| PROPERTY | DESCRIPTION | USER SETTABLE? | KEYWORD FOR ROUTING QUERY |
|-------------------------------------|---|----------------|------------------------------|
| message-id | <p>A user-settable identifier for the message used for request-reply patterns.</p> <p>Format: A case-sensitive string (up to 128 characters long) of ASCII 7-bit alphanumeric characters +</p> <div style="border: 1px solid black; padding: 2px; display: inline-block;"> {'-', ':', '.', '+', '%', '_', '#', '*', '?', '!', '(', ')', ',', '=', '@', ';', '\$', '''} </div> | Yes | messageId |
| iohub-enqueuedtime | Date and time the Device-to-Cloud message was received by IoT Hub. | No | enqueuedTime |
| user-id | An ID used to specify the origin of messages. When messages are generated by IoT Hub, it is set to <code>{iot hub name}</code> . | Yes | userId |
| iohub-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 | connectionDeviceId |
| iohub-connection-module-id | An ID set by IoT Hub on device-to-cloud messages. It contains the moduleId of the device that sent the message. | No | connectionModuleId |
| iohub-connection-auth-generation-id | An ID set by IoT Hub on device-to-cloud messages. It contains the connectionDeviceGenerationId (as per Device identity properties) of the device that sent the message. | No | connectionDeviceGenerationId |
| iohub-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 | connectionAuthMethod |

| PROPERTY | DESCRIPTION | USER SETTABLE? | KEYWORD FOR ROUTING QUERY |
|---------------|---|----------------|---------------------------|
| dt-dataschema | This value is set by IoT hub on device-to-cloud messages. It contains the device model ID set in the device connection. This feature is available as part of the IoT Plug and Play public preview . | No | N/A |
| dt-subject | The name of the component that is sending the device-to-cloud messages. This feature is available as part of the IoT Plug and Play public preview . | Yes | N/A |

System Properties of C2D IoT Hub messages

| PROPERTY | DESCRIPTION | 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 | |
| to | A destination specified in Cloud-to-Device messages. | No | |
| absolute-expiry-time | Date and time of message expiration. | No | |
| 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> . | Yes | |

| PROPERTY | DESCRIPTION | USER SETTABLE? | |
|-----------|--|----------------|--|
| iohub-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 | |

System Property Names

The system property names vary based on the endpoint to which the messages are being routed. Please see the table below for details on these names.

| SYSTEM PROPERTY NAME | EVENT HUBS | AZURE STORAGE | SERVICE BUS | EVENT GRID |
|-------------------------------|-------------------------------------|------------------------------|-------------------------------------|-------------------------------------|
| Message ID | message-id | messageld | MessageId | message-id |
| User id | user-id | userId | UserId | user-id |
| Connection device id | iohub-connection-device-id | connectionDeviceId | iohub-connection-device-id | iohub-connection-device-id |
| Connection module id | iohub-connection-module-id | connectionModuleId | iohub-connection-module-id | iohub-connection-module-id |
| Connection auth generation id | iohub-connection-auth-generation-id | connectionDeviceGenerationId | iohub-connection-auth-generation-id | iohub-connection-auth-generation-id |
| Connection auth method | iohub-connection-auth-method | connectionAuthMethod | iohub-connection-auth-method | iohub-connection-auth-method |
| contentType | content-type | contentType | ContentType | iohub-content-type |
| contentEncoding | content-encoding | contentEncoding | ContentEncoding | iohub-content-encoding |
| iohub-enqueuedtime | iohub-enqueuedtime | enqueuedTime | N/A | iohub-enqueuedtime |
| CorrelationId | correlation-id | correlationId | CorrelationId | correlation-id |
| dt-dataschema | dt-dataschema | dt-dataschema | dt-dataschema | dt-dataschema |

| SYSTEM PROPERTY NAME | EVENT HUBS | AZURE STORAGE | SERVICE BUS | EVENT GRID |
|----------------------|------------|---------------|-------------|------------|
| dt-subject | dt-subject | dt-subject | dt-subject | dt-subject |

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 values:

- 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

7/29/2020 • 3 minutes to read • [Edit Online](#)

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 time 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. Messages will expire and will not be accessible after the retention time has passed.

IoT Hub also enables you to manage consumer groups on the built-in device-to-cloud receive endpoint. You can have up to 20 consumer groups for each IoT Hub.

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 choose any shared access policy from the drop-down as shown in the screenshot above. It only shows policies that have 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 |
|----------|---|----------------------------|
| .NET | https://www.nuget.org/packages/Azure.Messaging.EventHubs | Quickstart |
| Java | https://mvnrepository.com/artifact/com.azure/azure-messaging-eventhubs | Quickstart |
| Node.js | https://www.npmjs.com/package/@azure/event-hubs | Quickstart |
| Python | https://pypi.org/project/azure-eventhub/ | Quickstart |

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 • [Edit Online](#)

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

7/29/2020 • 6 minutes to read • [Edit Online](#)

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": "2017-05-08T18:55:31.8514657Z"  
    },  
    "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. |
| iothub-connection-device-id | string | This value is set by IoT Hub and identifies the ID of the device. To query, use <code>\$connectionDeviceId</code> . |
| iothub-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> . |
| dt-dataschema | string | This value is set by IoT hub on device-to-cloud messages. It contains the device model ID set in the device connection. This feature is available as part of the IoT Plug and Play public preview . To query, use <code>\$dt-dataschema</code> . |
| dt-subject | string | The name of the component that is sending the device-to-cloud messages. This feature is available as part of the IoT Plug and Play public preview . To query, use <code>\$dt-subject</code> . |

As described in the [IoT Hub Messages](#), there are additional system properties in a message. In addition to above properties in the previous table, you can also query `connectionDeviceId`, `connectionModuleId`.

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 if 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 is shown 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);
});
```

NOTE

This shows how to handle the encoding of the body in javascript. If you want to see a sample in C#, download the [Azure IoT C# Samples](#). Unzip the master.zip file. The Visual Studio solution *SimulatedDevice*'s Program.cs file shows how to encode and submit messages to an IoT Hub. This is the same sample used for testing the message routing, as explained in the [Message Routing tutorial](#). At the bottom of Program.cs, it also has a method to read in one of the encoded files, decode it, and write it back out as ASCII so you can read it.

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. Querying on module twin is also 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 twin needs to be prefixed with the `$twin`. Your query expression can also combine a twin tag or property reference with a body reference, message system properties, and message application properties reference. We recommend using unique names in tags and properties as the query is not case-sensitive. This applies to both device twins and module twins. Also refrain from using `twin`, `$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
```

Routing query on body or device twin with a period in the payload or property name is not supported.

Next steps

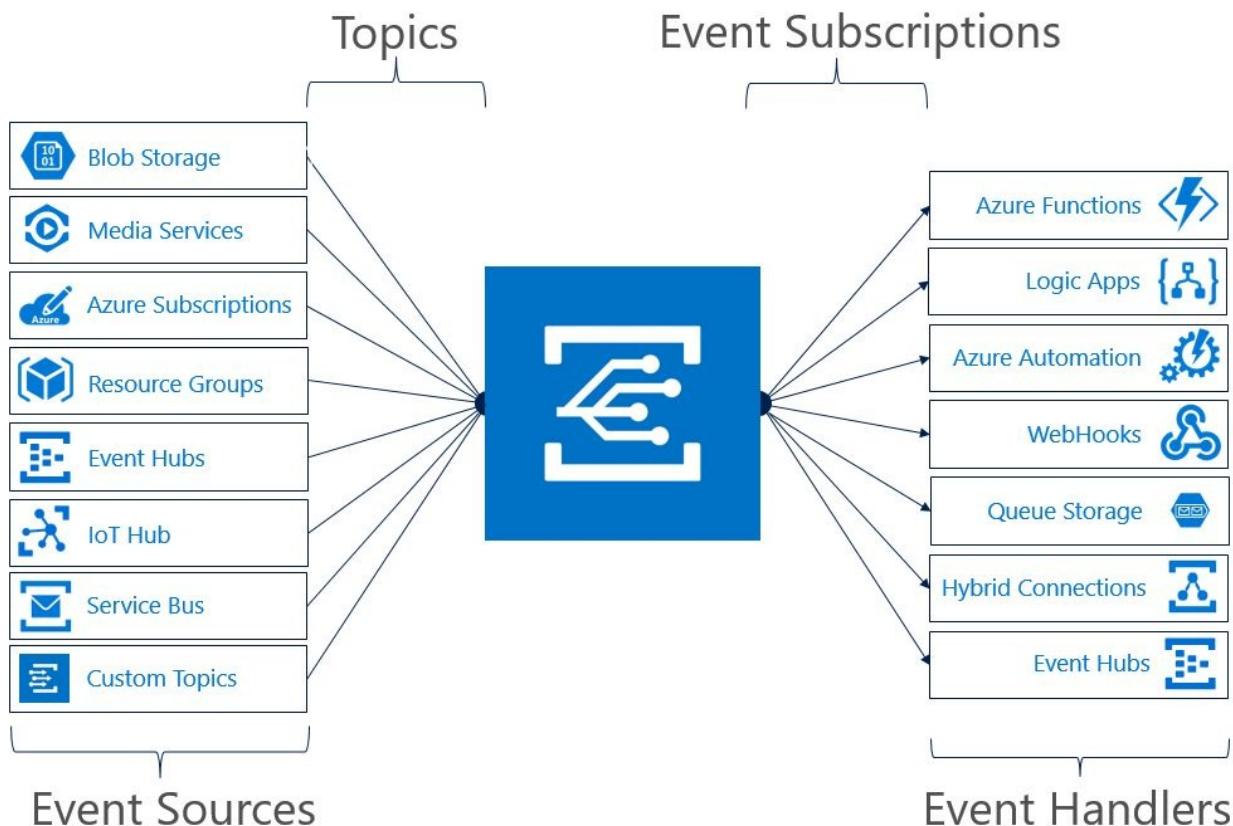
- Learn about [message routing](#).
- Try the [message routing tutorial](#).

React to IoT Hub events by using Event Grid to trigger actions

7/29/2020 • 6 minutes to read • [Edit Online](#)

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. |
| Microsoft.Devices.DeviceTelemetry | Published when a device telemetry message is sent to 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 Telemetry schema

Device telemetry message must be in a valid JSON format with the `contentType` set to **`application/json`** and `contentEncoding` set to **`UTF-8`** in the message [system properties](#). Both of these properties are case insensitive. If the content encoding is not set, then IoT Hub will write the messages in base 64 encoded format.

You can enrich device telemetry events before they are published to Event Grid by selecting the endpoint as Event Grid. For more information, see [Message Enrichments Overview](#).

The following example shows the schema of a device telemetry event:

```
[{
  "id": "9af86784-8d40-fe2g-8b2a-bab65e106785",
  "topic": "/SUBSCRIPTIONS/<subscription ID>/RESOURCEGROUPS/<resource group name>/PROVIDERS/MICROSOFT.DEVICES/IOTHUBS/<hub name>",
  "subject": "devices/LogicAppTestDevice",
  "eventType": "Microsoft.Devices.DeviceTelemetry",
  "eventTime": "2019-01-07T20:58:30.48Z",
  "data": {
    "body": {
      "Weather": {
        "Temperature": 900
      },
      "Location": "USA"
    },
    "properties": {
      "Status": "Active"
    },
    "systemProperties": {
      "iothub-content-type": "application/json",
      "iothub-content-encoding": "utf-8",
      "iothub-connection-device-id": "d1",
      "iothub-connection-auth-method": ""
    }
  },
  "scope": {
    "device": {
      "type": "sas",
      "issuer": "iothub",
      "acceptingIpFilterRule": null,
      "iothub-connection-auth-generation-id": "123455432199234570",
      "iothub-enqueuedtime": "2019-01-07T20:58:30.48Z",
      "iothub-message-source": "Telemetry"
    }
  },
  "dataVersion": "",
  "metadataVersion": "1"
}]
```

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": "0001-01-01T00:00:00",
      "connectionState": "Disconnected",
      "lastActivityTime": "0001-01-01T00:00:00",
      "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, data content and subject, which is the device name.

Event Grid enables [filtering](#) on event types, subjects and data content. While creating the Event Grid subscription, you can choose to subscribe to selected IoT events. 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}
```

Event Grid also allows for filtering on attributes of each event, including the data content. This allows you to

choose what events are delivered based on the contents of the telemetry message. Please see [advanced filtering](#) to view examples. For filtering on the telemetry message body, you must set the `contentType` to `application/json` and `contentEncoding` to `UTF-8` in the message [system properties](#). Both of these properties are case insensitive.

For non-telemetry events like `DeviceConnected`, `DeviceDisconnected`, `DeviceCreated` and `DeviceDeleted`, the Event Grid filtering can be used when creating the subscription. For telemetry events, in addition to the filtering in Event Grid, users can also filter on device twins, message properties and body through the message routing query.

When you subscribe to telemetry events via Event Grid, IoT Hub creates a default message route to send data source type device messages to Event Grid. For more information about message routing, see [IoT Hub message routing](#). This route will be visible in the portal under IoT Hub > Message Routing. Only one route to Event Grid is created regardless of the number of EG subscriptions created for telemetry events. So, if you need several subscriptions with different filters, you can use the OR operator in these queries on the same route. The creation and deletion of the route is controlled through subscription of telemetry events via Event Grid. You cannot create or delete a route to Event Grid using IoT Hub Message Routing.

To filter messages before telemetry data is sent, you can update your [routing query](#). Note that routing query can be applied to the message body only if the body is JSON. You must also set the `contentType` to `application/json` and `contentEncoding` to `UTF-8` in the message [system properties](#).

Limitations for device connected and device disconnected events

To receive device connection state events, a device must do either a 'D2C Send Telemetry' OR a 'C2D Receive Message' operation with IoT Hub. However, note that if a device is using AMQP protocol to connect with IoT Hub, it is recommended that they do a 'C2D Receive Message' operation otherwise their connection state notifications may be delayed by few minutes. 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](#), for IoT Hub C# SDK, or [device client for AMQP](#).

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 current connection state taken at a periodic snapshot, till the flickering continues. Receiving either the same connection state event with different sequence numbers or different connection state events both mean that there was a change in the device connection state.

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 for device created or device deleted events.

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
- Learn how to use IoT telemetry events to implement IoT spatial analytics using Azure Maps

Send cloud-to-device messages from an IoT hub

4/21/2020 • 7 minutes to read • [Edit Online](#)

To send one-way notifications to a device app from your solution back end, send cloud-to-device messages from your IoT hub to your device. For a discussion of other cloud-to-device options supported by Azure IoT Hub, see [Cloud-to-device communications guidance](#).

NOTE

The features described in this article are available only in the standard tier of IoT Hub. For more information about the basic and standard/free IoT Hub tiers, see [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`.

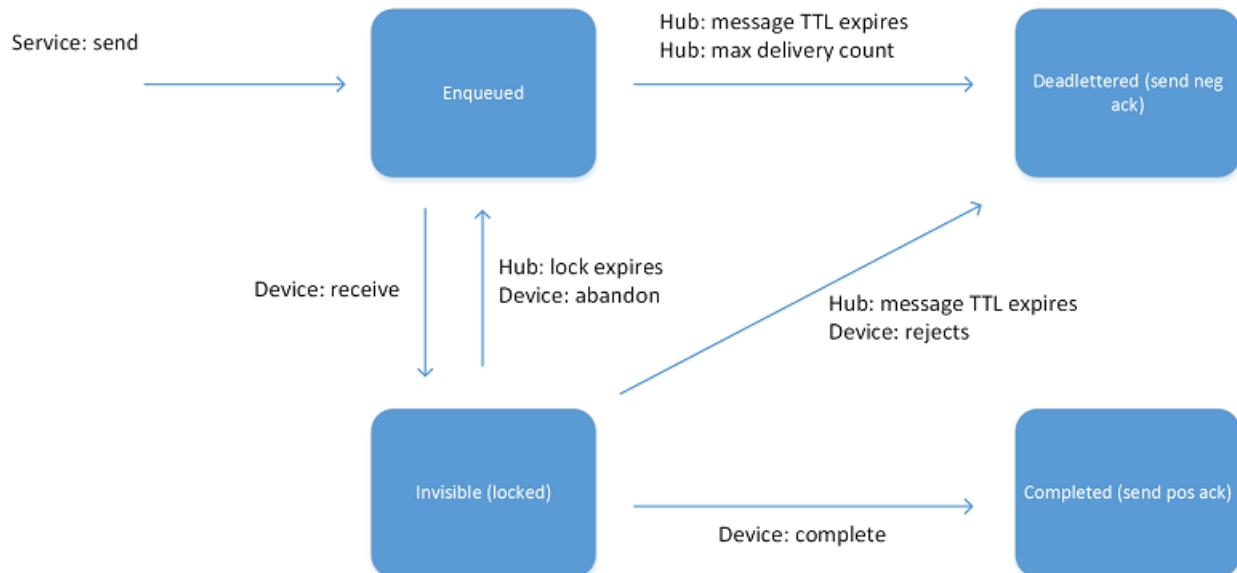
To target each cloud-to-device message at a single device, your IoT hub sets the `to` property to `/devices/{deviceId}/messages/devicebound`.

Each device queue holds, at most, 50 cloud-to-device messages. To try to send more messages to the same device results in an error.

The cloud-to-device message life cycle

To guarantee at-least-once message delivery, your IoT hub persists cloud-to-device messages in per-device queues. For the IoT hub to remove the messages from the queue, the devices must explicitly acknowledge *completion*. This approach guarantees resiliency against connectivity and device failures.

The life-cycle state graph is displayed in the following diagram:



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, the IoT hub *locks* the message by setting the state to *Invisible*. This state allows other threads on the device to start receiving other messages. When a device thread completes the processing of a message, it notifies the IoT hub by *completing* the message. The IoT hub then sets the state to *Completed*.

A device can also:

- *Reject* the message, which causes the IoT hub to set it to the *Dead lettered* state. Devices that connect over the Message Queuing Telemetry Transport (MQTT) Protocol can't reject cloud-to-device messages.
- *Abandon* the message, which causes the IoT hub to put the message back in the queue, with the state set to *Enqueued*. Devices that connect over the MQTT Protocol can't abandon cloud-to-device messages.

A thread could fail to process a message without notifying the IoT hub. In this case, messages automatically transition from the *Invisible* state back to the *Enqueued* state after a *visibility* time-out (or *lock* time-out). The value of this time-out is one minute and cannot be changed.

The **max delivery count** property on the IoT hub determines the maximum number of times a message can transition between the *Enqueued* and *Invisible* states. After that number of transitions, the IoT hub sets the state of the message to *Dead lettered*. Similarly, the IoT hub sets the state of a message to *Dead lettered* after its expiration time. For more information, see [Time to live](#).

The [How to send cloud-to-device messages with IoT Hub](#) article shows you how to send cloud-to-device messages from the cloud and receive them on a device.

A device ordinarily completes a cloud-to-device message when the loss of the message doesn't affect the application logic. An example of this might be when the device has persisted the message content locally or has successfully executed an operation. The message could also carry transient information, whose loss wouldn't impact the functionality of the application. Sometimes, for long-running tasks, you can:

- Complete the cloud-to-device message after the device has persisted 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 either of the following:

- The **ExpiryTimeUtc** property in the service
- The IoT hub, by using the default *time to live* that's specified as an IoT hub property

See [Cloud-to-device configuration options](#).

A common way to take advantage of a message expiration and to 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, but it is more efficient. When you request message acknowledgments, the 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 about the final state of that message. You do this by setting the **iothub-ack** application property in the cloud-to-device message that's being sent to one of the following four values:

| ACK PROPERTY VALUE | BEHAVIOR |
|--------------------|---|
| none | The IoT hub doesn't generate a feedback message (default behavior). |

| ACK PROPERTY | VALUE | BEHAVIOR |
|--------------|-------|--|
| positive | | If the cloud-to-device message reaches the <i>Completed</i> state, the IoT hub generates a feedback message. |
| negative | | If the cloud-to-device message reaches the <i>Dead lettered</i> state, the IoT hub generates a feedback message. |
| full | | The IoT hub generates a feedback message in either case. |

If the **Ack** value is *full*, and you don't receive a feedback message, it means that the feedback message has 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 expiration time is two days, which leaves time to get the service running again if a failure occurs.

As explained in [Endpoints](#), the 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 | A timestamp that indicates 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 | A timestamp that indicates when the outcome of the message happened (for example, the hub received the feedback message or the original message expired) |
| OriginalMessageId | The <i>MessageId</i> of the cloud-to-device message to which this feedback information relates |
| StatusCode | A required string, used in feedback messages that are generated by the IoT hub: <i>Success</i> <i>Expired</i> <i>DeliveryCountExceeded</i> <i>Rejected</i> <i>Purged</i> |
| Description | String values for <i>StatusCodes</i> |
| DeviceId | The <i>DeviceId</i> of the target device of the cloud-to-device message to which this piece of feedback relates |
| DeviceGenerationId | The <i>DeviceGenerationId</i> of the target device of the cloud-to-device message to which this piece of feedback relates |

For the cloud-to-device message to correlate its feedback with the original message, the service must specify a *MessageId*.

The body of a feedback message is shown in the following code:

```
[  
  {  
    "OriginalMessageId": "0987654321",  
    "EnqueuedTimeUtc": "2015-07-28T16:24:48.789Z",  
    "StatusCode": 0,  
    "Description": "Success",  
    "DeviceId": "123",  
    "DeviceGenerationId": "abcdefghijklmnopqrstuvwxyz"  
  },  
  {  
    ...  
  },  
  ...  
]
```

Pending feedback for deleted devices

When a device is deleted, any pending feedback is deleted as well. Device feedback is sent in batches. If a device is deleted in the narrow window (often less than 1 second) between when the device confirms receipt of the message and when the next feedback batch is prepared, the feedback will not occur.

You can address this behavior by waiting a period of time for pending feedback to arrive before deleting your device. Related message feedback should be assumed lost once a device is deleted.

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 2 days (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 2 days (minimum 1 minute); default: 1 hour |
| feedback.maxDeliveryCount | Maximum delivery count for the feedback queue | 1 to 100; default: 10 |
| feedback.lockDurationAsIso8601 | Maximum delivery count for the feedback queue | ISO_8601 interval from 5 to 300 seconds (minimum 5 seconds); default: 60 seconds. |

You can set the configuration options in one of the following ways:

- **Azure portal:** Under **Settings** on your IoT hub, select **Built-in endpoints** and expand **Cloud to device messaging**. (Setting the **feedback.maxDeliveryCount** and **feedback.lockDurationAsIso8601** properties is not currently supported in Azure portal.)

The screenshot shows the Azure IoT Hub 'contoso-sample-hub - Built-in endpoints' settings page. The left sidebar lists various settings like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Events, Settings (Shared access policies, Pricing and scale, IP Filter, Certificates), Built-in endpoints (selected and highlighted with a red box), and Failover. The main pane displays information about built-in system endpoints and the 'Cloud to device messaging' section, which is also highlighted with a red box. It includes controls for message retention time and retry attempts, with specific values for Default TTL (1 hour), Feedback retention time (1 hour), and Maximum delivery count (10 attempts).

- Azure CLI: Use the [az iot hub update](#) command:

```
az iot hub update --name {your IoT hub name} \
    --set properties.cloudToDevice.defaultTtlAsIso8601=PT1H0M0S

az iot hub update --name {your IoT hub name} \
    --set properties.cloudToDevice.maxDeliveryCount=10

az iot hub update --name {your IoT hub name} \
    --set properties.cloudToDevice.feedback.ttlAsIso8601=PT1H0M0S

az iot hub update --name {your IoT hub name} \
    --set properties.cloudToDevice.feedback.maxDeliveryCount=10

az iot hub update --name {your IoT hub name} \
    --set properties.cloudToDevice.feedback.lockDurationAsIso8601=PT0H1M0S
```

Next steps

For information about the SDKs that 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

7/29/2020 • 2 minutes to read • [Edit Online](#)

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.** MQTT and HTTPS support only a single device identity (device ID plus credentials) per TLS connection. For this reason, these protocols are not supported for [field gateway scenarios](#) that require multiplexing messages using multiple device identities across a single or a pool of upstream connections to IoT Hub. Such gateways can use a protocol that supports multiple device identities per connection, like AMQP, for their upstream traffic.
- **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 no more than once every 25 minutes. In development, each device can poll more frequently, if desired.

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

7/29/2020 • 5 minutes to read • [Edit Online](#)

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 life cycle](#). 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 |
|---|--|--|
| <code>enableFileUploadNotifications</code> | Controls whether file upload notifications are written to the file notifications endpoint. | Bool. Default: True. |
| <code>fileNotifications.ttlAsIso8601</code> | Default TTL for file upload notifications. | ISO_8601 interval up to 48H (minimum 1 minute). Default: 1 hour. |
| <code>fileNotifications.lockDuration</code> | Lock duration for the file upload notifications queue. | 5 to 300 seconds (minimum 5 seconds). Default: 60 seconds. |
| <code>fileNotifications.maxDeliveryCount</code> | Maximum delivery count for the file upload notification queue. | 1 to 100. Default: 100. |

You can set these properties on your IoT hub using the Azure portal, Azure CLI, or PowerShell. To learn how, see the topics under [Configure file upload](#).

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

7/29/2020 • 11 minutes to read • [Edit Online](#)

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 identities can also be exported and imported from an IoT Hub via the Service API via either the [REST API](#) or one of the IoT Hub [Service SDKs](#).

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> | <code>deviceLifecycleEvents</code> |
| <code>\$content-encoding</code> | utf-8 |
| <code>opType</code> | <code>createDeviceIdentity</code> or <code>deleteDeviceIdentity</code> |
| <code>hubName</code> | Name of IoT Hub |
| <code>deviceId</code> | ID of the device |
| <code>operationTimestamp</code> | ISO8601 timestamp of operation |

| NAME | VALUE |
|-----------------------|-----------------------------|
| iothub-message-schema | 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": "AAAAAAAAAAE=",
  "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

7/29/2020 • 17 minutes to read • [Edit Online](#)

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 SAS 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 (Deviceld is case-sensitive): `iothubname.azure-devices.net/DeviceId`

Password (You can generate a SAS token with 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: <div style="border: 1px solid black; padding: 2px;">{URL-encoded-resourceURI} + "\n" + expiry</div> <p>Important: The key is decoded from base64 and used as key to perform the HMAC-SHA256 computation.</p> |
| {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, <div style="border: 1px solid black; padding: 2px;">myHub.azure-devices.net/devices/device1</div> |
| {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', Buffer.from(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 parse
from hmac import HMAC

def generate_sas_token(uri, key, policy_name, expiry=3600):
    ttl = time() + expiry
    sign_key = "%s\n%d" % ((parse.quote_plus(uri)), int(ttl))
    print sign_key
    signature = b64encode(HMAC(b64decode(key), sign_key.encode('utf-8'), sha256).digest())

    rawtoken = {
        'sr' : uri,
        'sig': signature,
        'se' : str(int(ttl))
    }

    if policy_name is not None:
        rawtoken['skn'] = policy_name

    return 'SharedAccessSignature ' + parse.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 URL: `{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 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%2fddevices%2fdevice1&sig=13y8ejUk2z7PLmvtwR5RqlGBOVwiq7rQR3WZ5xZX3N4%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%2fddevices&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 verifies that the presented thumbprint matches the configured thumbprint. Authentication using certificate authority validates the certificate chain. Either way, TLS handshake requires the device to have a valid certificate and private key. Refer to the TLS specification for details, for example: [RFC 5246 - The Transport Layer Security \(TLS\) Protocol Version 1.2](#).

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:

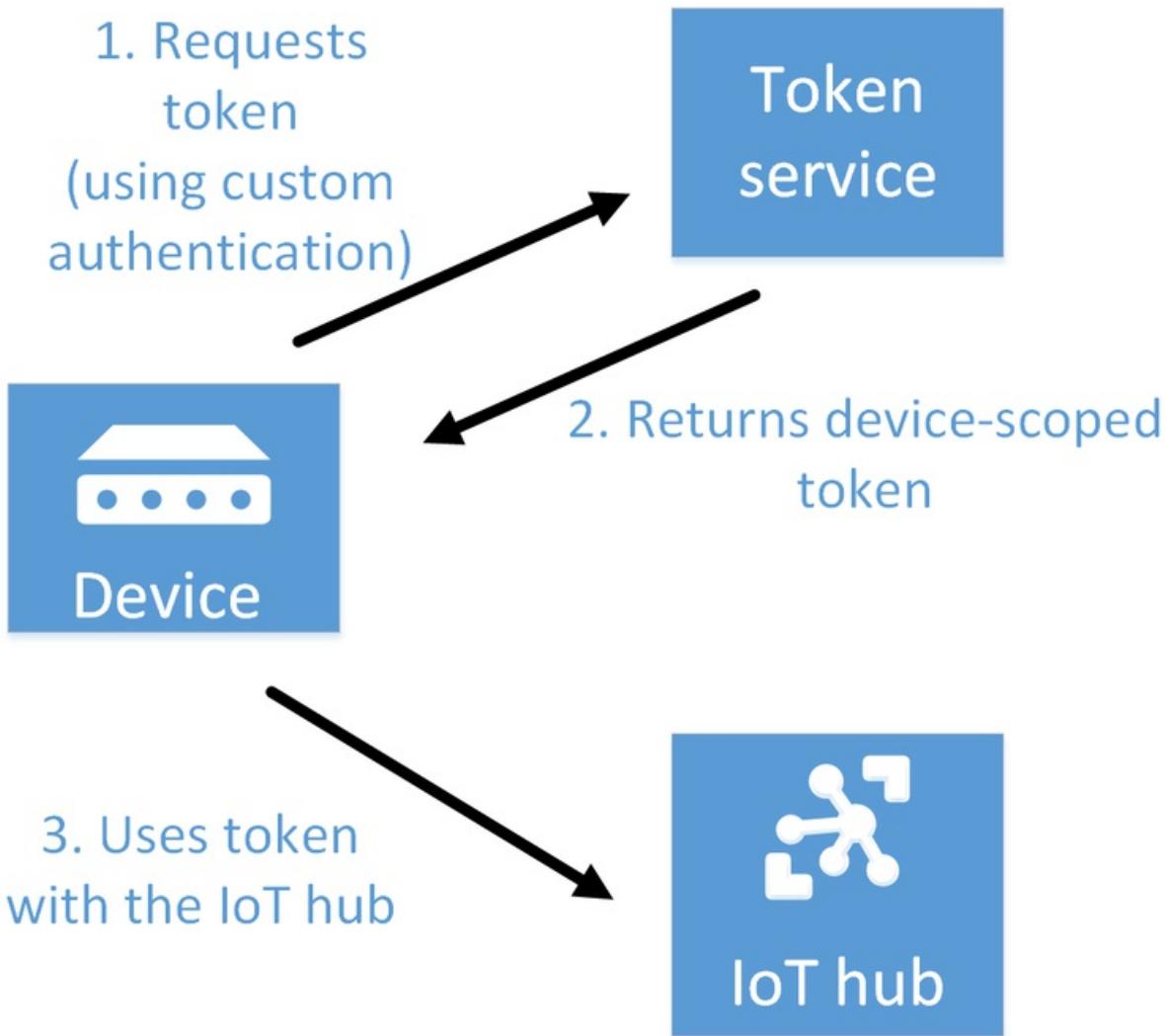
```
var authMethod = new DeviceAuthenticationWithX509Certificate("<device id>", x509Certificate);

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 acknowledgments 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.
- [RFC 5246 - The Transport Layer Security \(TLS\) Protocol Version 1.2](#) provides more information about TLS authentication.

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

7/29/2020 • 12 minutes to read • [Edit Online](#)

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 available only in the standard tier of IoT Hub. For more information about the basic and standard/free IoT Hub tiers, see [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.

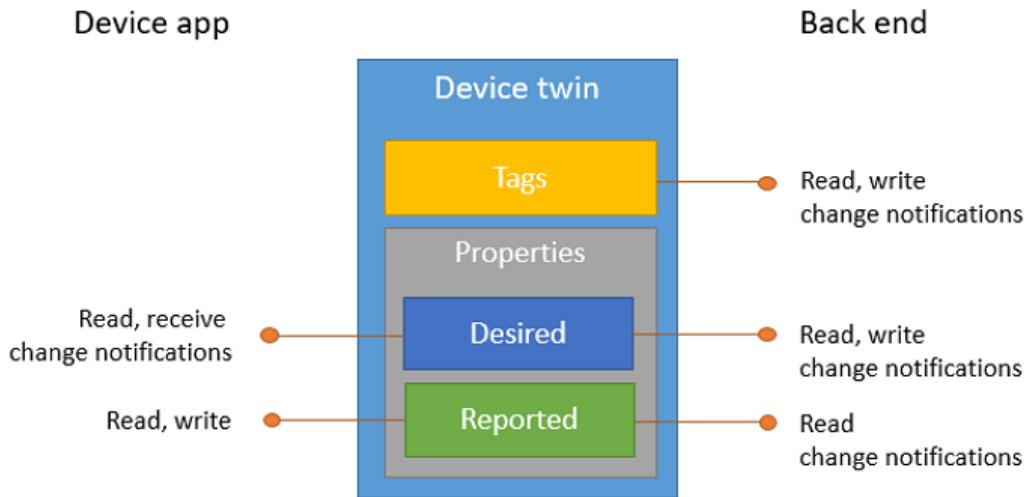
The lifecycle of a device twin is linked to the corresponding [device identity](#). Device twins are implicitly created and deleted when a device identity is created or deleted in IoT Hub.

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](#). Properties `connectionStateUpdatedTime` and `generationId` will not be included.



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 | twinChangeNotification |
| 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 desired and reported system properties) for the currently connected device. (Tags are not visible to device apps.)
- **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:

- **Keys:** All keys in JSON objects are UTF-8 encoded, case-sensitive, and up-to 1 KB in length. Allowed characters exclude UNICODE control characters (segments C0 and C1), and `.`, `$`, and SP.
- **Values:** All values in JSON objects can be of the following JSON types: boolean, number, string, object. Arrays are not allowed.
 - Integers can have a minimum value of -4503599627370496 and a maximum value of

4503599627370495.

- String values are UTF-8 encoded and can have a maximum length of 4 KB.
- **Depth:** The maximum depth of JSON objects in tags, desired properties, and reported properties is 10. For example, the following object is valid:

```
{  
  ...  
  "tags": {  
    "one": {  
      "two": {  
        "three": {  
          "four": {  
            "five": {  
              "six": {  
                "seven": {  
                  "eight": {  
                    "nine": {  
                      "ten": {  
                        "property": "value"  
                      }  
                    }  
                  }  
                }  
              }  
            }  
          }  
        }  
      }  
    }  
  },  
  ...  
}
```

Device twin size

IoT Hub enforces an 8 KB size limit on the value of `tags`, and a 32 KB size limit each on the value of `properties/desired` and `properties/reported`. These totals are exclusive of read-only elements like `$etag`, `$version`, and `$metadata/$lastUpdated`.

Twin size is computed as follows:

- For each property in the JSON document, IoT Hub cumulatively computes and adds the length of the property's key and value.
- Property keys are considered as UTF8-encoded strings.
- Simple property values are considered as UTF8-encoded strings, numeric values (8 Bytes), or Boolean values (4 Bytes).
- The size of UTF8-encoded strings is computed by counting all characters, excluding UNICODE control characters (segments C0 and C1).
- Complex property values (nested objects) are computed based on the aggregate size of the property keys and property values that they contain.

IoT Hub rejects with an error all operations that would increase the size of the `tags`, `properties/desired`, or `properties/reported` 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

7/29/2020 • 10 minutes to read • [Edit Online](#)

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 50 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 available only in the standard tier of IoT Hub. For more information about the basic and standard/free IoT Hub tiers, see [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:

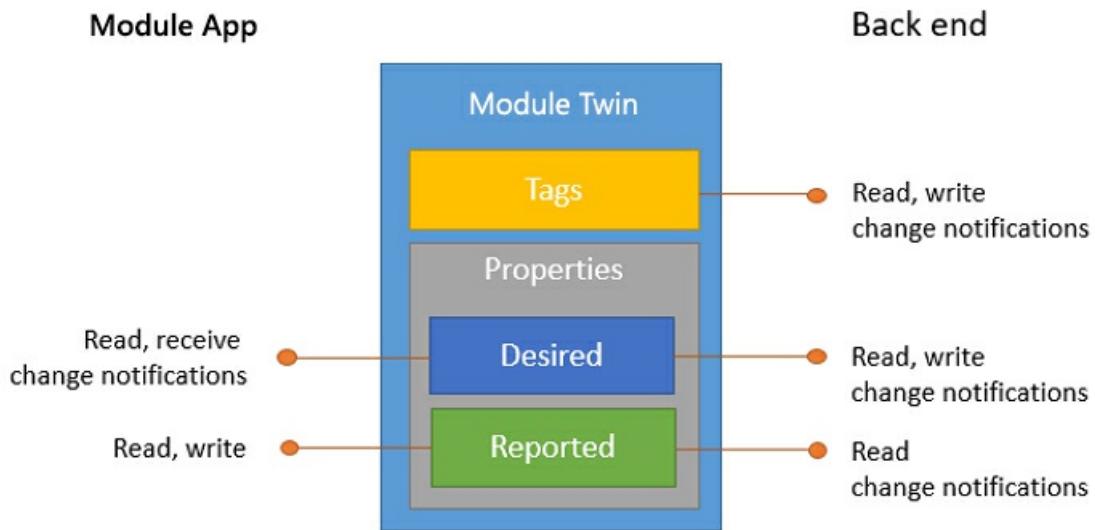
- Modules on the device and IoT Hub can use to synchronize module conditions and configuration.
- The solution back end can use to query and target long-running operations.

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:

```
{
  "tags": [
    "tag1"
  ],
  "properties": {
    "desired": {
      "temp": 20
    },
    "reported": {
      "temp": 20
    }
  }
}
```

```
{
    "deviceId": "devA",
    "moduleId": "moduleA",
    "etag": "AAAAAAAAAc=",
    "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 | twinChangeNotification |
| 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:

- **Keys:** All keys in JSON objects are UTF-8 encoded, case-sensitive, and up-to 1 KB in length. Allowed characters exclude UNICODE control characters (segments C0 and C1), and `.`, `$`, and SP.
- **Values:** All values in JSON objects can be of the following JSON types: boolean, number, string, object. Arrays are not allowed.
 - Integers can have a minimum value of -4503599627370496 and a maximum value of 4503599627370495.
 - String values are UTF-8 encoded and can have a maximum length of 4 KB.

- **Depth:** The maximum depth of JSON objects in tags, desired properties, and reported properties is 10. For example, the following object is valid:

```
{
  ...
  "tags": {
    "one": {
      "two": {
        "three": {
          "four": {
            "five": {
              "six": {
                "seven": {
                  "eight": {
                    "nine": {
                      "ten": {
                        "property": "value"
                      }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  },
  ...
}
```

Module twin size

IoT Hub enforces an 8 KB size limit on the value of `tags`, and a 32 KB size limit each on the value of `properties/desired` and `properties/reported`. These totals are exclusive of read-only elements like `$etag`, `$version`, and `$metadata/$lastUpdated`.

Twin size is computed as follows:

- For each property in the JSON document, IoT Hub cumulatively computes and adds the length of the property's key and value.
- Property keys are considered as UTF8-encoded strings.
- Simple property values are considered as UTF8-encoded strings, numeric values (8 Bytes), or Boolean values (4 Bytes).
- The size of UTF8-encoded strings is computed by counting all characters, excluding UNICODE control characters (segments C0 and C1).
- Complex property values (nested objects) are computed based on the aggregate size of the property keys and property values that they contain.

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

7/29/2020 • 7 minutes to read • [Edit Online](#)

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 available only in the standard tier of IoT Hub. For more information about the basic and standard/free IoT Hub tiers, see [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 between 5 and 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, AMQP, MQTT over WebSockets, or AMQP over WebSockets 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"  
    }  
}
```

The value provided as `responseTimeoutInSeconds` in the request is the amount of time that IoT Hub service must await for completion of a direct method execution on a device. Set this timeout to be at least as long as the expected execution time of a direct method by a device. If timeout is not provided, it the default value of 30 seconds is used. The minimum and maximum values for `responseTimeoutInSeconds` are 5 and 300 seconds, respectively.

The value provided as `connectTimeoutInSeconds` in the request is the amount of time upon invocation of a direct method that IoT Hub service must await for a disconnected device to come online. The default value is 0, meaning that devices must already be online upon invocation of a direct method. The maximum value for `connectTimeoutInSeconds` is 300 seconds.

Example

This example will allow you to securely initiate a request to invoke a Direct Method on an IoT device registered to an Azure IoT Hub.

To begin, use the [Microsoft Azure IoT extension for Azure CLI](#) to create a SharedAccessSignature.

```
az iot hub generate-sas-token -n <iothubName> -du <duration>
```

Next, replace the Authorization header with your newly generated SharedAccessSignature, then modify the `iothubName`, `deviceId`, `methodName` and `payload` parameters to match your implementation in the example `curl` command below.

```
curl -X POST \
https://<iothubName>.azure-devices.net/twins/<deviceId>/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"
    }
}'
```

Execute the modified command to invoke the specified Direct Method. Successful requests will return an HTTP 200 status code.

NOTE

The above example demonstrates invoking a Direct Method on a device. If you wish to invoke a Direct Method in an IoT Edge Module, you would need to modify the url request as shown below:

```
https://<iothubName>.azure-devices.net/twins/<deviceId>/modules/<moduleName>/methods?api-version=2018-06-30
```

Response

The back-end app receives a response that is made up of the following items:

- *HTTP status code*:
 - 200 indicates successful execution of direct method;
 - 404 indicates that either device ID is invalid, or that the device was not online upon invocation of a direct method and for `connectTimeoutInSeconds` thereafter (use accompanied error message to understand the root cause);
 - 504 indicates gateway timeout caused by device not responding to a direct method call within `responseTimeoutInSeconds`.
- *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

4/21/2020 • 4 minutes to read • [Edit Online](#)

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 available only in the standard tier of IoT Hub. For more information about the basic and standard/free IoT Hub tiers, see [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

{
    "jobId": "<jobId>",
    "type": "scheduleDeviceMethod",
    "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.

The following snippet shows the request and response for a job scheduled to call a direct method named testMethod on all devices on contoso-hub-1:

```

PUT https://contoso-hub-1.azure-devices.net/jobs/v2/job01?api-version=2018-06-30 HTTP/1.1
Authorization: SharedAccessSignature sr=contoso-hub-1.azure-devices.net&sig=68iv-----
-----v8Hxalg%3D&se=1556849884&skn=iothubowner
Content-Type: application/json; charset=utf-8
Host: contoso-hub-1.azure-devices.net
Content-Length: 317

{
    "jobId": "job01",
    "type": "scheduleDeviceMethod",
    "cloudToDeviceMethod": {
        "methodName": "testMethod",
        "payload": {},
        "responseTimeoutInSeconds": 30
    },
    "queryCondition": "*",
    "startTime": "2019-05-04T15:53:00.077Z",
    "maxExecutionTimeInSeconds": 20
}

HTTP/1.1 200 OK
Content-Length: 65
Content-Type: application/json; charset=utf-8
Vary: Origin
Server: Microsoft-HTTPAPI/2.0
Date: Fri, 03 May 2019 01:46:18 GMT

{"jobId":"job01","type":"scheduleDeviceMethod","status":"queued"}

```

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

{
    "jobId": "<jobId>",
    "type": "scheduleUpdateTwin",
    "updateTwin": <patch>           // Valid JSON object
    "queryCondition": "<queryOrDevices>", // query condition
    "startTime": <jobStartTime>,        // as an ISO-8601 date string
    "maxExecutionTimeInSeconds": <maxExecutionTimeInSeconds>
}
```

NOTE

The *updateTwin* property requires a valid etag match; for example, `etag="*"`.

The following snippet shows the request and response for a job scheduled to update device twin properties for test-device on contoso-hub-1:

```
PUT https://contoso-hub-1.azure-devices.net/jobs/v2/job02?api-version=2018-06-30 HTTP/1.1
Authorization: SharedAccessSignature sr=contoso-hub-1.azure-devices.net&sig=BN0U-----
-----RuA%3D&se=1556925787&skn=iothubowner
Content-Type: application/json; charset=utf-8
Host: contoso-hub-1.azure-devices.net
Content-Length: 339

{
    "jobId": "job02",
    "type": "scheduleUpdateTwin",
    "updateTwin": {
        "properties": {
            "desired": {
                "test1": "value1"
            }
        },
        "etag": "*"
    },
    "queryCondition": "deviceId = 'test-device'",
    "startTime": "2019-05-08T12:19:56.868Z",
    "maxExecutionTimeInSeconds": 20
}

HTTP/1.1 200 OK
Content-Length: 63
Content-Type: application/json; charset=utf-8
Vary: Origin
Server: Microsoft-HTTPAPI/2.0
Date: Fri, 03 May 2019 22:45:13 GMT

{"jobId":"job02","type":"scheduleUpdateTwin","status":"queued"}
```

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
```

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: scheduleUpdateTwin : A job used to update a set of desired properties or tags. |
| | scheduleDeviceMethod : 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

7/29/2020 • 4 minutes to read • [Edit Online](#)

NOTE

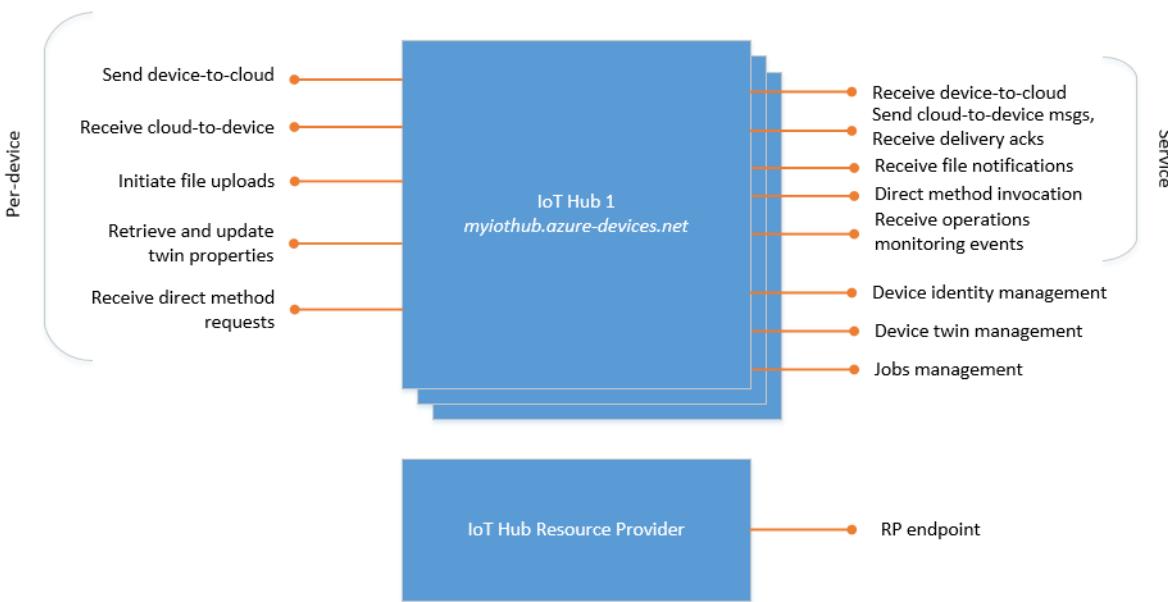
Some of the features mentioned in this article, like cloud-to-device messaging, device twins, and device management, are only available in the standard tier of IoT Hub. For more information about the basic and standard IoT Hub tiers, see [How to choose the right IoT Hub tier](#).

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`.

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. Except where noted, these endpoints are exposed using [MQTT v3.1.1](#), HTTPS 1.1, and [AMQP 1.0](#) protocols. AMQP and MQTT are also available over [WebSockets](#) on port 443.
 - *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](#). HTTPS is not supported.
 - *Receive direct method requests.* A device uses this endpoint to listen for [direct method's requests](#). HTTPS is not supported.
- **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](#) and [AMQP](#) over WebSockets protocols. The direct 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](#).

Endpoint Health

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, as we expect latency to be higher when the endpoint is in one of those states.

| HEALTH STATUS | DESCRIPTION |
|---------------|--|
| healthy | The endpoint is accepting messages as expected. |
| unhealthy | The endpoint is not accepting messages and IoT Hub is retrying to send messages to this endpoint. |
| unknown | IoT Hub has not attempted to deliver messages to this endpoint. |
| degraded | The endpoint is accepting messages slower than expected or is recovering from an unhealthy state. |
| dead | IoT Hub is no longer delivering messages to this endpoint. Retries to send messages to this endpoint failed. |

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](#)
- [Understand your IoT hub IP address](#)

IoT Hub query language for device and module twins, jobs, and message routing

4/3/2020 • 12 minutes to read • [Edit Online](#)

IoT Hub provides a powerful SQL-like language to retrieve information regarding [device twins](#), [module twins](#), [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; instead of from `devices`, you query from `devices.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 [get twin REST API](#). This API always returns the latest values and has higher throttling limits. You can issue the REST API directly or use the equivalent functionality in one of the [Azure IoT Hub Service SDKs](#).

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`, `devices.modules`, 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 three values: `FROM devices` to query device twins, `FROM devices.modules` to query module 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. This is supported only when the value is a primitive type. Primitive types include string, Boolean, numeric, or <code>null</code> . DateTime, object types and arrays are not supported. |
| 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. |

| FUNCTION | DESCRIPTION |
|---------------|---|
| 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

7/29/2020 • 7 minutes to read • [Edit Online](#)

This article explains the quotas for an IoT Hub, and provides information to help you understand how throttling works.

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.

IoT Plug and Play

IoT Plug and Play devices send at least one telemetry message for each interface, including the root, which may increase the number of messages counted towards your message quota.

Operation throttles

Operation throttles are rate limitations that are applied in minute ranges and are intended to prevent abuse. They're also subject to [traffic shaping](#).

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 (5,000/min/unit) |
| New device connections (this limit applies to the rate of <i>new connections</i> , 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 | 6,000 new connections/sec/unit |

| THROTTLE | FREE, B1, AND S1 | B2 AND S2 | B3 AND S3 |
|--|---|---|---|
| Device-to-cloud sends | <p>Higher of 100 send operations/sec or 12 send operations/sec/unit For example, two S1 units are $2 \times 12 = 24$/sec, but you have at least 100 send operations/sec across your units. With nine S1 units, you have 108 send operations/sec (9×12) across your units.</p> | 120 send operations/sec/unit | 6,000 send operations/sec/unit |
| Cloud-to-device sends ¹ | 1.67 send operations/sec/unit (100 messages/min/unit) | 1.67 send operations/sec/unit (100 send operations/min/unit) | 83.33 send operations/sec/unit (5,000 send operations/min/unit) |
| Cloud-to-device receives ¹ (only when device uses HTTPS) | 16.67 receive operations/sec/unit (1,000 receive operations/min/unit) | 16.67 receive operations/sec/unit (1,000 receive operations/min/unit) | 833.33 receive operations/sec/unit (50,000 receive operations/min/unit) |
| File upload | 1.67 file upload initiations/sec/unit (100/min/unit) | 1.67 file upload initiations/sec/unit (100/min/unit) | 83.33 file upload initiations/sec/unit (5,000/min/unit) |
| Direct methods ¹ | 160KB/sec/unit ² | 480KB/sec/unit ² | 24MB/sec/unit ² |
| Queries | 20/min/unit | 20/min/unit | 1,000/min/unit |
| 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 (create, update, list, delete) | 1.67/sec/unit (100/min/unit) | 1.67/sec/unit (100/min/unit) | 83.33/sec/unit (5,000/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 ¹ | 5 new streams/sec | 5 new streams/sec | 5 new streams/sec |
| Maximum number of concurrently connected device streams ¹ | 50 | 50 | 50 |

| THROTTLE | FREE, B1, AND S1 | B2 AND S2 | B3 AND S3 |
|---|------------------|-----------|-----------|
| Maximum device stream data transfer ¹ (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.

Throttling details

- The meter size determines at what increments your throttling limit is consumed. If your direct call's payload is between 0 and 4 KB, it is counted as 4 KB. You can make up to 40 calls per second per unit before hitting the limit of 160 KB/sec/unit.

Similarly, if your payload is between 4 KB and 8 KB, each call accounts for 8 KB and you can make up to 20 calls per second per unit before hitting the max limit.

Finally, if your payload size is between 156KB and 160 KB, you'll be able to make only 1 call per second per unit in your hub before hitting the limit of 160 KB/sec/unit.

- For *Jobs device operations (update twin, invoke direct method)* for tier S2, 50/sec/unit only applies to when you invoke methods using jobs. If you invoke direct methods directly, the original throttling limit of 24 MB/sec/unit (for S2) applies.
- Quota** is the aggregate number of messages you can send in your hub *per day*. You can find your hub's quota limit under the column **Total number of messages /day** on the [IoT Hub pricing page](#).
- Your cloud-to-device and device-to-cloud throttles determine the maximum *rate* at which you can send messages -- number of messages irrespective of 4 KB chunks. Each message can be up to 256 KB which is the [maximum message size](#).
- It's a good practice to throttle your calls so that you don't hit/exceed the throttling limits. If you do hit the limit, IoT Hub responds with error code 429 and the client should back-off and retry. These limits are per hub (or in some cases per hub/unit). For more information, refer to [Manage connectivity and reliable messaging/Retry patterns](#).

Traffic shaping

To accommodate burst traffic, IoT Hub accepts requests above the throttle for a limited time. The first few of these requests are processed immediately. However, if the number of requests continues violate the throttle, IoT Hub starts placing the requests in a queue and processed at the limit rate. This effect is called *traffic shaping*. Furthermore, the size of this queue is limited. If the throttle violation continues, eventually the queue fills up, and IoT Hub starts rejecting requests with [429 ThrottlingException](#).

For example, you use a simulated device to send 200 device-to-cloud messages per second to your S1 IoT Hub (which has a limit of 100/sec D2C sends). For the first minute or two, the messages are processed immediately. However, since the device continues to send more messages than the throttle limit, IoT Hub begins to only process 100 messages per second and puts the rest in a queue. You start noticing increased latency. Eventually, you start getting [429 ThrottlingException](#) as the queue fills up, and the "number of throttle errors" in [IoT Hub's metrics](#) starts increasing.

Identity registry operations throttle

Device 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](#).

Device connections throttle

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 1,000 seconds (approximately 16 minutes). However, you can have as many simultaneously connected devices as you have devices registered in your identity registry.

Other limits

IoT Hub enforces other operational limits:

| OPERATION | LIMIT |
|---|--|
| Devices | The total number of devices plus modules that can be registered to a single IoT hub is capped at 1,000,000. The only way to increase this limit is to contact Microsoft Support . |
| File uploads | 10 concurrent file uploads per device. |
| Jobs ¹ | Maximum concurrent jobs is 1 (for Free and S1), 5 (for S2), and 10 (for S3). However, the max concurrent device import/export jobs is 1 for all tiers. Job history is retained up to 30 days. |
| Additional endpoints | Paid SKU hubs may have 10 additional endpoints. Free SKU hubs may have one additional endpoint. |
| Message routing queries | Paid SKU hubs may have 100 routing queries. Free SKU hubs may have five routing queries. |
| Message enrichments | Paid SKU hubs can have up to 10 message enrichments. Free SKU hubs can have up to 2 message enrichments. |
| 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 per device. |
| Direct method ¹ | Maximum direct method payload size is 128 KB. |
| Automatic device and module configurations ¹ | 100 configurations per paid SKU hub. 20 configurations per free SKU hub. |
| IoT Edge automatic deployments ¹ | 50 modules per deployment. 100 deployments (including layered deployments) per paid SKU hub. 10 deployments per free SKU hub. |
| Twins ¹ | Maximum size of desired properties and reported properties sections are 32 KB each. Maximum size of tags section is 8 KB. |
| Shared access policies | Maximum number of shared access policies is 16 |

¹This feature is not available in the basic tier of IoT Hub. For more information, see [How to choose the right IoT Hub](#).

Increasing the quota or throttle limit

At any given time, you can increase quotas or throttle limits by [increasing the number of provisioned units in an IoT hub](#).

Latency

IoT Hub strives to provide low latency for all operations. However, due to network conditions and other unpredictable factors it cannot guarantee a certain 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

For an in-depth discussion of IoT Hub throttling behavior, see the blog post [IoT Hub throttling and you](#).

Other reference topics in this IoT Hub developer guide include:

- [IoT Hub endpoints](#)

Azure IoT Hub pricing information

4/21/2020 • 3 minutes to read • [Edit Online](#)

[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

7/29/2020 • 4 minutes to read • [Edit Online](#)

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 C SDK resource consumption information](#)

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](#)

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](#)
- [API reference](#)

Azure IoT Hub service SDK for C:

The Azure IoT Service SDK for C is no longer under active development. We will continue to fix critical bugs such as crashes, data corruption, and security vulnerabilities. We will NOT add any new feature or fix bugs that are not critical, however.

Azure IoT Service SDK support is available in higher-level languages ([C#, Java, Node, Python](#)).

- 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.
- [azure-iot-explorer](#): a cross-platform desktop application to connect to your IoT Hub and add/manage/communicate with IoT devices.

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

7/29/2020 • 17 minutes to read • [Edit Online](#)

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 any of the following options.

- Libraries in the [Azure IoT SDKs](#).
- The MQTT protocol directly.

The MQTT port (8883) is blocked in many corporate and educational networking environments. If you can't open port 8883 in your firewall, we recommend using MQTT over Web Sockets. MQTT over Web Sockets communicates over port 443, which is almost always open in networking environments. To learn how to specify the MQTT and MQTT over Web Sockets protocols when using the Azure IoT SDKs, see [Using the device SDKs](#).

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**. You can also specify MQTT over Web Sockets in the client protocol parameter. 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. While it's possible to configure **QoS 0** for faster message exchange, you should note that the delivery isn't guaranteed nor acknowledged. For this reason, **QoS 0** is often referred as "fire and forget".

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 or the MQTT over Web Sockets protocol.

| LANGUAGE | MQTT PROTOCOL PARAMETER | MQTT OVER WEB SOCKETS PROTOCOL PARAMETER |
|----------|----------------------------|--|
| Node.js | azure-iot-device-mqtt.Mqtt | azure-iot-device-mqtt.MqttWs |
| Java | IotHubClientProtocol.MQTT | IotHubClientProtocol.MQTT_WS |
| C | MQTT_Protocol | MQTT_WebSocket_Protocol |
| C# | TransportType.Mqtt | TransportType.Mqtt falls back to MQTT over Web Sockets if MQTT fails. To specify MQTT over Web Sockets only, use TransportType.Mqtt_WebSocket_Only |
| Python | Supports MQTT by default | Add <code>websockets=True</code> in the call to create the client |

The following fragment shows how to specify the MQTT over Web Sockets protocol when using the Azure IoT Node.js SDK:

```
var Client = require('azure-iot-device').Client;
var Protocol = require('azure-iot-device-mqtt').MqttWs;
var client = Client.fromConnectionString(deviceConnectionString, Protocol);
```

The following fragment shows how to specify the MQTT over Web Sockets protocol when using the Azure IoT Python SDK:

```
from azure.iot.device.aio import IoTHubDeviceClient
device_client = IoTHubDeviceClient.create_from_connection_string(deviceConnectionString, websockets=True)
```

Default keep-alive timeout

In order to ensure a client/IoT Hub connection stays alive, both the service and the client regularly send a *keep-alive* ping to each other. The client using IoT SDK sends a keep-alive at the interval defined in this table below:

| LANGUAGE | DEFAULT KEEP-ALIVE INTERVAL | CONFIGURABLE |
|----------|-----------------------------|--------------|
| Node.js | 180 seconds | No |
| Java | 230 seconds | No |
| C | 240 seconds | Yes |
| C# | 300 seconds | Yes |
| Python | 60 seconds | No |

Following [MQTT spec](#), IoT Hub's keep-alive ping interval is 1.5 times the client keep-alive value. However, IoT Hub limits the maximum server-side timeout to 29.45 minutes (1767 seconds) because all Azure services are bound to the Azure load balancer TCP idle timeout, which is 29.45 minutes.

For example, a device using the Java SDK sends the keep-alive ping then loses network connectivity. 230 seconds later, the device misses the keep-alive ping because it's offline. However, IoT Hub doesn't close the connection immediately - it waits another $(230 * 1.5) - 230 = 115$ seconds before disconnecting the device

with the error [404104 DeviceConnectionClosedRemotely](#).

The maximum client keep-alive value you can set is $1767 / 1.5 = 1177$ seconds. Any traffic will reset the keep-alive. For example, a successful SAS token refresh resets the keep-alive.

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](#).
- AMQP is not supported in the Python SDK

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](#) and follow code instructions [below](#).

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 CLI extension command `az iot hub generate-sas-token` 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.

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.

An example of C code using MQTT without Azure IoT C SDK

In this [repository](#), you'll find a couple of C/C++ demo projects showing how to send telemetry messages, receive events with an IoT hub without using the Azure IoT C SDK.

These samples use the Eclipse Mosquitto library to send message to the MQTT Broker implemented in the IoT hub.

This repository contains:

For Windows:

- TelemetryMQTTWin32: contains code to send a telemetry message to an Azure IoT hub, built and run on a Windows machine.
- SubscribeMQTTWin32: contains code to subscribe to events of a given IoT hub on a Windows machine.
- DeviceTwinMQTTWin32: contains code to query and subscribe to the device twin events of a device in the Azure IoT hub on a Windows machine.
- PhPMQTTWin32: contains code to send a telemetry message with IoT Plug & Play preview Device capabilities to an Azure IoT hub, built and run on a Windows machine. More on IoT Plug & Play [here](#)

For Linux:

- MQTTLinux: contains code and build script to run on Linux (WSL, Ubuntu, and Raspbian have been tested so far).
- LinuxConsoleVS2019: contains the same code but in a VS2019 project targeting WSL (Windows Linux sub system). This project allows you to debug the code running on Linux step by step from Visual Studio.

For mosquitto_pub:

This folder contains two samples commands used with mosquitto_pub utility tool provided by Mosquitto.org.

- Mosquitto_sendmessage: to send a simple text message to an Azure IoT hub acting as a device.
- Mosquitto_subscribe: to see events occurring in an Azure IoT hub.

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 file>"
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 + "/?api-version=2018-06-30", 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_2, 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()

```

To authenticate using a device certificate, update the code snippet above with the following changes (see [How to get an X.509 CA certificate](#) on how to prepare for certificate-based authentication):

```

# Create the client as before
# ...

# Set the username but not the password on your client
client.username_pw_set(username=iot_hub_name+".azure-devices.net/" +
                      device_id + "/?api-version=2018-06-30", password=None)

# Set the certificate and key paths on your client
cert_file = "<local path to your certificate file>"
key_file = "<local path to your device key file>"
client.tls_set(ca_certs=path_to_root_cert, certfile=cert_file, keyfile=key_file,
               cert_reqs=ssl.CERT_REQUIRED, tls_version=ssl.PROTOCOL_TLSv1_2, ciphers=None)

# Connect as before
client.connect(iot_hub_name+".azure-devices.net", port=8883)

```

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 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. For additional details about the format of the property bag, see [Sending device-to-cloud messages](#).

In cloud-to-device messages, values in the property bag are represented as in the following table:

| PROPERTY VALUE | REPRESENTATION | DESCRIPTION |
|-------------------|------------------|--|
| <code>null</code> | <code>key</code> | Only the key appears in the property bag |

| PROPERTY VALUE | REPRESENTATION | DESCRIPTION |
|---------------------------|----------------|---|
| empty string | key= | The key followed by an equal sign with no value |
| non-null, non-empty value | key=value | The key followed by an equal sign and the value |

The following example shows a property bag that contains three application properties: **prop1** with a value of `null`; **prop2**, an empty string (""); and **prop3** with a value of "a string".

```
/?prop1&prop2=&prop3=a%20string
```

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 |
|--------|--|
| 200 | Success |
| 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 |
|--------|--|
| 204 | Success (no content is returned) |
| 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 mean 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](#)

Communicate with your IoT hub by using the AMQP Protocol

4/21/2020 • 9 minutes to read • [Edit Online](#)

Azure IoT Hub supports [OASIS Advanced Message Queuing Protocol \(AMQP\) version 1.0](#) to deliver a variety of functionalities through device-facing and service-facing endpoints. This document describes the use of AMQP clients to connect to an IoT hub to use IoT Hub functionality.

Service client

Connect and authenticate to an IoT hub (service client)

To connect to an IoT hub by using AMQP, a client can use the [claims-based security \(CBS\)](#) or [Simple Authentication and Security Layer \(SASL\) authentication](#).

The following information is required for the service client:

| INFORMATION | VALUE |
|-------------------------|--|
| IoT hub hostname | <iot-hub-name>.azure-devices.net |
| Key name | service |
| Access key | A primary or secondary key that's associated with the service |
| Shared access signature | A short-lived shared access signature in the following format: SharedAccessSignature sig={signature-string}&se={expiry}&skn={policyName}&sr={URL-encoded-resourceURI} . To get the code for generating this signature, see Control access to IoT Hub . |

The following code snippet uses the [uAMQP library in Python](#) to connect to an IoT hub via a sender link.

```
import uamqp
import urllib
import time

# Use generate_sas_token implementation available here:
# https://docs.microsoft.com/azure/iot-hub/iot-hub-devguide-security#security-token-structure
from helper import generate_sas_token

iot_hub_name = '<iot-hub-name>'
hostname = '{iot_hub_name}.azure-devices.net'.format(iot_hub_name=iot_hub_name)
policy_name = 'service'
access_key = '<primary-or-secondary-key>'
operation = '<operation-link-name>' # example: '/messages/devicebound'

username = '{policy_name}@sas.root.{iot_hub_name}'.format(
    iot_hub_name=iot_hub_name, policy_name=policy_name)
sas_token = generate_sas_token(hostname, access_key, policy_name)
uri = 'amqps://{}:{}@{}{}'.format(urllib.quote_plus(username),
                                  urllib.quote_plus(sas_token), hostname, operation)

# Create a send or receive client
send_client = uamqp.SendClient(uri, debug=True)
receive_client = uamqp.ReceiveClient(uri, debug=True)
```

Invoke cloud-to-device messages (service client)

To learn about the cloud-to-device message exchange between the service and the IoT hub and between the device and the IoT hub, see [Send cloud-to-device messages from your IoT hub](#). The service client uses two links to send messages and receive feedback for previously sent messages from devices, as described in the following table:

| CREATED BY | LINK TYPE | LINK PATH | DESCRIPTION |
|------------|---------------|---------------------------------|--|
| Service | Sender link | /messages/devicebound | Cloud-to-device messages that are destined for devices are sent to this link by the service. Messages sent over this link have their <code>To</code> property set to the target device's receiver link path, /devices/<deviceID>/messages/devicebound |
| Service | Receiver link | /messages/serviceBound/feedback | Completion, rejection, and abandonment feedback messages that come from devices received on this link by service. For more information about feedback messages, see Send cloud-to-device messages from an IoT hub . |

The following code snippet demonstrates how to create a cloud-to-device message and send it to a device by using the [uAMQP library in Python](#):

```
import uuid
# Create a message and set message property 'To' to the device-bound link on device
msg_id = str(uuid.uuid4())
msg_content = b"Message content goes here!"
device_id = '<device-id>'
to = '/devices/{device_id}/messages/devicebound'.format(device_id=device_id)
ack = 'full' # Alternative values are 'positive', 'negative', and 'none'
app_props = {'iothub-ack': ack}
msg_props = uamqp.message.MessageProperties(message_id=msg_id, to=to)
msg = uamqp.Message(msg_content, properties=msg_props,
                     application_properties=app_props)

# Send the message by using the send client that you created and connected to the IoT hub earlier
send_client.queue_message(msg)
results = send_client.send_all_messages()

# Close the client if it's not needed
send_client.close()
```

To receive feedback, the service client creates a receiver link. The following code snippet demonstrates how to create a link by using the [uAMQP library in Python](#):

```

import json

operation = '/messages/serviceBound/feedback'

# ...
# Re-create the URI by using the preceding feedback path and authenticate it
uri = 'amqps://{}:{}@{}{}'.format(urllib.quote_plus(username),
                                   urllib.quote_plus(sas_token), hostname, operation)

receive_client = uamqp.ReceiveClient(uri, debug=True)
batch = receive_client.receive_message_batch(max_batch_size=10)
for msg in batch:
    print('received a message')
    # Check content_type in message property to identify feedback messages coming from device
    if msg.properties.content_type == 'application/vnd.microsoft.iothub.feedback.json':
        msg_body_raw = msg.get_data()
        msg_body_str = ''.join(msg_body_raw)
        msg_body = json.loads(msg_body_str)
        print(json.dumps(msg_body, indent=2))
        print('******')
        for feedback in msg_body:
            print('feedback received')
            print('\tstatusCode: ' + str(feedback['statusCode']))
            print('\toriginalMessageId: ' + str(feedback['originalMessageId']))
            print('\tdeviceId: ' + str(feedback['deviceId']))
            print
    else:
        print('unknown message:', msg.properties.content_type)

```

As shown in the preceding code, a cloud-to-device feedback message has a content type of *application/vnd.microsoft.iothub.feedback.json*. You can use the properties in the message's JSON body to infer the delivery status of the original message:

- Key `statusCode` in the feedback body has one of the following values: *Success*, *Expired*, *DeliveryCountExceeded*, *Rejected*, or *Purged*.
- Key `deviceId` in the feedback body has the ID of the target device.
- Key `originalMessageId` in the feedback body has the ID of the original cloud-to-device message that was sent by the service. You can use this delivery status to correlate feedback to cloud-to-device messages.

Receive telemetry messages (service client)

By default, your IoT hub stores ingested device telemetry messages in a built-in event hub. Your service client can use the AMQP Protocol to receive the stored events.

For this purpose, the service client first needs to connect to the IoT hub endpoint and receive a redirection address to the built-in event hubs. The service client then uses the provided address to connect to the built-in event hub.

In each step, the client needs to present the following pieces of information:

- Valid service credentials (service shared access signature token).
- A well-formatted path to the consumer group partition that it intends to retrieve messages from. For a given consumer group and partition ID, the path has the following format: `/messages/events/ConsumerGroups/<consumer_group>/Partitions/<partition_id>` (the default consumer group is `$Default`).
- An optional filtering predicate to designate a starting point in the partition. This predicate can be in the form of a sequence number, offset, or enqueued timestamp.

The following code snippet uses the [uAMQP library in Python](#) to demonstrate the preceding steps:

```

import json
import uamqp
import urllib
import time

# Use the generate_sas_token implementation that's available here: https://docs.microsoft.com/azure/iot-hub/iot-hub-devguide-security#security-token-structure
from helper import generate_sas_token

iot_hub_name = '<iot-hub-name>'
hostname = '{iot_hub_name}.azure-devices.net'.format(iot_hub_name=iot_hub_name)
policy_name = 'service'
access_key = '<primary-or-secondary-key>'
operation = '/messages/events/ConsumerGroups/{consumer_group}/Partitions/{p_id}'.format(
    consumer_group='$Default', p_id=0)

username = '{policy_name}@sas.root.{iot_hub_name}'.format(
    policy_name=policy_name, iot_hub_name=iot_hub_name)
sas_token = generate_sas_token(hostname, access_key, policy_name)
uri = 'amqps://{}:{}@{}{}'.format(urllib.quote_plus(username),
                                  urllib.quote_plus(sas_token), hostname, operation)

# Optional filtering predicates can be specified by using endpoint_filter
# Valid predicates include:
# - amqp.annotation.x-opt-sequence-number
# - amqp.annotation.x-opt-offset
# - amqp.annotation.x-opt-enqueued-time
# Set endpoint_filter variable to None if no filter is needed
endpoint_filter = b'amqp.annotation.x-opt-sequence-number > 2995'

# Helper function to set the filtering predicate on the source URI

def set_endpoint_filter(uri, endpoint_filter=''):
    source_uri = uamqp.address.Source(uri)
    source_uri.set_filter(endpoint_filter)
    return source_uri

receive_client = uamqp.ReceiveClient(
    set_endpoint_filter(uri, endpoint_filter), debug=True)
try:
    batch = receive_client.receive_message_batch(max_batch_size=5)
except uamqp.errors.LinkRedirect as redirect:
    # Once a redirect error is received, close the original client and recreate a new one to the re-directed address
    receive_client.close()

    sas_auth = uamqp.authentication.SASTokenAuth.from_shared_access_key(
        redirect.address, policy_name, access_key)
    receive_client = uamqp.ReceiveClient(set_endpoint_filter(
        redirect.address, endpoint_filter), auth=sas_auth, debug=True)

    # Start receiving messages in batches
    batch = receive_client.receive_message_batch(max_batch_size=5)
for msg in batch:
    print('*** received a message ***')
    print(''.join(msg.get_data()))
    print('\t: ' + str(msg.annotations['x-opt-sequence-number']))
    print('\t: ' + str(msg.annotations['x-opt-offset']))
    print('\t: ' + str(msg.annotations['x-opt-enqueued-time']))

```

For a given device ID, the IoT hub uses a hash of the device ID to determine which partition to store its messages in. The preceding code snippet demonstrates how events are received from a single such partition. However, note that a typical application often needs to retrieve events that are stored in all event hub partitions.

Device client

Connect and authenticate to an IoT hub (device client)

To connect to an IoT hub by using AMQP, a device can use [claims based security \(CBS\)](#) or [Simple Authentication and Security](#)

Layer (SASL) authentication.

The following information is required for the device client:

| INFORMATION | VALUE |
|-------------------------|---|
| IoT hub hostname | <iot-hub-name>.azure-devices.net |
| Access key | A primary or secondary key that's associated with the device |
| Shared access signature | A short-lived shared access signature in the following format: <code>SharedAccessSignature sig={signature-string}&se={expiry}&skn={policyName}&sr={URL-encoded-resourceURI}</code> . To get the code for generating this signature, see Control access to IoT Hub . |

The following code snippet uses the [uAMQP library in Python](#) to connect to an IoT hub via a sender link.

```
import uamqp
import urllib
import uuid

# Use generate_sas_token implementation available here:
# https://docs.microsoft.com/azure/iot-hub/iot-hub-devguide-security#security-token-structure
from helper import generate_sas_token

iot_hub_name = '<iot-hub-name>'
hostname = '{iot_hub_name}.azure-devices.net'.format(iot_hub_name=iot_hub_name)
device_id = '<device-id>'
access_key = '<primary-or-secondary-key>'
username = '{device_id}@sas.{iot_hub_name}'.format(
    device_id=device_id, iot_hub_name=iot_hub_name)
sas_token = generate_sas_token('{hostname}/devices/{device_id}'.format(
    hostname=hostname, device_id=device_id), access_key, None)

# e.g., '/devices/{device_id}/messages/devicebound'
operation = '<operation-link-name>'
uri = 'amqps://{}:{}@{}{}'.format(urllib.quote_plus(username),
                                   urllib.quote_plus(sas_token), hostname, operation)

receive_client = uamqp.ReceiveClient(uri, debug=True)
send_client = uamqp.SendClient(uri, debug=True)
```

The following link paths are supported as device operations:

| CREATED BY | LINK TYPE | LINK PATH | DESCRIPTION |
|------------|---------------|--|--|
| Devices | Receiver link | /devices/<deviceID>/messages/devicebound | Cloud-to-device messages that are destined for devices are received on this link by each destination device. |
| Devices | Sender link | /devices/<deviceID>/messages/events | Device-to-cloud messages that are sent from a device are sent over this link. |
| Devices | Sender link | /messages/serviceBound/feedback | Cloud-to-device message feedback sent to the service over this link by devices. |

Receive cloud-to-device commands (device client)

Cloud-to-device commands that are sent to devices arrive on a /devices/<deviceID>/messages/devicebound link. Devices can receive these messages in batches, and use the message data payload, message properties, annotations, or application properties in the message as needed.

The following code snippet uses the [uAMQP library in Python](#) to receive cloud-to-device messages by a device.

```
# ...
# Create a receive client for the cloud-to-device receive link on the device
operation = '/devices/{device_id}/messages/devicebound'.format(
    device_id=device_id)
uri = 'amqps://{}:{}@{}{}'.format(urllib.quote_plus(username),
                                    urllib.quote_plus(sas_token), hostname, operation)

receive_client = uamqp.ReceiveClient(uri, debug=True)
while True:
    batch = receive_client.receive_message_batch(max_batch_size=5)
    for msg in batch:
        print('*** received a message ***')
        print(''.join(msg.get_data()))

        # Property 'to' is set to: '/devices/device1/messages/devicebound',
        print('\tto:          ' + str(msg.properties.to))

        # Property 'message_id' is set to value provided by the service
        print('\tmessage_id:    ' + str(msg.properties.message_id))

        # Other properties are present if they were provided by the service
        print('\tcreation_time: ' + str(msg.properties.creation_time))
        print('\tcorrelation_id: ' +
              str(msg.properties.correlation_id))
        print('\tcontent_type:   ' + str(msg.properties.content_type))
        print('\treply_to_group_id: ' +
              str(msg.properties.reply_to_group_id))
        print('\tsubject:         ' + str(msg.properties.subject))
        print('\tuser_id:         ' + str(msg.properties.user_id))
        print('\tgroup_sequence:  ' +
              str(msg.properties.group_sequence))
        print('\tcontent_encoding: ' +
              str(msg.properties.content_encoding))
        print('\treply_to:         ' + str(msg.properties.reply_to))
        print('\tabolute_expiry_time: ' +
              str(msg.properties.absolute_expiry_time))
        print('\tgroup_id:         ' + str(msg.properties.group_id))

        # Message sequence number in the built-in event hub
        print('\tx-opt-sequence-number: ' +
              str(msg.annotations['x-opt-sequence-number']))
```

Send telemetry messages (device client)

You can also send telemetry messages from a device by using AMQP. The device can optionally provide a dictionary of application properties, or various message properties, such as message ID.

The following code snippet uses the [uAMQP library in Python](#) to send device-to-cloud messages from a device.

```

# ...
# Create a send client for the device-to-cloud send link on the device
operation = '/devices/{device_id}/messages/events'.format(device_id=device_id)
uri = 'amqps://{}:{}@{}{}'.format(urllib.quote_plus(username), urllib.quote_plus(sas_token), hostname, operation)

send_client = uamqp.SendClient(uri, debug=True)

# Set any of the applicable message properties
msg_props = uamqp.message.MessageProperties()
msg_props.message_id = str(uuid.uuid4())
msg_props.creation_time = None
msg_props.correlation_id = None
msg_props.content_type = None
msg_props.reply_to_group_id = None
msg_props.subject = None
msg_props.user_id = None
msg_props.group_sequence = None
msg_props.to = None
msg_props.content_encoding = None
msg_props.reply_to = None
msg_props.absolute_expiry_time = None
msg_props.group_id = None

# Application properties in the message (if any)
application_properties = { "app_property_key": "app_property_value" }

# Create message
msg_data = b"Your message payload goes here"
message = uamqp.Message(msg_data, properties=msg_props, application_properties=application_properties)

send_client.queue_message(message)
results = send_client.send_all_messages()

for result in results:
    if result == uamqp.constants.MessageState.SendFailed:
        print result

```

Additional notes

- The AMQP connections might be disrupted because of a network glitch or the expiration of the authentication token (generated in the code). The service client must handle these circumstances and reestablish the connection and links, if needed. If an authentication token expires, the client can avoid a connection drop by proactively renewing the token prior to its expiration.
- Your client must occasionally be able to handle link redirections correctly. To understand such an operation, see your AMQP client documentation.

Next steps

To learn more about the AMQP Protocol, see the [AMQP v1.0 specification](#).

To learn more about IoT Hub messaging, see:

- [Cloud-to-device messages](#)
- [Support for additional protocols](#)
- [Support for the Message Queuing Telemetry Transport \(MQTT\) Protocol](#)

Glossary of IoT Hub terms

7/29/2020 • 20 minutes to read • [Edit Online](#)

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 Explorer

The [Azure IoT Explorer](#) is used to view the telemetry the device is sending, work with device properties, and call commands. You can also use the explorer to interact with and test your devices, and to manage plug and play devices.

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 has been replaced with the [Azure IoT Explorer](#), which is used to view the telemetry the device is sending, work with device properties, and call commands. You can also use the explorer to interact with and test your devices, and to manage plug and play 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' 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 [Job](#) 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 URLs. 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 a [Job](#) 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

7/29/2020 • 4 minutes to read • [Edit Online](#)

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.

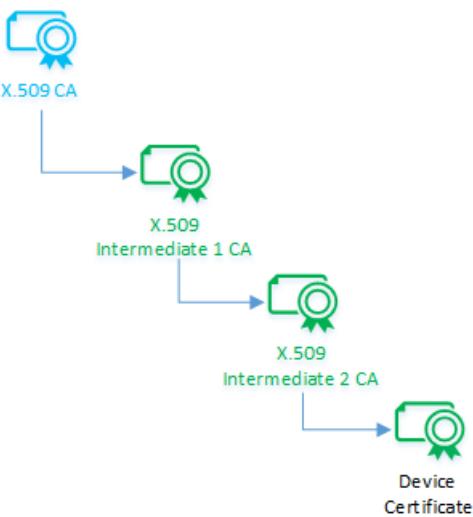
You may also create a self-signed X.509 CA for experimentation or for use in closed IoT networks.

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.



The device certificate (also called a leaf certificate) must have the *Subject Name* set to the **Device ID** (`CN=deviceId`) that was used when registering the IoT device in the Azure IoT Hub. This setting is required for authentication.

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](#).

Create an X.509 device for your 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

4/15/2019 • 11 minutes to read • [Edit Online](#)

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.

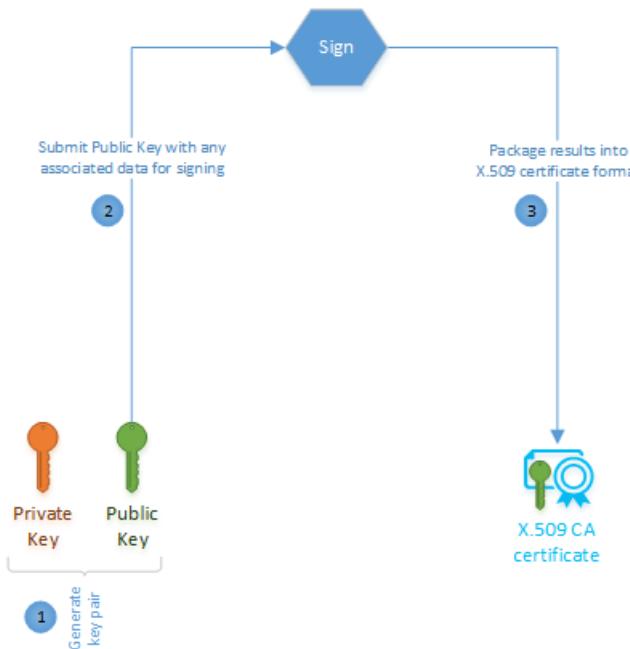
The use of X.509 CA is best understood in relation to a concrete example. Consider Company-X, a maker of Smart-X-Widgets designed for professional installation. Company-X outsources both manufacturing and installation. It contracts manufacturer Factory-Y to manufacture the Smart-X-Widgets, and service 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

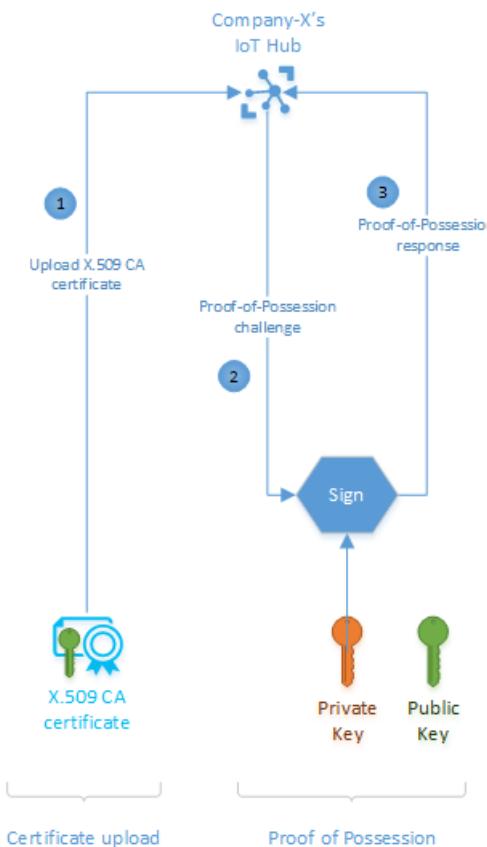
The process to create a Self-Signed X.509 CA certificate is similar to purchasing with the exception of involving a third party signer like the root certificate authority. In our example, Company-X will sign its authority certificate instead of a root certificate authority. Company-X may choose this 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.

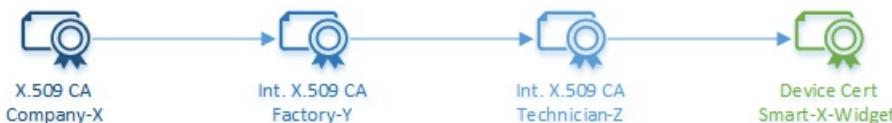
Sign Devices into a Certificate Chain of Trust

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.

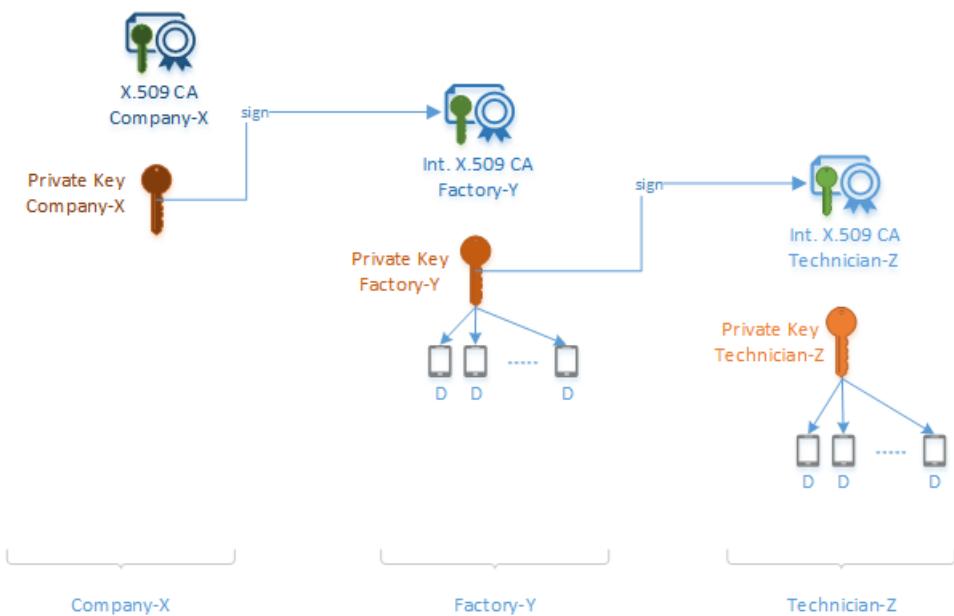
X.509 CA certificate authentication offers elegant solutions to afore listed challenges through the use of certificate chains. A certificate chain results from a CA signing an intermediate CA that in turn signs another intermediate CA and so goes on until a final intermediate CA signs a device. In our 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.

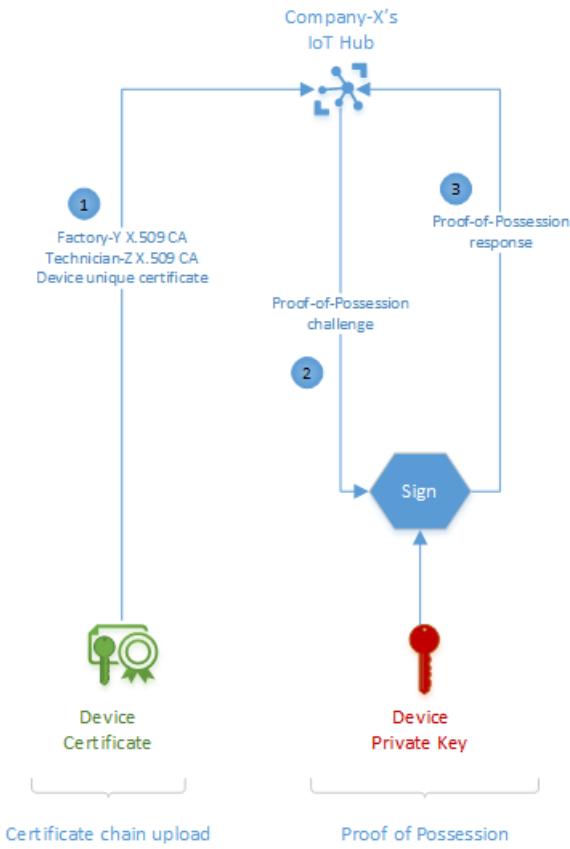
Device Connection

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

7/29/2020 • 4 minutes to read • [Edit Online](#)

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 C SDK resource consumption information](#)

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](#)

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](#)
- [API reference](#)

Azure IoT Hub service SDK for C:

The Azure IoT Service SDK for C is no longer under active development. We will continue to fix critical bugs such as crashes, data corruption, and security vulnerabilities. We will NOT add any new feature or fix bugs that are not critical, however.

Azure IoT Service SDK support is available in higher-level languages ([C#](#), [Java](#), [Node](#), [Python](#)).

- 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.
- [azure-iot-explorer](#): a cross-platform desktop application to connect to your IoT Hub and add/manage/communicate with IoT devices.

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 Device SDKs Platform Support

1/16/2020 • 5 minutes to read • [Edit Online](#)

Microsoft strives to continually expand the universe of Azure IoT Hub capable devices. Microsoft publishes open-source device SDKs on GitHub to help connect devices to Azure IoT Hub and the Device Provisioning Service. The device SDKs are available for C, .NET (C#), Java, Node.js, and Python. Microsoft tests each SDK to ensure that it runs on the supported configurations detailed for it in the [Microsoft SDKs and device platform support](#) section.

In addition to the device SDKs, Microsoft provides several other avenues to empower customers and developers to connect their devices to Azure IoT:

- Microsoft collaborates with several partner companies to help them publish development kits, based on the Azure IoT C SDK, for their hardware platforms.
- Microsoft works with Microsoft trusted partners to provide an ever-expanding set of devices that have been tested and certified for Azure IoT. For a current list of these devices, see the [Azure certified for IoT device catalog](#).
- Microsoft provides a platform abstraction layer (PAL) in the Azure IoT Hub Device C SDK that helps developers to easily port the SDK to their platform. To learn more, see the [C SDK porting guidance](#).

This topic provides information about the Microsoft SDKs and the platform configurations they support, as well as each of the other options listed above.

Microsoft SDKs and device platform support

Microsoft publishes open-source SDKs on GitHub for the following languages: C, .NET (C#), Node.js, Java, and Python. The SDKs and their dependencies are listed in this section. The SDKs are supported on any device platform that satisfies these dependencies.

For each of the listed SDKs, Microsoft:

- Continuously builds and runs end-to-end tests against the master branch of the relevant SDK in GitHub on several popular platforms. To provide test coverage across different compiler versions, we generally test against the latest LTS version and the most popular version.
- Provides installation guidance or installation packages if applicable.
- Fully supports the SDKs on GitHub with open-source code, a path for customer contributions, and product team engagement with GitHub issues.

C SDK

The [Azure IoT Hub C device SDK](#) is tested with and supports the following configurations.

| OS | TLS LIBRARY | ADDITIONAL REQUIREMENTS |
|-------------------|------------------------------|---|
| Linux | OpenSSL, WolfSSL, or BearSSL | Berkeley sockets Portable Operating System Interface (POSIX) |
| iOS 12.2 | OpenSSL | XCode emulated in OSX 10.13.4 |
| Windows 10 family | SChannel | |

| OS | TLS LIBRARY | ADDITIONAL REQUIREMENTS |
|-----------------|-------------|-------------------------------------|
| Mbed OS 5.4 | Mbed TLS 2 | MXChip IoT dev kit |
| Azure Sphere OS | WolfSSL | Azure Sphere MT3620 |
| Arduino | BearSSL | ESP32 or ESP8266 |

Python SDK

The [Azure IoT Hub Python device SDK](#) is tested with and supports the following configurations.

| OS | COMPILER |
|-------------------|----------------------------|
| Linux | Python 2.7.*, 3.5 or later |
| MacOS High Sierra | Python 2.7.*, 3.5 or later |
| Windows 10 family | Python 2.7.*, 3.5 or later |

Only Python version 3.5.3 or later support the asynchronous APIs, we recommend using version 3.7 or later.

.NET SDK

The [Azure IoT Hub .NET \(C#\) device SDK](#) is tested with and supports the following configurations.

| OS | STANDARD |
|------------------------------------|--|
| Linux | .NET Core 2.1 |
| Windows 10 Desktop and Server SKUs | .NET Core 2.1, .NET Framework 4.5.1, or .NET Framework 4.7 |

The .NET SDK can also be used with Windows IoT Core with the [Azure Device Agent](#) or a custom NTService that can use RPC to communicate with UWP applications.

Node.js SDK

The [Azure IoT Hub Node.js device SDK](#) is tested with and supports the following configurations.

| OS | NODE VERSION |
|-------------------|-----------------|
| Linux | LTS and Current |
| Windows 10 family | LTS and Current |

Java SDK

The [Azure IoT Hub Java device SDK](#) is tested with and supports the following configurations.

| OS | JAVA VERSION |
|-----------------------|--------------|
| Android API 28 | Java 8 |
| Linux x64 | Java 8 |
| Windows 10 family x64 | Java 8 |

Partner supported development kits

Microsoft works with various partners to provide development kits for several microprocessor architectures. These partners have ported the Azure IoT C SDK to their platform. Partners create and maintain the platform abstraction layer (PAL) of the SDK. Microsoft works with these partners to provide extended support.

| 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-AZURE P-NUCLEO-AZURE FP-CLD-AZURE | Support |
| Texas Instruments | CC3220SF LaunchPad CC3220S LaunchPad CC3235SF LaunchPad CC3235S LaunchPad MSP432E4 LaunchPad | Azure IoT Plugin for SimpleLink | TI E2E Forum TI E2E Forum for CC3220 TI E2E Forum for MSP432E4 |

Porting the Microsoft Azure IoT C SDK

If your device platform isn't covered by one of the previous sections, you can consider porting the Azure IoT C SDK. Porting the C SDK primarily involves implementing the platform abstraction layer (PAL) of the SDK. The PAL defines primitives that provide the glue between your device and higher-level functions in the SDK. For more information, see [Porting Guidance](#).

Microsoft partners and certified Azure IoT devices

Microsoft works with a number of partners to continually expand the Azure IoT universe with Azure IoT tested and certified devices.

- To browse Azure IoT certified devices, see [Microsoft Azure Certified for IoT Device Catalog](#).
- To learn more about the Azure Certified for IoT ecosystem, see [Join the Certified for IoT ecosystem](#).

Connecting to IoT Hub without an SDK

If you're not able to use one of the IoT Hub device SDKs, you can connect directly to IoT Hub using the [IoT Hub REST APIs](#) from any application capable of sending and receiving HTTPS requests and responses.

Support and other resources

If you experience problems while using the Azure IoT device SDKs, there are several ways to seek support. You can try one of the following channels:

Reporting bugs – Bugs in the device SDKs can be reported on the issues page of the relevant GitHub project. Fixes rapidly make their way from the project in to product updates.

- [Azure IoT Hub C SDK issues](#)

- [Azure IoT Hub .NET \(C#\) SDK issues](#)
- [Azure IoT Hub Java SDK issues](#)
- [Azure IoT Hub Node.js SDK issues](#)
- [Azure IoT Hub Python SDK issues](#)

Microsoft Customer Support team – Users who have a [support plan](#) can engage the Microsoft Customer Support team by creating a new support request directly from the [Azure portal](#).

Feature requests – Azure IoT feature requests are tracked via the product's [User Voice page](#).

Next steps

- [Device and service SDKs](#)
- [Porting Guidance](#)

Azure IoT device SDK for C

7/29/2020 • 17 minutes to read • [Edit Online](#)

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 | Date |
|-----------------------|--|-----------------------------------|
| 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.

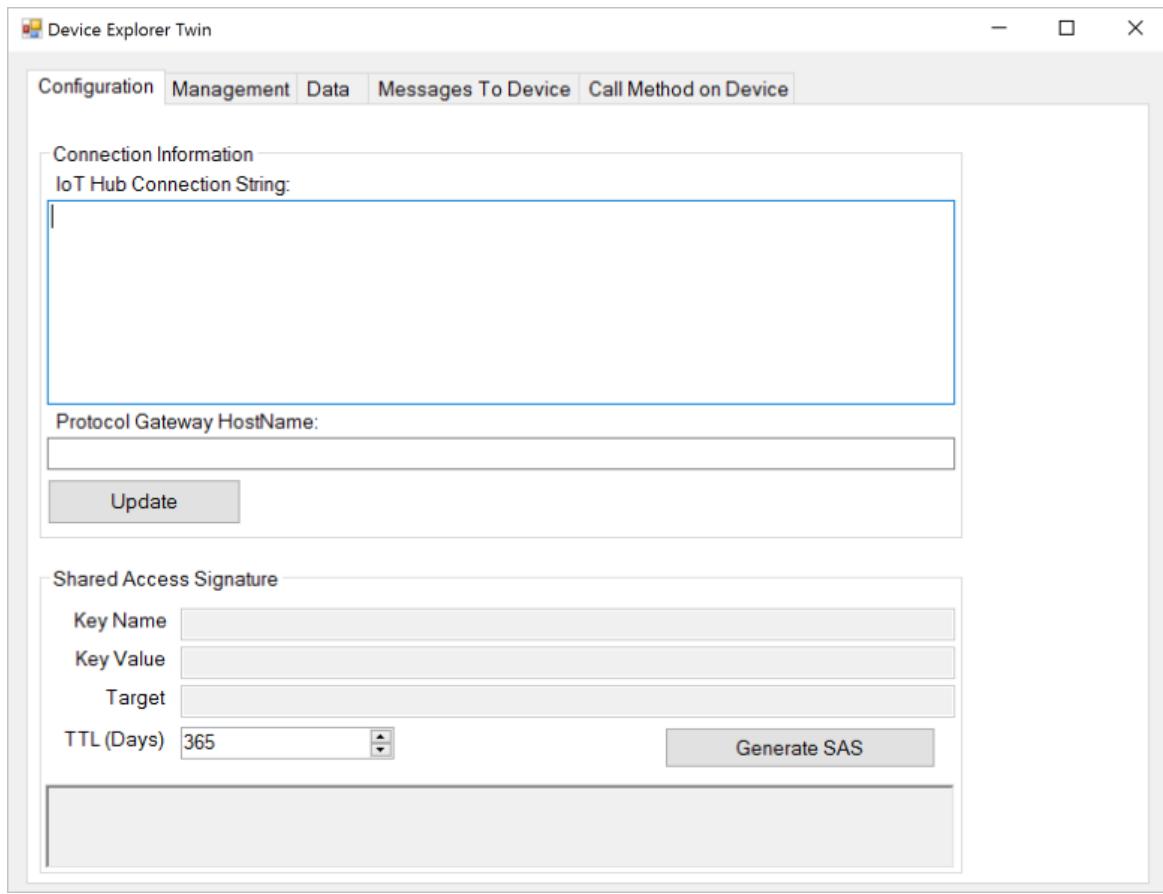
- A Windows application called [Azure IoT Explorer](#).
- A cross-platform Visual Studio Code extension called [Azure IoT Tools](#).
- A cross-platform Python CLI called [the IoT extension for Azure CLI](#).

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 *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.

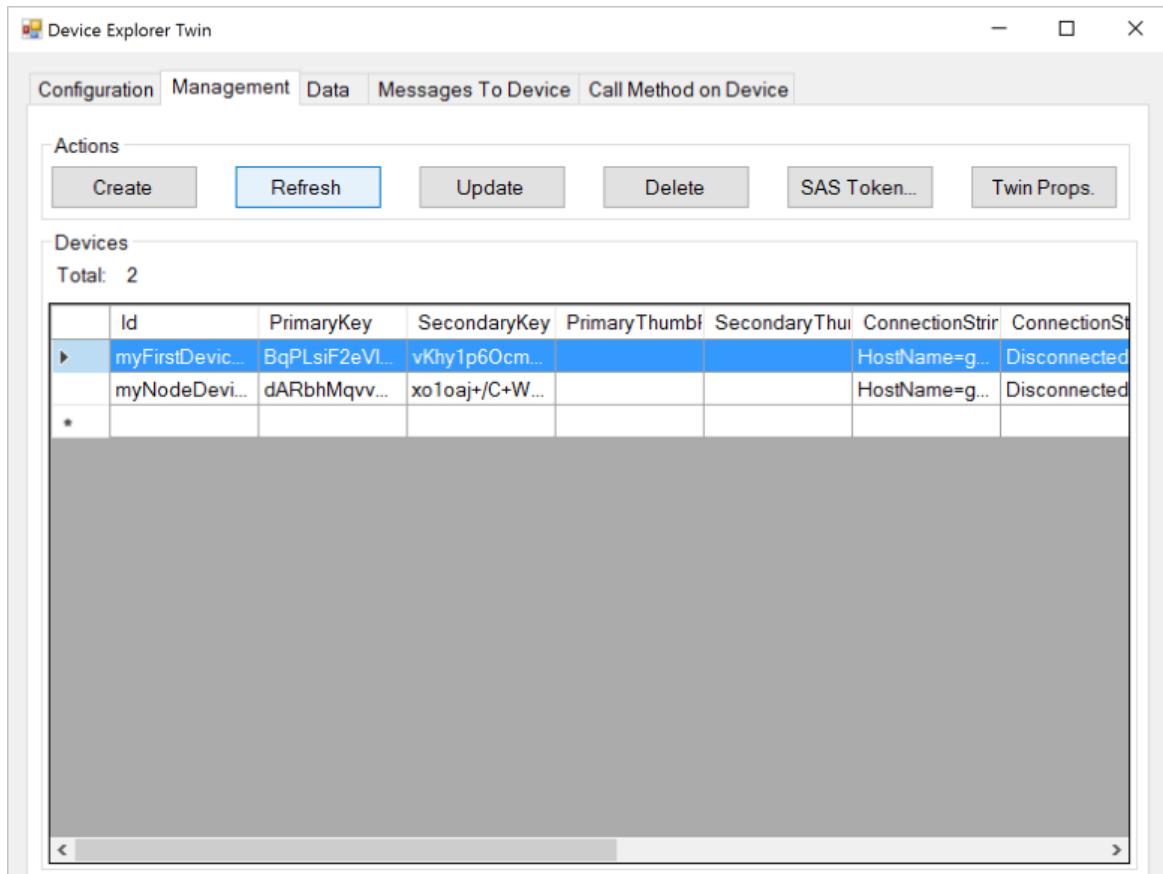
1. To install the device explorer tool, see [How to use the Device Explorer for IoT Hub devices](#).
2. When you run the program, you see this interface:



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.

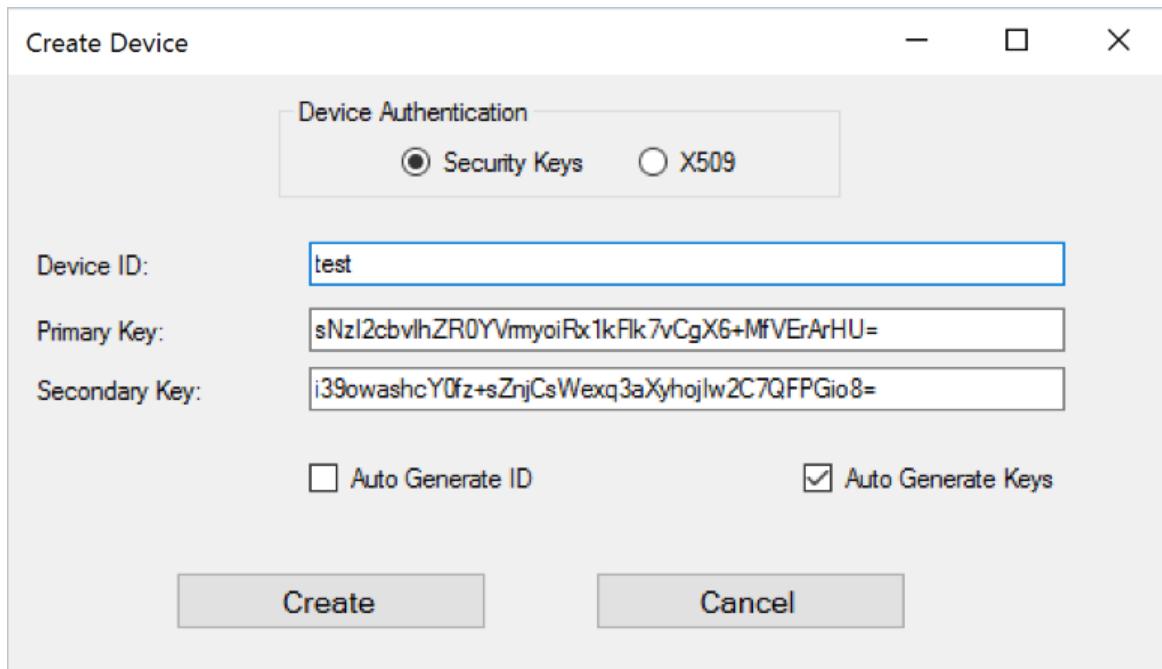
The **Connection String** can be found under **IoT Hub Service > Settings > Shared Access Policy > iothubowner**.

1. When the IoT Hub connection string is configured, click the **Management** tab:

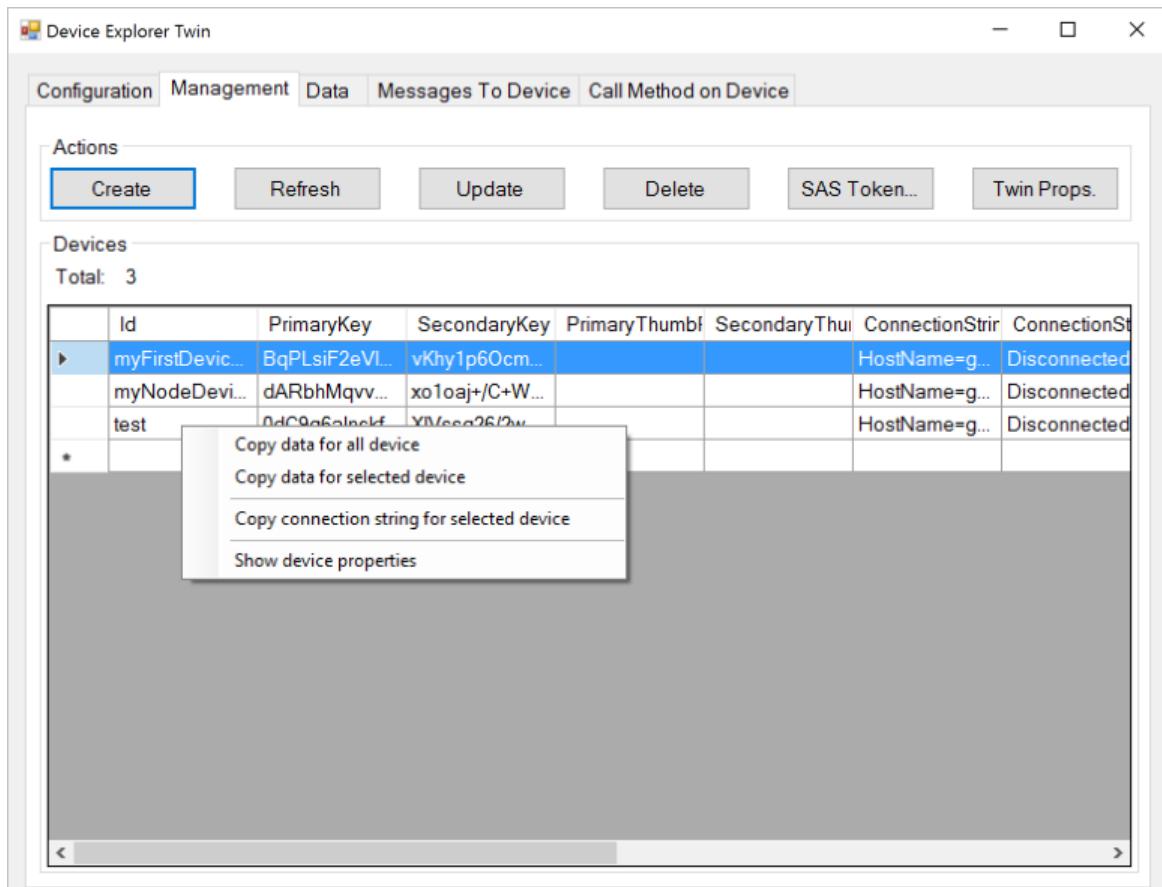


This tab is where you manage the devices registered in your IoT hub.

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

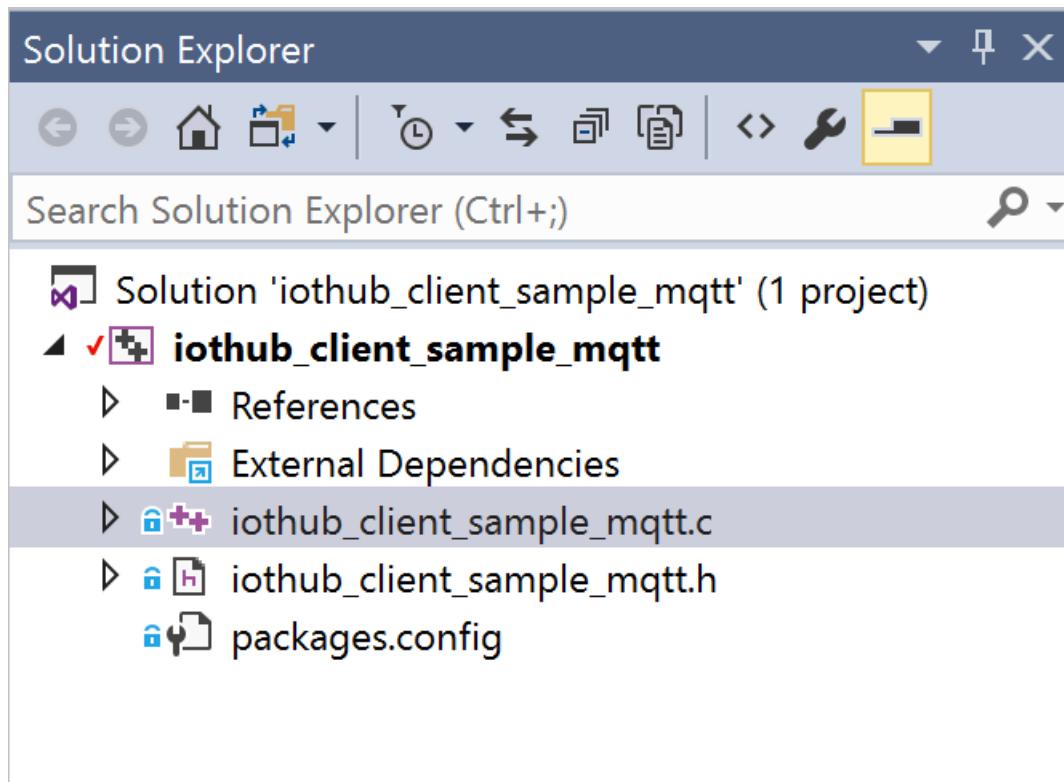
constant at the top of the main source file that enables you to enter a connection string. For example, the corresponding line from the `iothub_client_samples_iothub_convenience_sample` application appears as follows.

```
static const char* connectionString = "[device connection string]";
```

Use the IoTHttpClient 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 Visual Studio asks you to retarget the project to the latest version, accept the prompt.

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.IoTHubClient
- 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 `IoTHubClient` library, you must also include the `Microsoft.Azure.IoTHub.IoTHubClient` 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, MU_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.

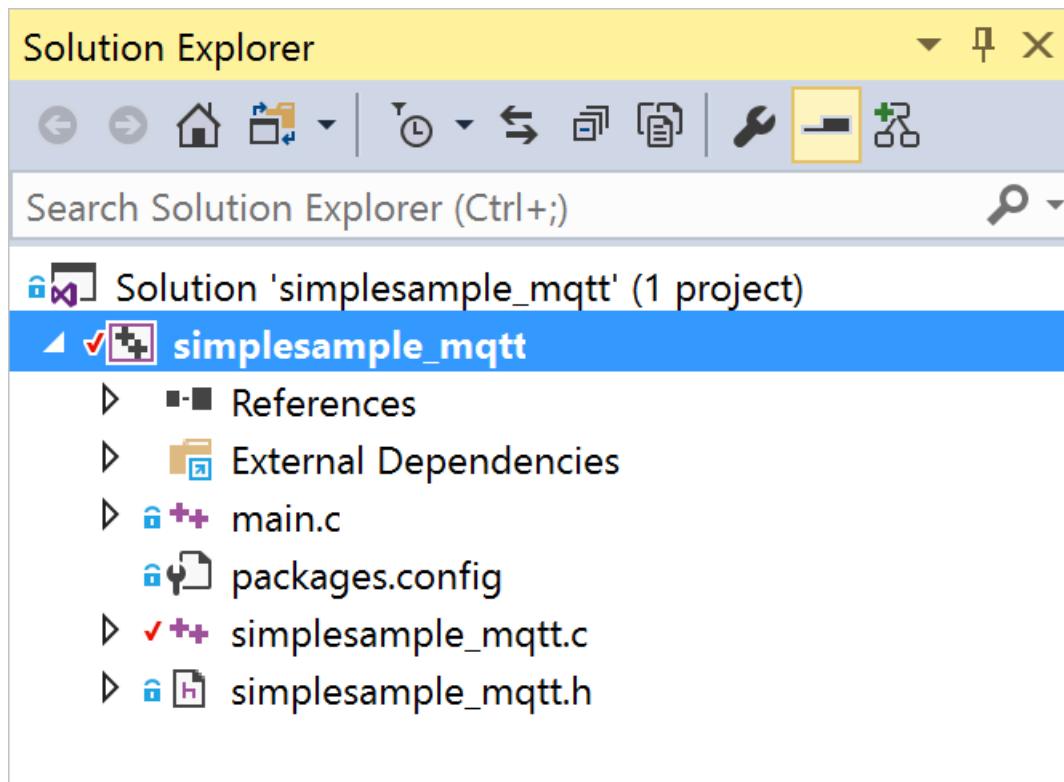
As you can see, it's easy to send and receive messages with the `IoTHubClient` library. The library handles the details of communicating with IoT Hub, including which protocol to use (from the perspective of the developer, this is a simple configuration option).

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 `simplsample_mqtt`. The Windows version of this sample includes the following Visual Studio solution:



NOTE

If Visual Studio asks you to retarget the project to the latest version, accept the prompt.

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
{
    IOOTHUB_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",
MU_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

4/21/2020 • 13 minutes to read • [Edit Online](#)

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**. Batching over AMQP / AMQP-WS, as well as adding system properties on D2C messages, is supported.
- **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

4/21/2020 • 21 minutes to read • [Edit Online](#)

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 **simplsample_amqp** and **simplsample_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 **simplsample_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 and issued the `git submodule update --init` command, then you will find this shared utility library here:

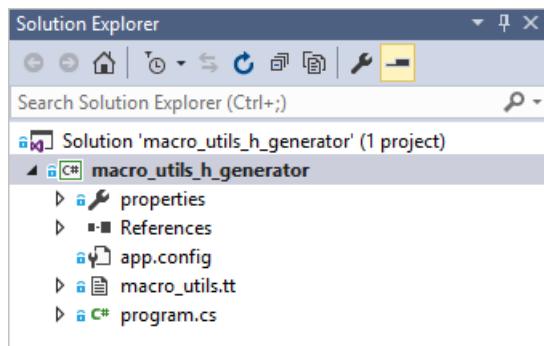
```
.\\c-utility
```

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 |
| X 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 |
| X 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**:

| | | |
|--|---|--------------------------------------|
| X C2143 | syntax error: missing ')' before '(' | simplesample_htt simplesample_htt 34 |
| X C2122 | 'ContosoAnemometer': prototype parameter in name list illegal | simplesample_htt simplesample_htt 34 |
| X C2081 | 'Devideld': name in formal parameter list illegal | simplesample_htt simplesample_htt 34 |
| X 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 icon: Gear icon

| Prerelease | Package | Description | Checkmark |
|--------------------------------------|--|--|-----------|
| Apache.QPID.Proton.AzureDotNet | 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 the following project structure:

- Solution 'simplesample_amqp' (2 projects)
 - serializer
 - External Dependencies
 - Header Files
 - References
 - Resource Files
 - Source Files
 - simplesample_amqp
 - External Dependencies
 - Header Files
 - References
 - Resource Files
 - Source Files
 - main.c
 - simplesample_amqp.c
 - 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

4/21/2020 • 3 minutes to read • [Edit Online](#)

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.

NOTE

- Azure IoT C SDK regularly publishes resource consumption information to help with development. Please visit our [GitHub repository](#) and review the latest benchmark.

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, AMQP, 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

8/8/2019 • 2 minutes to read • [Edit Online](#)

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

5/19/2020 • 4 minutes to read • [Edit Online](#)

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/iOS SDK](#)
- [.NET SDK](#)
- [Java SDK](#)
- [Node SDK](#)
- [Python SDK](#) (Reliability not yet implemented)

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, re-connection, 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 re-connection 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 Architecture Center.

Retry patterns

The following steps describe the retry process when connection errors are detected:

- The SDK detects the error and the associated error in the network, protocol, or application.
- The SDK uses the error filter to determine the error type and decide if a retry is needed.
- 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.
- 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.
- When the defined timeout expires, the SDK stops trying to connect or send. It notifies the user.
- 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, and it is not currently supported on the Python SDK. The Python SDK reconnects as-needed.
- 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/iOS | <code>IOTHUB_CLIENT_RETRY</code> <code>IoTHubClient_SetRetryPolicy</code> | Default: <code>IOTHUB_CLIENT_RETRY_EXponential_BACOFF</code> Custom: use available <code>retryPolicy</code> No retry: <code>IOTHUB_CLIENT_RETRY_NONE</code> | C/iOS implementation |

| SDK | SETRETRYPOLICY METHOD | POLICY IMPLEMENTATIONS | IMPLEMENTATION GUIDANCE |
|--------|---|--|-------------------------------------|
| Java | SetRetryPolicy | Default: ExponentialBackoffWithJitter class Custom: implement IRetryPolicy interface No retry: NoRetry class | Java implementation |
| .NET | DeviceClient.SetRetryPolicy | Default: ExponentialBackoff class Custom: implement IRetryPolicy interface No retry: NoRetry class | C# implementation |
| Node | setRetryPolicy | Default: ExponentialBackoffWithJitter class | Node implementation |
| Python | Not currently supported | Not currently supported | Not currently supported |

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
IRetryPolicy 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
IRetryPolicy 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/iOS SDK](#)
- [.NET SDK](#)
- [Java SDK](#)
- [Node SDK](#)

- [Python SDK](#) (Reliability not yet implemented)

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

3/6/2020 • 7 minutes to read • [Edit Online](#)

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](#)

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|--|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal . | |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.

4. Select **Enter** to run the code.

Create an IoT hub

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

1. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.
 - **IoT Hub Name:** Enter a name for your hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

The screenshot shows the 'Basics' tab of the IoT hub creation wizard. A red box highlights the 'Subscription', 'Resource group', 'Region', and 'IoT hub name' fields. At the bottom, a red box highlights the 'Next: Size and scale >' button.

Home > New > IoT hub

IoT hub
Microsoft

Basics [Size and scale](#) [Tags](#) [Review + create](#)

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

Project details

Choose the subscription you'll use to manage deployments and costs. Use resource groups like folders to help you organize and manage resources.

Subscription * [Personal IoT items](#)

Resource group * [Create new](#)

Region * [East Asia](#)

IoT hub name * [Once your hub is created, this name can't be changed](#)

Review + create [< Previous](#) [Next: Size and scale >](#) [Automation options](#)

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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 [Determines how your IoT hub can scale. You can change this later if your needs increase.](#)

Azure Security Center [On](#) Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.
If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to

categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure resources](#).

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags **Review + create**

Tags are name/value pairs. To categorize resources and consolidate billing, apply the same tag to multiple resources and resource groups. Your tags will update automatically if you change your resources. [Learn more](#)

| Name ⓘ | Value ⓘ | Resource | ⋮ |
|------------|------------|----------|---|
| department | accounting | IoT Hub | |
| | | IoT Hub | |

Review + create < Previous: Size and scale Next: Review + create > Automation options

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags **Review + create**

Basics

| | |
|----------------|------------------|
| Subscription | Personal testing |
| Resource group | iot-hubs |
| Region | West US 2 |
| IoT hub name | you-hub-name |

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 |
| Azure Security Center | 0.001 USD per device per month |

Tags

| | |
|------------|------------|
| department | accounting |
|------------|------------|

Create < Previous: Tags Next > Automation options

8. Select **Create** to create your new 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-iot  
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. Replace **YourIoTHubName** 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 in 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:

```
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.

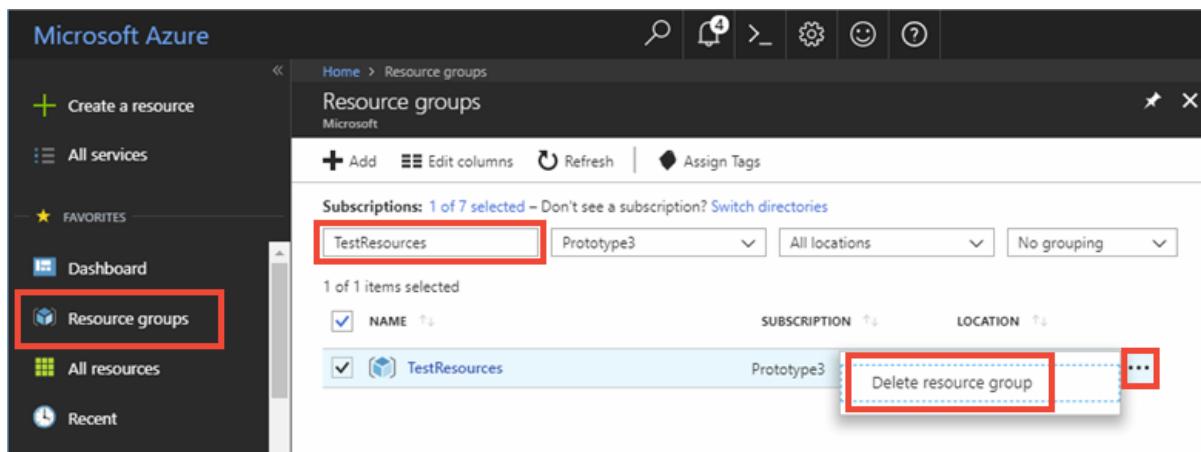
Otherwise, you can delete the Azure resources created in this article to avoid charges.

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 select **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, select ... 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 select **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

11/4/2019 • 4 minutes to read • [Edit Online](#)

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, by default IoT Hub writes the content in Avro format, which has both a message body property and a message property. The Avro format is not used for any other endpoints. 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. IoT Hub now supports writing data to Blob storage in JSON as well as AVRO.

For more information, see [Using Azure Storage as a routing endpoint](#).

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 'Routes' blade in the Azure IoT Hub interface. At the top, there is a note: 'Send data from your devices to endpoints that you choose.' Below this, there are two tabs: 'Routes' (selected) and 'Custom endpoints'. Under the 'Routes' tab, there is a sub-section titled 'Choose which Azure services will receive your messages. You can add up to 10 endpoints to an IoT hub.' It includes a 'Add' button, a 'Synchronize keys' button, and a 'Delete' button. A list of supported endpoints is shown, with 'Blob storage' highlighted with a red box. Below this, there is a section titled 'Recommended for storage.' with a table showing a single entry:

| 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](#)

| 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
<your subscription id>

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 |

4. 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

Add data source

Data sources

| NAME | TYPE |
|------------------------|-----------------------|
| kevinsaydemo (default) | Azure Data Lake Store |
| kevinsayazstorage | Azure Storage |

Search (Ctrl+ /)

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

SETTINGS Firewall

Data sources

5. 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

kevinsaydemo kevinsaydemo Assemblies Avro

NAME SIZE

| | |
|---|---------|
| Avro.dll | 124 KB |
| log4net.dll | 270 KB |
| Microsoft.Analytics.Samples.Formats.dll | 28.2 KB |
| Newtonsoft.Json.dll | 654 KB |

Search (Ctrl+ /)

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

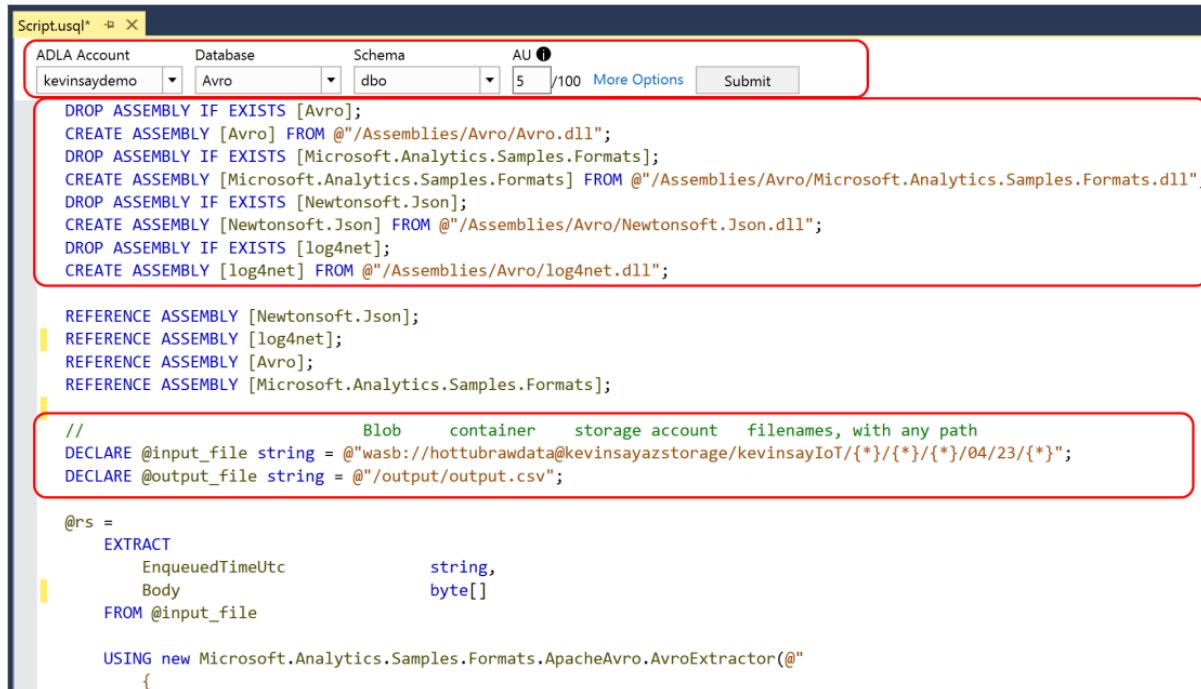
SETTINGS Encryption Firewall Pricing tier Properties Locks Automation script

DATA LAKE STORE Quick start Data explorer

6. 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 U-SQL script editor window titled "Script.usql". The top navigation bar includes "File", "Edit", "View", "Run", "Help", and "Exit". Below the title, there are dropdown menus for "ADLA Account" (set to "kevinsaydemo"), "Database" (set to "Avro"), "Schema" (set to "dbo"), and "AU" (set to "5 /100"). A "More Options" button and a "Submit" button are also present. The main code area contains several U-SQL statements:

```

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 = @"wab://hottubrawdata@kevinsayazstorage/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""]
        }
    ]
}
);

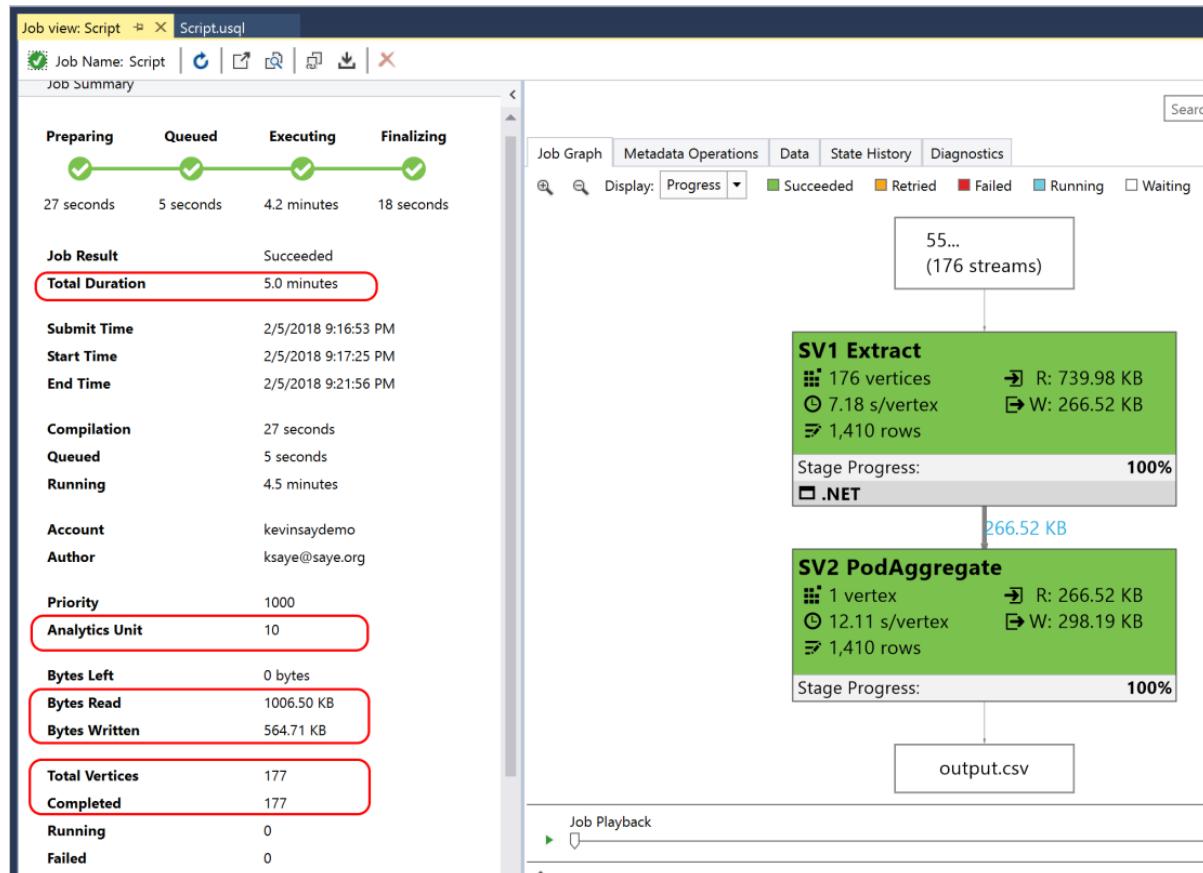
@cnt =
SELECT EnqueuedTimeUtc AS time, Encoding.UTF8.GetString(Body) AS jsonmessage
FROM @rs;

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.int... |
| 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.int... |

To parse the JSON, continue to step 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 | Properties |
| 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

5/21/2020 • 9 minutes to read • [Edit Online](#)

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.

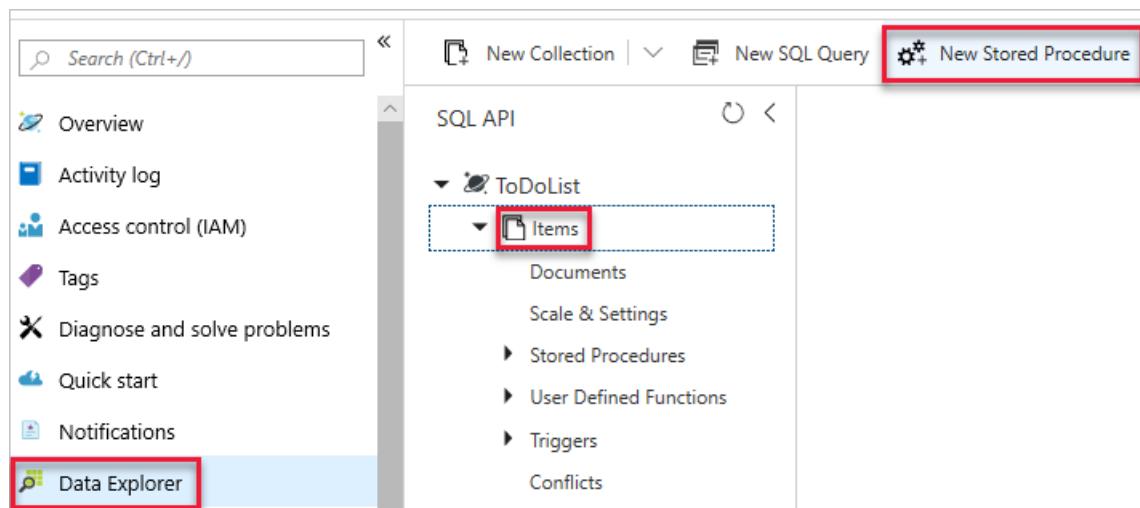
Prerequisites

- 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. When you create your collection, use `/id` for the partition key.
- 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 `LatestDeviceConnectionState` for the 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()
```

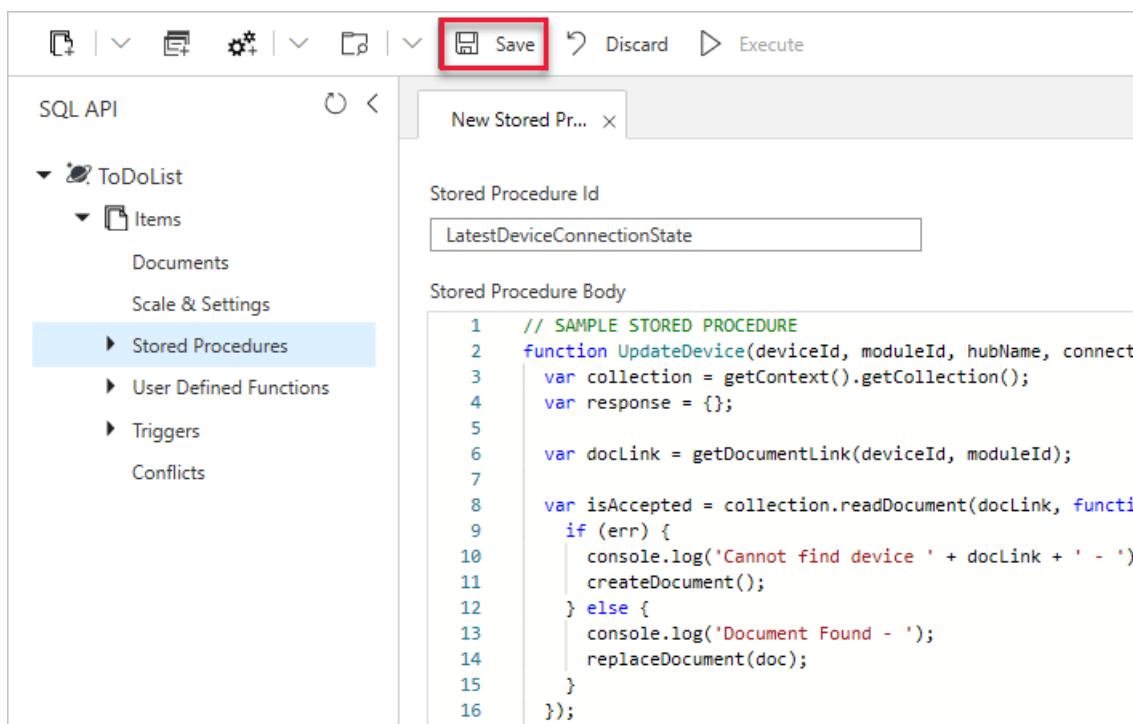
```

    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

1. In the [Azure portal](#), select **+Create a resource**, select **Integration** and then **Logic App**.

The screenshot shows the Azure Marketplace interface. On the left, there's a sidebar with various service categories like Home, Dashboard, All services, and Favorites. The 'Logic App' service is highlighted with a red box. At the top, there's a search bar labeled 'Search the Marketplace'. Below the search bar, there are tabs for 'Azure Marketplace' (selected), 'See all', and 'Featured'. The 'Featured' tab is also highlighted with a red box. A list of services is shown, each with an icon, name, and a 'Quickstart tutorial' link. The 'Integration' service is also highlighted with a red dashed box.

| Service | Icon | Description | Link |
|---|--------------------|---|-------------------------------------|
| Integration Service Environment (preview) | User icon | Integration Service Environment (preview) | Learn more |
| Logic App | { icon } | Logic App | Quickstart tutorial |
| API management | Cloud icon | API management | Quickstart tutorial |
| Service Bus | Envelope icon | Service Bus | Quickstart tutorial |
| Integration Account | Yellow square icon | Integration Account | Quickstart tutorial |
| Logic Apps Custom Connector | Connector icon | Logic Apps Custom Connector | Learn more |
| Data Factory | Factory icon | Data Factory | Quickstart tutorial |

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.

The screenshot shows the 'Logic App' creation wizard. It has several input fields:

- * Name: 'eventgrid-logic-app' (highlighted with a purple box)
- * Subscription: 'Microsoft Azure Internal Consumption (a42)' (highlighted with a blue box)
- * Resource group:
 - Create new
 - Use existing

'contoso-order-eventgrid' (highlighted with a blue box)
- * Location: 'West US 2'
- Log Analytics:
 - On
 - Off

A note at the bottom says: 'You can add triggers and actions to your Logic App after creation.' At the bottom, there are 'Create' and 'Automation options' buttons.

3. Select **Create** to create the logic app.

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

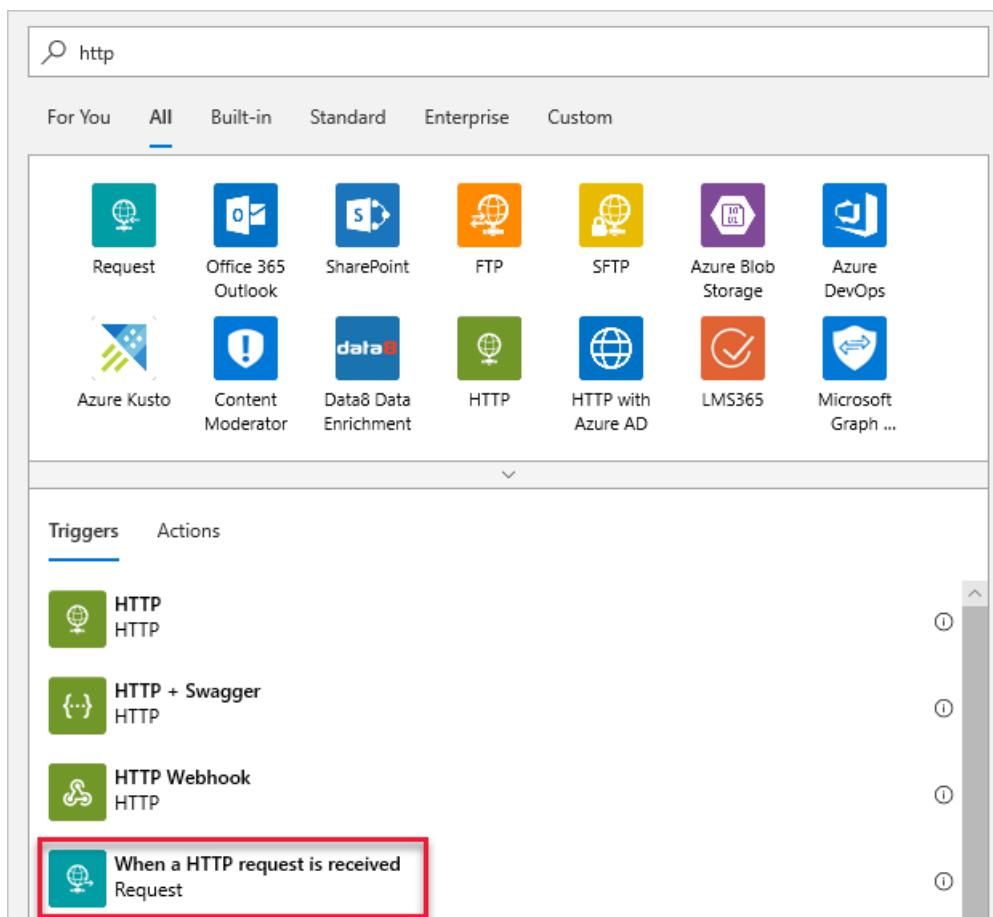
To find and open your logic app again, select **Resource groups** and select the resource group you are using for this how-to. Then select your new logic app. This opens the Logic App Designer.

4. In the Logic App Designer, scroll to the right until you see common triggers. 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** and hit Enter.
2. Select **Request - When an HTTP request is received** as the trigger.



3. Select **Use sample payload to generate schema**.

When a HTTP request is received

HTTP POST URL URL will be generated after save [Copy]

Request Body JSON Schema

Use sample payload to generate schema

Add new parameter ▼

4. Paste the following sample JSON code into the text box, then select Done:

Enter or paste a sample JSON payload.

Dane

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.

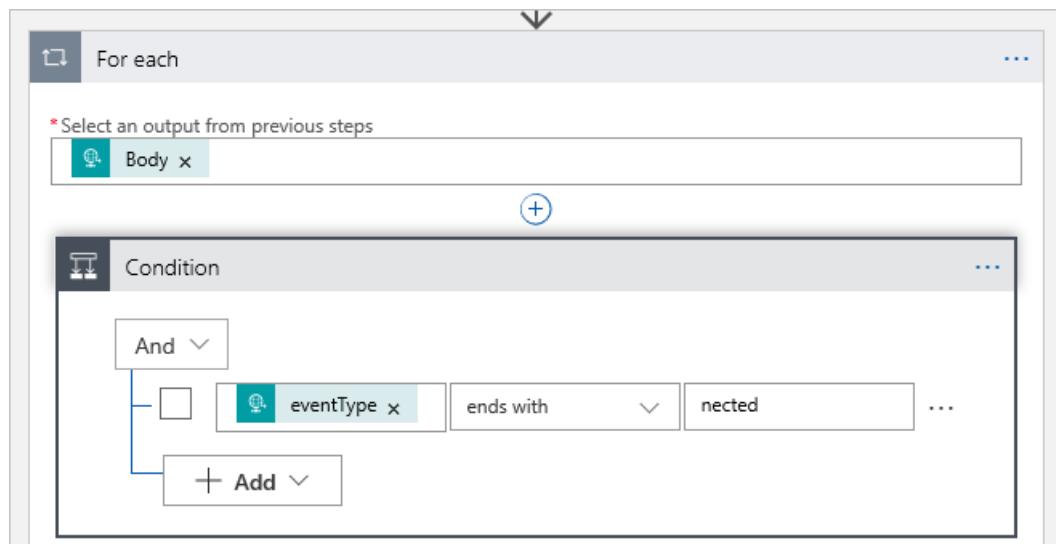
Create a condition

In your logic app workflow, conditions help run specific actions after passing that specific condition. Once the

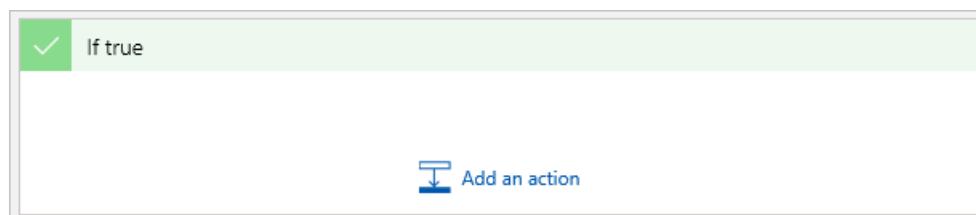
condition is met, a desired action can be defined. For this tutorial, the condition is to check whether eventType is device connected or device disconnected. The action will be to execute the stored procedure in your database.

1. Select + New step then Built-in, then find and select Condition. Click in Choose a value and a box will pop up showing the Dynamic content -- the fields that can be selected. Fill in the fields as shown below to only execute this for Device Connected and Device Disconnected events:

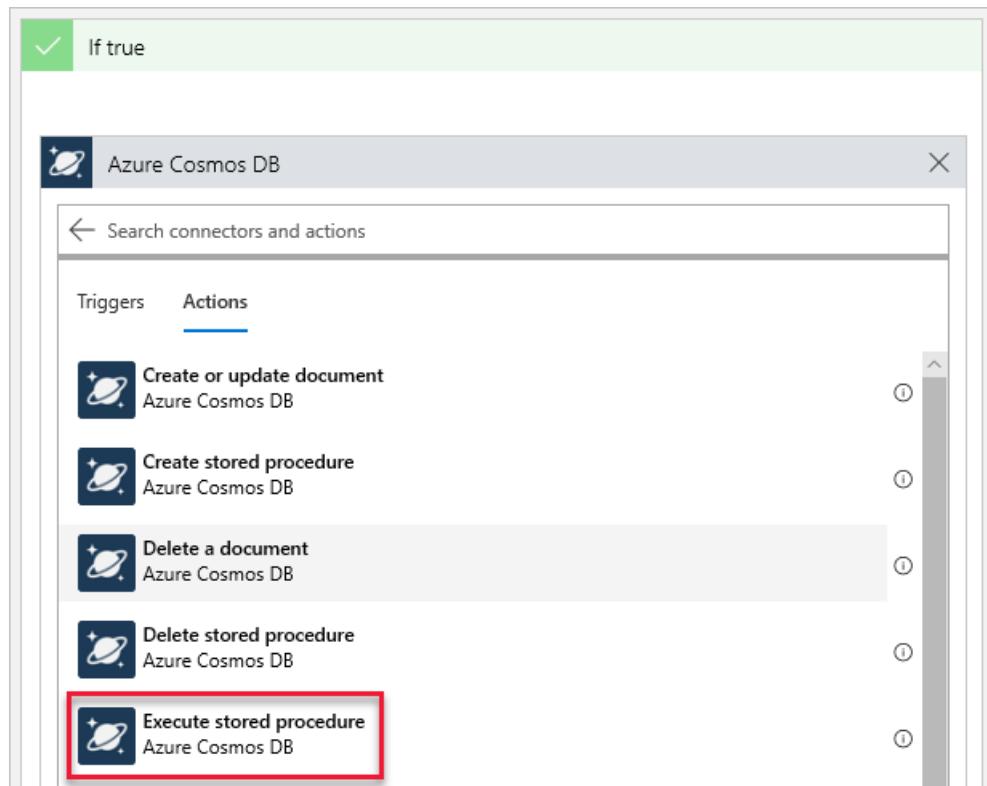
- Choose a value: **eventType** -- select this from the fields in the dynamic content that appear when you click on this field.
- Change "is equal to" to **ends with**.
- Choose a value: **nected**.



2. In the if true dialog, click on Add an action.



3. Search for Cosmos DB and select Azure Cosmos DB - Execute stored procedure



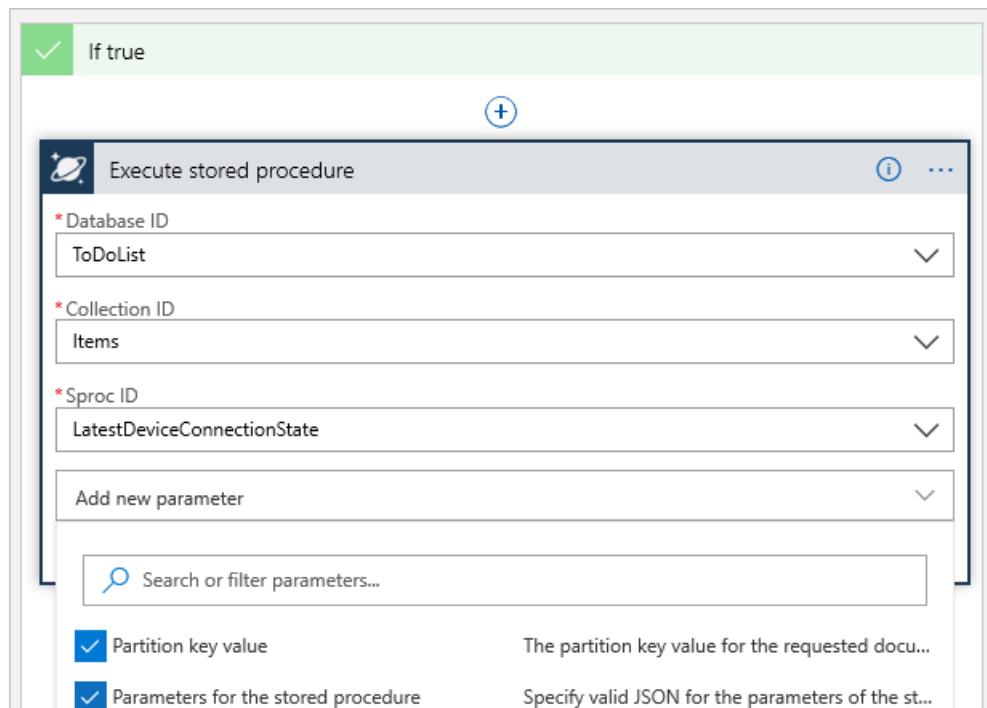
- Fill in **cosmosdb-connection** for the **Connection Name** and select the entry in the table, then select **Create**. You see the **Execute stored procedure** panel. Enter the values for the fields:

Database ID: ToDoList

Collection ID: Items

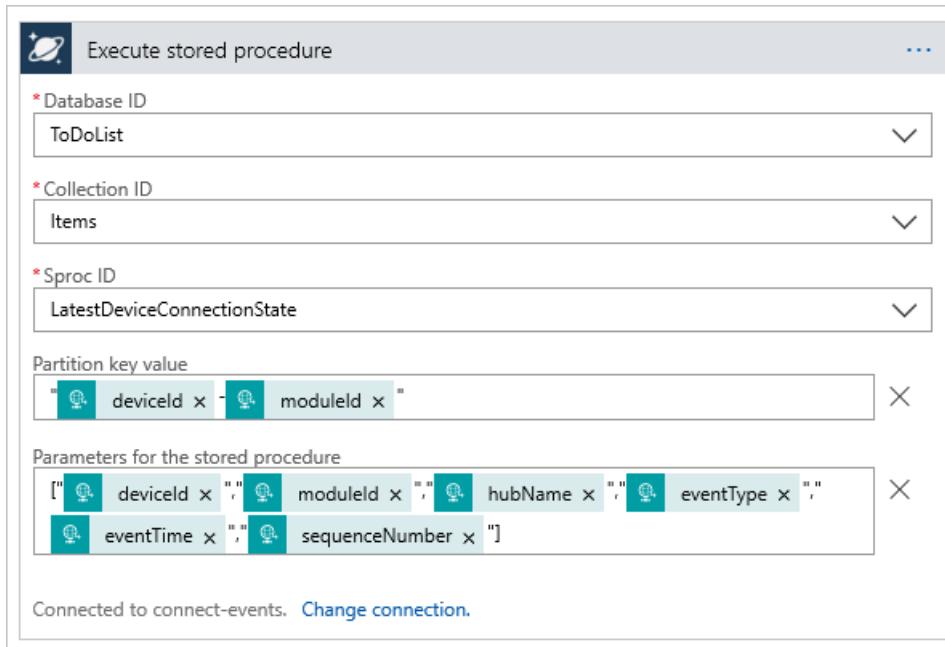
Sproc ID: LatestDeviceConnectionState

- Select **Add new parameter**. In the dropdown that appears, check the boxes next to **Partition key** and **Parameters for the stored procedure**, then click anywhere else on the screen; it adds a field for partition key value and a field for parameters for the stored procedure.



- Now enter the partition key value and parameters as shown below. Be sure to put in the brackets and double-quotes as shown. You may have to click **Add dynamic content** to get the valid values you can use

here.



- At the top of the pane where it says **For Each**, under **Select an output from previous steps**, make sure it **Body** is selected.

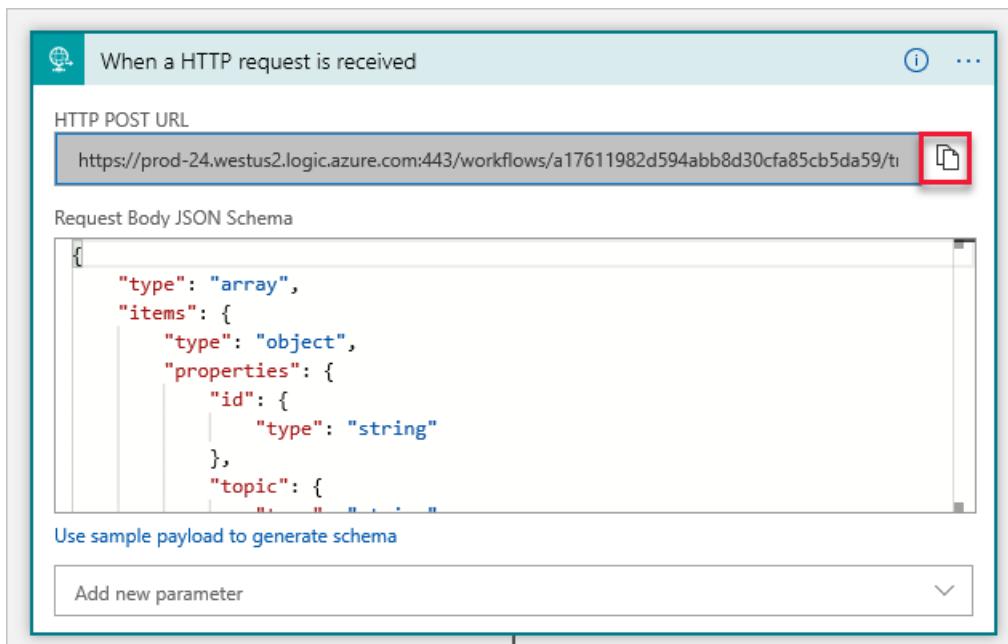


- Save your logic app.

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.

- Expand the **When a HTTP request is received** trigger configuration box by clicking on it.
- Copy the value of **HTTP POST URL** by selecting the copy button next to it.



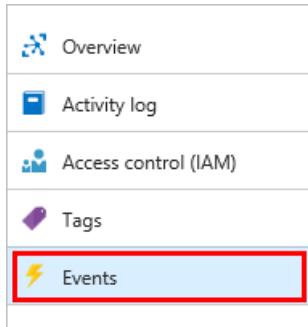
3. Save this URL so that you can refer to it in the next section.

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.

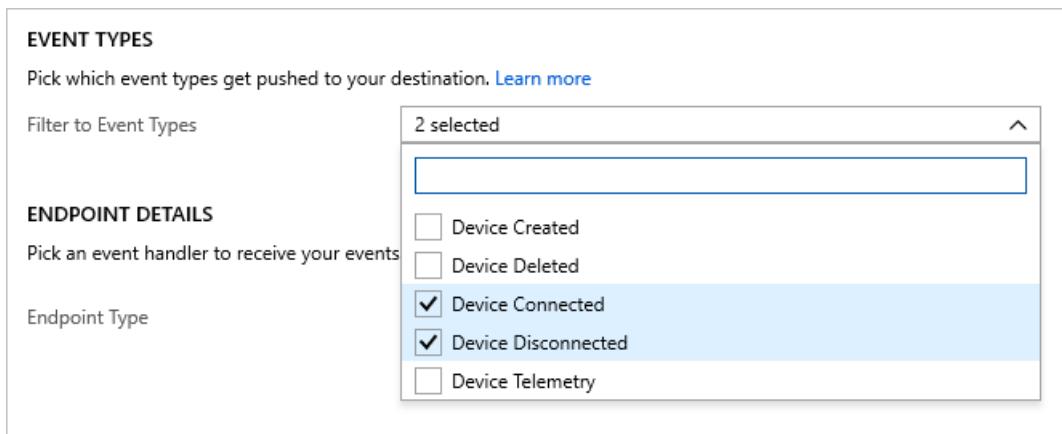
2. Select **Events**.



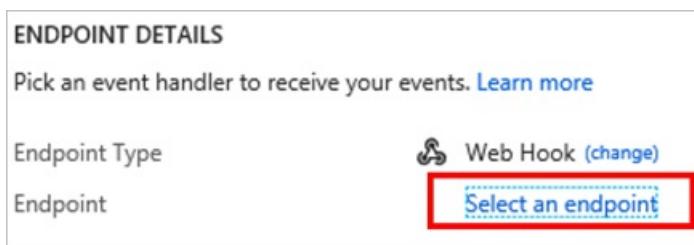
3. Select + Event subscription.



4. Fill in **Event Subscription Details**: Provide a descriptive name and select **Event Grid Schema**.
5. Fill in the **Event Types** fields. In the dropdown list, select only **Device Connected** and **Device Disconnected** from the menu. Click anywhere else on the screen to close the list and save your selections.



6. For **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.



7. The form should now look similar to the following example:

 **Create Event Subscription**

Event Grid

Basic **Filters** Additional Features

Event Subscriptions listen for events emitted by the topic resource and send them to the endpoint resource. [Learn more](#)

EVENT SUBSCRIPTION DETAILS

Name ✓

Event Schema

TOPIC DETAILS

Pick a topic resource for which events should be pushed to your destination. [Learn more](#)

Topic Type  IoT Hub

Topic Resource [test-order-events](#)

EVENT TYPES

Pick which event types get pushed to your destination. [Learn more](#)

Filter to Event Types

ENDPOINT DETAILS

Pick an event handler to receive your events. [Learn more](#)

Endpoint Type  Web Hook ([change](#))

Endpoint <https://prod-24.westus2.logic.azure.com:443/workflows/a17...> ([change](#))

Create

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** at the top of the pane.
3. For **Device ID**, enter .
4. Select **Save**.
5. You can add multiple devices with different device IDs.

The screenshot shows the Azure IoT Hub management interface for the 'test-order-events' hub. On the left, a sidebar lists various management options like Overview, Activity log, Access control (IAM), Tags, Events, Settings, Explorers, and IoT devices. The 'IoT devices' option is selected and highlighted with a red box. The main area displays a table of devices with one entry: 'Demo-Device-1' (Status: Enabled). At the top, there's a search bar, an 'Add' button (also highlighted with a red box), a 'Refresh' button, and a 'Delete' button.

- Click on the device again; now the connection strings and keys will be filled in. Copy the **Connection string -- primary key** for later use.

The screenshot shows the 'Device details' page for 'Demo-Device-1'. It displays the device ID ('Demo-Device-1'), primary key ('OpOUKIR6JCZjDQpoutlTc0T/i3bgVlwPwmj4ltYuAE='), secondary key ('hhqlvr+z8TJoRm0BQTHpbGDLbetUENJx2rmQSiGSOB0='), and connection strings. The 'Connection string (primary key)' field is highlighted with a red box and contains the value 'HostName=test-order-events.azure-devices.net;DeviceId=Demo-Device-1;SharedAccessKey=OpOUKIR6'.

Start Raspberry Pi simulator

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.

- In the coding area, replace the placeholder in Line 15 with your Azure IoT Hub device connection string that you saved at the end of the previous section.

```

14
15 const connectionString = '[Your IoT hub device connection string]';
16 const LEDPin = 4;
17

```

2. Run the application by selecting Run.

You see something similar to the following output that shows the sensor data and the messages that are sent to your IoT hub.

```
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":24.7546640485}
>
Message sent to Azure IoT Hub
>
Sending message: {"messageId":2,"deviceId":"Raspberry Pi Web Client","temperature":23.4544804284}
>
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.

The screenshot shows the Azure portal interface for managing Cosmos DB documents. On the left, a list of documents is displayed with columns for **id** and **/id**. The list includes entries for Demo-Device-13, Demo-Device-25, Demo-Device-23, Demo-Device-3, Demo-Device-12, Demo-Device-9, Demo-Device-5, and Demo-Device-6. The row for Demo-Device-23 is currently selected. On the right, the details for this selected document are shown in a JSON format:

```
1 {  
2   "id": "Demo-Device-23-",  
3   "deviceId": "Demo-Device-23",  
4   "moduleId": "",  
5   "hubName": "MYIOTHUB",  
6   "connectionState": "Microsoft.Devices.DeviceConnected",  
7   "connectionStateUpdatedTime": "2018-07-10T20:26:18.3142711+00:00",  
8   "sequenceNumber": "000000000000000001D4188B20571F480000000200000000000000000000000000000029",  
9   "_rid": "g3FiAON1EAcYFgAAAAAAA=",  
10  "_self": "dbs/g3FiAA=/colls/g3FiAON1EAc=/docs/g3FiAON1EAcYFgAAAAAAA=/",  
11  "_etag": "\"0a0003ee-0000-0000-0000-5b4516720000\"",  
12  "_attachments": "attachments/",  
13  "_ts": 1531254386  
14 }
```

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 finished 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.

3. Navigate to your IoT hub.

4. On the **Overview** blade, select **Delete**.

Even if you keep your IoT hub, you may want to delete the event subscription that you created.

5. In your IoT hub, select **Event Grid**.

6. Select the event subscription that you want to remove.

7. 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)

7/29/2020 • 7 minutes to read • [Edit Online](#)

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](#) quickstart 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 available only in the standard tier of IoT Hub. For more information about the basic and standard/free IoT Hub tiers, see [Choose the right IoT Hub tier](#).

This tutorial builds on [Send telemetry from a device to an IoT hub](#). It shows you how to do the following tasks:

- 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 acknowledgment (*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**. This app connects to your IoT hub and receives cloud-to-device messages. This app is a modified version of the app created in [Send telemetry from a device to an IoT hub](#).
- **SendCloudToDevice**. This app sends a cloud-to-device message to the device app through IoT Hub, and then receives its delivery acknowledgment.

NOTE

IoT Hub has SDK support for many device platforms and languages, including C, Java, Python, 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](#).

Prerequisites

- Visual Studio
- An active Azure account. If you don't have an account, you can create a [free account](#) in just a couple of minutes.
- Make sure that port 8883 is open in your firewall. The device sample in this article uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

Receive messages in the device app

In this section, 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 **SimulatedDevice** class.

```
private static async void ReceiveC2dAsync()
{
    Console.WriteLine("\nReceiving cloud to device messages from service");
    while (true)
    {
        Message receivedMessage = await s_deviceClient.ReceiveAsync();
        if (receivedMessage == null) continue;

        Console.ForegroundColor = ConsoleColor.Yellow;
        Console.WriteLine("Received message: {0}",
            Encoding.ASCII.GetString(receivedMessage.GetBytes()));
        Console.ResetColor();

        await s_deviceClient.CompleteAsync(receivedMessage);
    }
}
```

2. Add the following method in the **Main** method, right before the `Console.ReadLine()` line:

```
ReceiveC2dAsync();
```

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 example, 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. The message processing logic in the device app must be *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](#).

Get the IoT hub connection string

In this article, you create a back-end service to send cloud-to-device messages through the IoT hub you created in [Send telemetry from a device to an IoT hub](#). To send cloud-to-device messages, your service needs the **service**

connect permission. By default, every IoT Hub is created with a shared access policy named **service** that grants this permission.

To get the IoT Hub connection string for the **service** policy, follow these steps:

1. In the [Azure portal](#), select **Resource groups**. Select the resource group where your hub is located, and then select your hub from the list of resources.
2. On the left-side pane of your IoT hub, select **Shared access policies**.
3. From the list of policies, select the **service** policy.
4. Under **Shared access keys**, select the copy icon for the **Connection string -- primary key** and save the value.

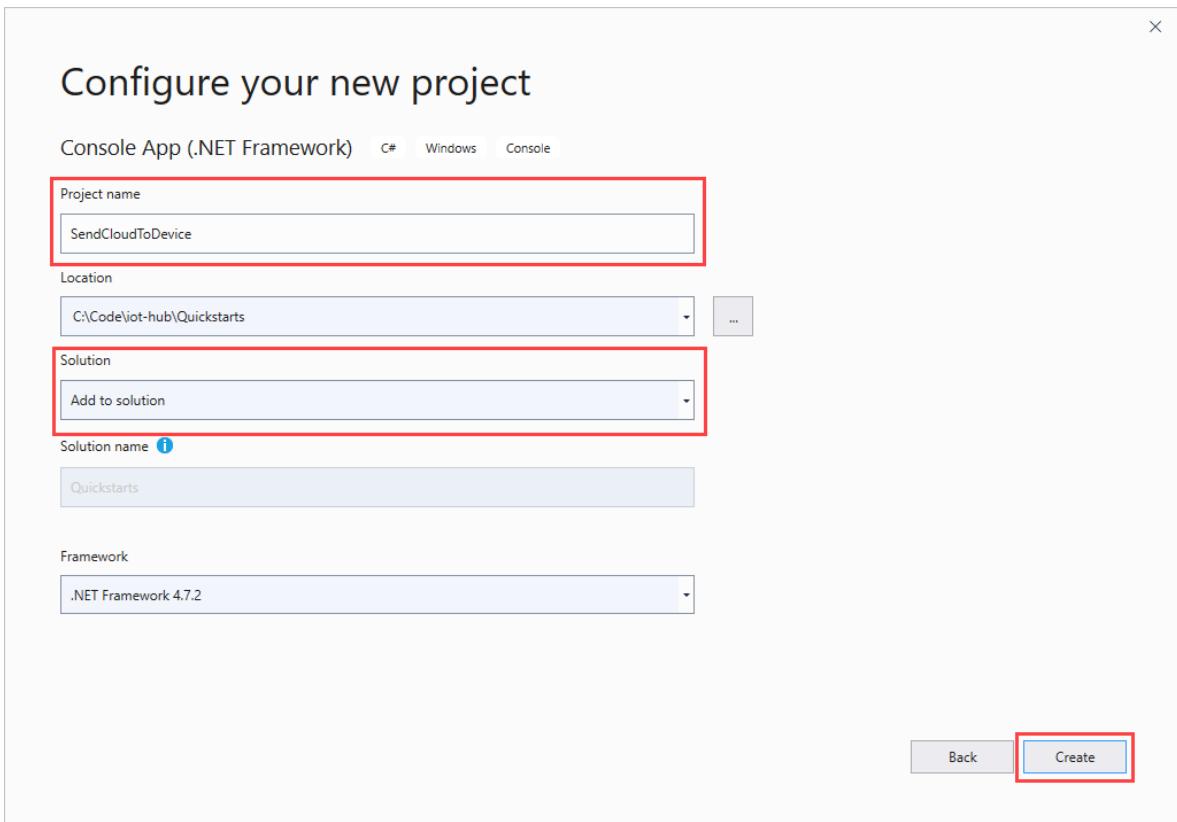
The screenshot shows the Azure portal interface for managing an IoT hub. On the left, the navigation pane includes options like Overview, Activity log, Access control (IAM), Tags, Events, Settings, and Shared access policies (which is highlighted with a red box). The main content area displays the 'Shared access policies' for the 'contoso-hub-1' IoT hub. A table lists four policies: 'iothubowner' (permissions: registry write, service connect), 'device' (permissions: device connect), 'registryRead' (permissions: registry read), and 'registryReadWrite' (permissions: registry write). On the right, a detailed view of the 'service' policy is shown, with the 'Access policy name' set to 'service'. Under 'Permissions', 'Service connect' is checked. The 'Shared access keys' section contains three fields: 'Primary key' (redacted), 'Secondary key' (redacted), and 'Connection string—primary key' (redacted, which is also highlighted with a red box). Buttons for Save, Discard, and More are at the top right of the details view.

For more information about IoT Hub shared access policies and permissions, see [Access control and permissions](#).

Send a cloud-to-device message

In this section, you create a .NET console app that sends cloud-to-device messages to the simulated device app.

1. In the current Visual Studio solution, select **File > New > Project**. In **Create a new project**, select **Console App (.NET Framework)**, and then select **Next**.
2. Name the project *SendCloudToDevice*. Under **Solution**, select **Add to solution** and accept the most recent version of the .NET Framework. Select **Create** to create the project.



3. In Solution Explorer, right-click the new project, and then select **Manage NuGet Packages**.
4. In **Manage NuGet Packages**, select **Browse**, and then search for and select **Microsoft.Azure.Devices**. Select **Install**.

This step downloads, installs, and adds a reference to the [Azure IoT service SDK NuGet package](#).

5. Add the following `using` statement at the top of the **Program.cs** file.

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

6. Add the following fields to the **Program** class. Replace the `{iot hub connection string}` placeholder value with the IoT hub connection string you noted previously in [Get the IoT hub connection string](#). Replace the `{device id}` placeholder value with the device ID of the device you added in the [Send telemetry from a device to an IoT hub](#) quickstart.

```
static ServiceClient serviceClient;
static string connectionString = "{iot hub connection string}";
static string targetDevice = "{device id}";
```

7. Add the following method to the **Program** class to send a message to your device.

```
private async static Task SendCloudToDeviceMessageAsync()
{
    var commandMessage = new
        Message(Encoding.ASCII.GetBytes("Cloud to device message."));
    await serviceClient.SendAsync(targetDevice, commandMessage);
}
```

8. 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();

```

9. In Solutions Explorer, right-click your solution, and select **Set StartUp Projects**.
10. In **Common Properties > Startup Project**, select **Multiple startup projects**, then select the **Start** action for **SimulatedDevice** and **SendCloudToDevice**. Select **OK** to save your changes.
11. Press **F5**. Both applications should start. Select the **SendCloudToDevice** window, and press **Enter**. You should see the message being received by the device app.

The screenshot shows a terminal window with the following text:

```

Send Cloud-to-Device message
Press any key to send a C2D message.

Receiving cloud to device messages from service
7/7/2020 5:15:36 PM > Sending message: {"temperature":25.779664304936148,"humidity":66.023079541522577}
7/7/2020 5:15:37 PM > Sending message: {"temperature":27.613294305099778,"humidity":68.0059813838481}
7/7/2020 5:15:38 PM > Sending message: {"temperature":20.42217853265916,"humidity":65.290552091454416}
7/7/2020 5:15:39 PM > Sending message: {"temperature":27.164465224912608,"humidity":73.6231740720678}
7/7/2020 5:15:40 PM > Sending message: {"temperature":25.212272100715094,"humidity":67.1568628620155441}
7/7/2020 5:15:41 PM > Sending message: {"temperature":27.51779631467927,"humidity":67.704069794941771}
7/7/2020 5:15:42 PM > Sending message: {"temperature":34.79974787440139,"humidity":63.2013792745775481}
7/7/2020 5:15:43 PM > Sending message: {"temperature":29.963241745700707,"humidity":64.471686447258891}
7/7/2020 5:15:44 PM > Sending message: {"temperature":30.809200411107952,"humidity":79.456074982628266}
Received message: Cloud to device message
7/7/2020 5:15:45 PM > Sending message: {"temperature":34.295896705843461,"humidity":69.86340051976191}
7/7/2020 5:15:46 PM > Sending message: {"temperature":22.320016635265208,"humidity":71.108831498356921}
7/7/2020 5:15:47 PM > Sending message: {"temperature":21.283014419620397,"humidity":77.299586021015231}
7/7/2020 5:15:49 PM > Sending message: {"temperature":32.96121161801797,"humidity":68.036658339219471}
7/7/2020 5:15:50 PM > Sending message: {"temperature":24.707021273070492,"humidity":78.088671508286467}
7/7/2020 5:15:51 PM > Sending message: {"temperature":34.915918013973126,"humidity":68.998479586559569}
7/7/2020 5:15:52 PM > Sending message: {"temperature":30.570210090638234,"humidity":60.358102806079252}
7/7/2020 5:15:53 PM > Sending message: {"temperature":22.920435305182092,"humidity":60.668926108940006}
7/7/2020 5:15:54 PM > Sending message: {"temperature":22.94287644696556,"humidity":72.6450149494433}
7/7/2020 5:15:55 PM > Sending message: {"temperature":22.830662356145055,"humidity":62.326911875199016}
7/7/2020 5:15:56 PM > Sending message: {"temperature":26.731488433075832,"humidity":75.591355709168766}
7/7/2020 5:15:57 PM > Sending message: {"temperature":31.3221593486714,"humidity":65.813301823015}
7/7/2020 5:15:58 PM > Sending message: {"temperature":23.771187036191666,"humidity":71.523696888016389}
7/7/2020 5:15:59 PM > Sending message: {"temperature":27.7537165173493,"humidity":76.683244275247318}
7/7/2020 5:16:00 PM > Sending message: {"temperature":22.952384675365121,"humidity":67.898918654722593}
7/7/2020 5:16:15 PM > Sending message: {"temperature":27.86056708395364,"humidity":63.469839898622517}

```

Receive delivery feedback

It is possible to request delivery (or expiration) acknowledgments 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 line in the **Main** method, right after

```
serviceClient = ServiceClient.CreateFromConnectionString(connectionString) .
```

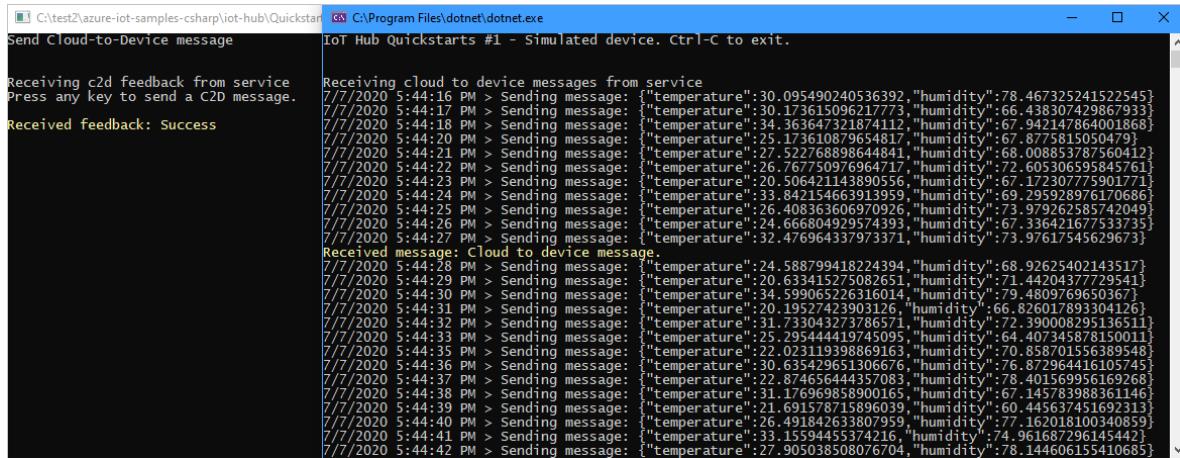
```
ReceiveFeedbackAsync();
```

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(...);
```

```
commandMessage.Ack = DeliveryAcknowledgement.Full;
```

4. Run the apps by pressing **F5**. You should see both applications start. Select the **SendCloudToDevice** window, 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.



The screenshot shows two terminal windows. The left window is titled "Send Cloud-to-Device message" and contains the following text:

```
Receiving c2d feedback from service
Press any key to send a C2D message.

Received feedback: Success
```

The right window is titled "IoT Hub Quickstarts #1 - Simulated device. Ctrl-C to exit." and contains the following text:

```
Receiving cloud to device messages from service
7/7/2020 5:44:16 PM > Sending message: {"temperature":30.095490240536392, "humidity":78.467325241522545}
7/7/2020 5:44:17 PM > Sending message: {"temperature":30.173615096217773, "humidity":66.438307429867933}
7/7/2020 5:44:18 PM > Sending message: {"temperature":34.363647321874112, "humidity":67.942147864001868}
7/7/2020 5:44:20 PM > Sending message: {"temperature":25.173610879654817, "humidity":67.8775815050479}
7/7/2020 5:44:21 PM > Sending message: {"temperature":27.522768898644841, "humidity":68.008853787560412}
7/7/2020 5:44:22 PM > Sending message: {"temperature":26.767750976964717, "humidity":72.605306595845761}
7/7/2020 5:44:23 PM > Sending message: {"temperature":20.50642114389056, "humidity":67.172307775901771}
7/7/2020 5:44:24 PM > Sending message: {"temperature":33.842154663913959, "humidity":79.295928976170686}
7/7/2020 5:44:25 PM > Sending message: {"temperature":26.408363606970926, "humidity":73.979262585742049}
7/7/2020 5:44:26 PM > Sending message: {"temperature":24.666804929574393, "humidity":67.336421677533735}
7/7/2020 5:44:27 PM > Sending message: {"temperature":32.476964337973371, "humidity":73.97617545623673}

Received message: Cloud to device message.
7/7/2020 5:44:28 PM > Sending message: {"temperature":24.588799418224294, "humidity":68.92625402143517}
7/7/2020 5:44:29 PM > Sending message: {"temperature":20.633415275082654, "humidity":71.44204377729541}
7/7/2020 5:44:30 PM > Sending message: {"temperature":34.599065226316014, "humidity":79.48097696503671}
7/7/2020 5:44:31 PM > Sending message: {"temperature":20.19527423903126, "humidity":66.826017893304126}
7/7/2020 5:44:32 PM > Sending message: {"temperature":31.732043273786571, "humidity":72.390008295136511}
7/7/2020 5:44:33 PM > Sending message: {"temperature":25.295444419745095, "humidity":64.407345878150011}
7/7/2020 5:44:33 PM > Sending message: {"temperature":22.023119398869163, "humidity":70.858701556389548}
7/7/2020 5:44:35 PM > Sending message: {"temperature":30.635429651906676, "humidity":76.872964416105745}
7/7/2020 5:44:37 PM > Sending message: {"temperature":22.874656444357083, "humidity":78.401569956169268}
7/7/2020 5:44:38 PM > Sending message: {"temperature":31.176969858900165, "humidity":67.145783988361146}
7/7/2020 5:44:39 PM > Sending message: {"temperature":21.691578715896039, "humidity":60.445637451692311}
7/7/2020 5:44:40 PM > Sending message: {"temperature":26.491842633807959, "humidity":77.6248100340893}
7/7/2020 5:44:41 PM > Sending message: {"temperature":33.15594455374216, "humidity":74.961687296145442}
7/7/2020 5:44:42 PM > Sending message: {"temperature":27.905038508076704, "humidity":78.144606155410685}
```

NOTE

For simplicity, this tutorial does not implement any retry policy. In production code, you should implement retry policies, such as exponential backoff, as suggested in [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)

7/29/2020 • 6 minutes to read • [Edit Online](#)

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](#) quickstart 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 available only in the standard tier of IoT Hub. For more information about the basic and standard/free IoT Hub tiers, see [Choose the right IoT Hub tier](#).

This tutorial builds on [Send telemetry from a device to an IoT hub](#). 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 acknowledgment (*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 an IoT hub](#), 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 acknowledgment.

NOTE

IoT Hub has SDK support for many device platforms and languages (including C, Java, Python, 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](#).

Prerequisites

- A complete working version of the [Send telemetry from a device to an IoT hub](#) quickstart or the [Configure message routing with IoT Hub](#) tutorial.
- [Java SE Development Kit 8](#). Make sure you select **Java 8** under **Long-term support** to get to downloads for JDK 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.
- Make sure that port 8883 is open in your firewall. The device sample in this article uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network

environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

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 an IoT hub](#) 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
```

Get the IoT hub connection string

In this article you create a backend service to send cloud-to-device messages through the IoT hub you created in [Send telemetry from a device to an IoT hub](#). To send cloud-to-device messages, your service needs the **service connect** permission. By default, every IoT Hub is created with a shared access policy named **service** that grants this permission.

To get the IoT Hub connection string for the **service** policy, follow these steps:

1. In the [Azure portal](#), select **Resource groups**. Select the resource group where your hub is located, and then select your hub from the list of resources.

2. On the left-side pane of your IoT hub, select **Shared access policies**.
3. From the list of policies, select the **service** policy.
4. Under **Shared access keys**, select the copy icon for the **Connection string -- primary key** and save the value.

The screenshot shows the Azure IoT Hub Shared access policies page for the 'contoso-hub-1' hub. The 'service' policy is selected. The 'Permissions' section includes 'Service connect' (checked) and 'Device connect' (unchecked). The 'Shared access keys' section shows two keys: 'Primary key' and 'Connection string—primary key'. The 'Connection string—primary key' is highlighted with a red box. Below it is the 'Secondary key' and its corresponding 'Connection string—secondary key', also with a red box around the copy icons.

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

For more information about IoT Hub shared access policies and permissions, see [Access control and permissions](#).

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 an IoT hub quickstart](#). You also need the the IoT hub connection string you copied previously in [Get the IoT hub connection string](#).

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, 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:

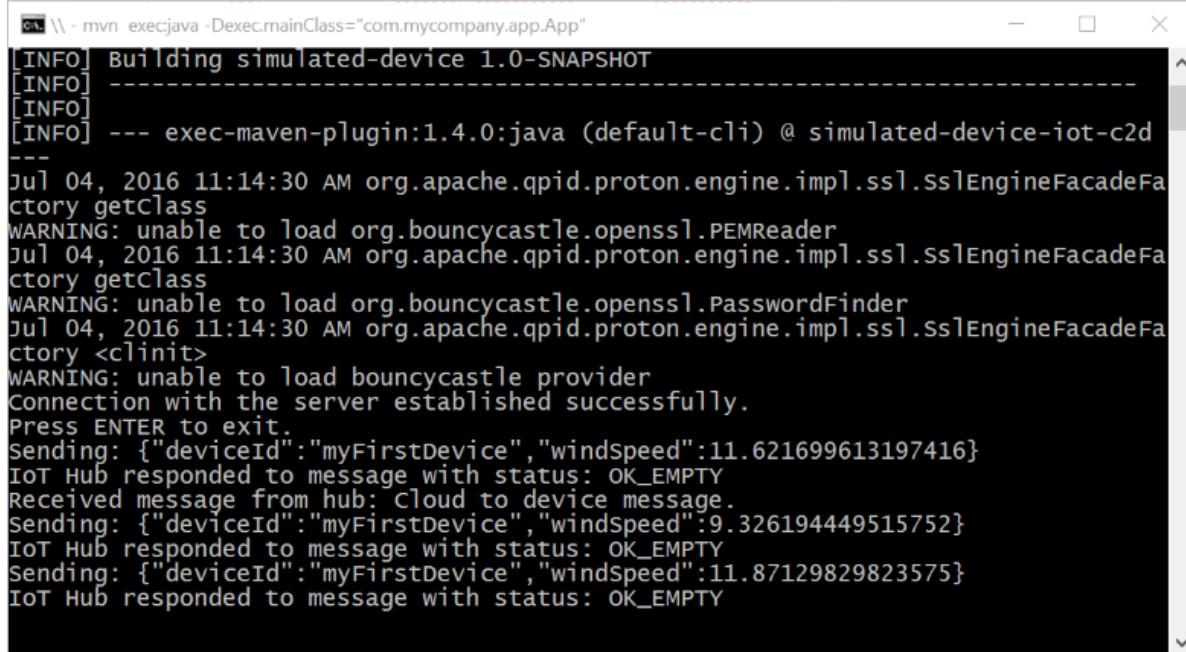
```
mvn clean package -DskipTests
```

Run the applications

You are now ready to run the applications.

- 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:

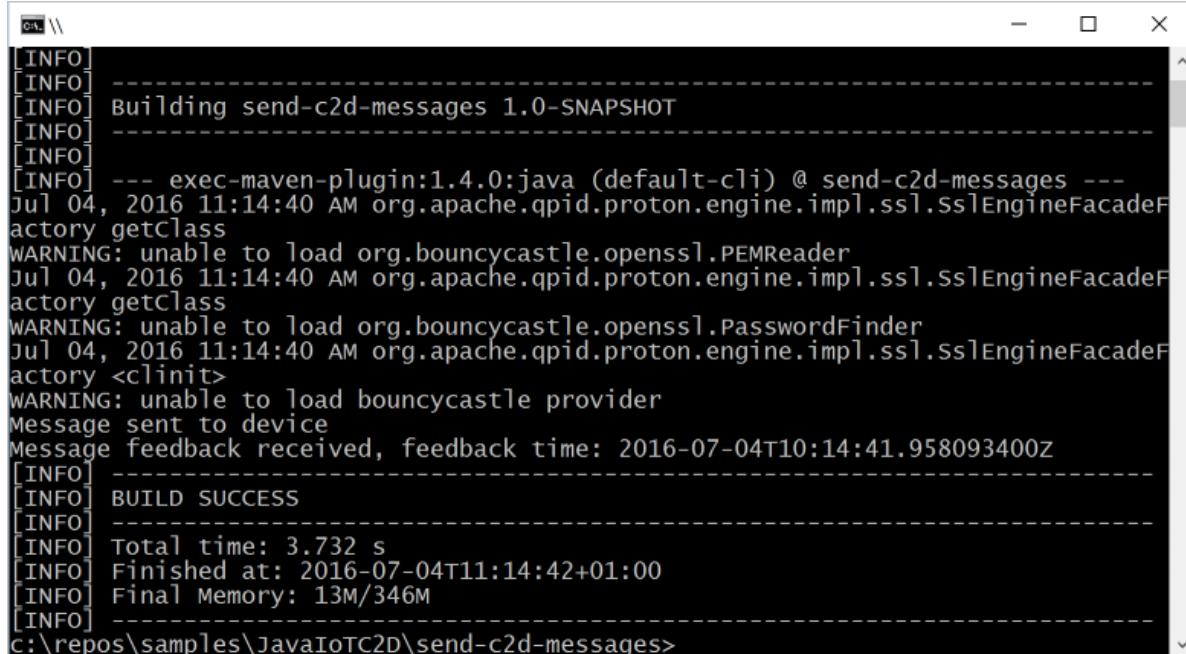
```
mvn exec:java -Dexec.mainClass="com.mycompany.app.App"
```



```
[INFO] Building simulated-device 1.0-SNAPSHOT
[INFO] -----
[INFO] --- exec-maven-plugin:1.4.0:java (default-cli) @ simulated-device-iot-c2d ---
[INFO] --- 
[INFO] Jul 04, 2016 11:14:30 AM org.apache.qpid.proton.engine.impl.ssl.SslEngineFacadeFactory getClass
[WARNING: unable to load org.bouncycastle.openssl.PEMReader
[INFO] Jul 04, 2016 11:14:30 AM org.apache.qpid.proton.engine.impl.ssl.SslEngineFacadeFactory getClass
[WARNING: unable to load org.bouncycastle.openssl.PasswordFinder
[INFO] Jul 04, 2016 11:14:30 AM org.apache.qpid.proton.engine.impl.ssl.SslEngineFacadeFactory <clinit>
[WARNING: unable to load bouncycastle provider
[INFO] Connection with the server established successfully.
[INFO] Press ENTER to exit.
[INFO] Sending: {"deviceId": "myFirstDevice", "windSpeed": 11.621699613197416}
[INFO] IoT Hub responded to message with status: OK_EMPTY
[INFO] Received message from hub: Cloud to device message.
[INFO] Sending: {"deviceId": "myFirstDevice", "windSpeed": 9.326194449515752}
[INFO] IoT Hub responded to message with status: OK_EMPTY
[INFO] Sending: {"deviceId": "myFirstDevice", "windSpeed": 11.87129829823575}
[INFO] IoT Hub responded to message with status: OK_EMPTY
```

- 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"
```



```
[INFO] 
[INFO] -----
[INFO] Building send-c2d-messages 1.0-SNAPSHOT
[INFO] -----
[INFO] --- exec-maven-plugin:1.4.0:java (default-cli) @ send-c2d-messages ---
[INFO] --- 
[INFO] Jul 04, 2016 11:14:40 AM org.apache.qpid.proton.engine.impl.ssl.SslEngineFacadeFactory getClass
[WARNING: unable to load org.bouncycastle.openssl.PEMReader
[INFO] Jul 04, 2016 11:14:40 AM org.apache.qpid.proton.engine.impl.ssl.SslEngineFacadeFactory getClass
[WARNING: unable to load org.bouncycastle.openssl.PasswordFinder
[INFO] Jul 04, 2016 11:14:40 AM org.apache.qpid.proton.engine.impl.ssl.SslEngineFacadeFactory <clinit>
[WARNING: unable to load bouncycastle provider
[INFO] Message sent to device
[INFO] Message feedback received, feedback time: 2016-07-04T10:14:41.958093400Z
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 3.732 s
[INFO] Finished at: 2016-07-04T11:14:42+01:00
[INFO] Final Memory: 13M/346M
[INFO] -----
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.js)

4/21/2020 • 6 minutes to read • [Edit Online](#)

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](#) quickstart 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 available only in the standard tier of IoT Hub. For more information about the basic and standard/free IoT Hub tiers, see [Choose the right IoT Hub tier](#).

This tutorial builds on [Send telemetry from a device to an 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 acknowledgment (*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 [Send telemetry from a device to an 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 acknowledgment.

NOTE

IoT Hub has SDK support for many device platforms and languages (including C, Java, Python, 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](#).

Prerequisites

- Node.js version 10.0.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.)
- Make sure that port 8883 is open in your firewall. The device sample in this article uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

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 an IoT hub](#) to receive cloud-to-device messages from the IoT hub.

1. Using a text editor, open the `SimulatedDevice.js` file. This file is located in the `iot-hub\Quickstarts\simulated-device` folder off of the root folder of the Node.js sample code you downloaded in the [Send telemetry from a device to an IoT hub](#) quickstart.
2. Register a handler with the device client to receive messages sent from IoT Hub. Add the call to `client.on` just after the line that creates the device client as in the following snippet:

```
var client = DeviceClient.fromConnectionString(connectionString, Mqtt);

client.on('message', function (msg) {
    console.log('Id: ' + msg.messageId + ' Body: ' + msg.data);
    client.complete(msg, function (err) {
        if (err) {
            console.error('complete error: ' + err.toString());
        } else {
            console.log('complete sent');
        }
    });
});
```

In this example, the device invokes the `complete` function to notify IoT Hub that it has processed the message. The call to `complete` is not required if you are using MQTT transport and can be omitted. It is required for HTTPS and AMQP.

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](#).

Get the IoT hub connection string

In this article, you create a backend service to send cloud-to-device messages through the IoT hub you created in [Send telemetry from a device to an IoT hub](#). To send cloud-to-device messages, your service needs the **service connect** permission. By default, every IoT Hub is created with a shared access policy named **service** that grants this permission.

To get the IoT Hub connection string for the **service** policy, follow these steps:

1. In the [Azure portal](#), select **Resource groups**. Select the resource group where your hub is located, and then select your hub from the list of resources.
2. On the left-side pane of your IoT hub, select **Shared access policies**.
3. From the list of policies, select the **service** policy.
4. Under **Shared access keys**, select the copy icon for the **Connection string -- primary key** and save the value.

The screenshot shows the 'Shared access policies' section of the Azure IoT Hub. A new policy named 'service' has been created, granting 'Service connect' permission. The primary and secondary connection strings are displayed.

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

For more information about IoT Hub shared access policies and permissions, see [Access control and permissions](#).

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 [Send telemetry from a device to an IoT hub](#) quickstart. You also need the IoT hub connection string you copied previously in [Get the IoT hub connection string](#).

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}" and "{device id}" placeholder values with the IoT hub connection string and device ID you noted previously:

```
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.

- At the command prompt in the **simulated-device** folder, run the following command to send telemetry to IoT Hub and to listen for cloud-to-device messages:

```
node SimulatedDevice.js
```

```
c:\ Command Prompt
Sending message: {"temperature":30.571832186866242,"humidity":74.78960791317392}
message sent
Sending message: {"temperature":26.395512140028316,"humidity":73.46879966130824}
message sent
Id: My Message ID Body: Cloud to device message.
complete sent
Sending message: {"temperature":28.806316985159192,"humidity":78.34255452684832}
message sent
Sending message: {"temperature":32.76608025397037,"humidity":63.430348050855535}
message sent
Sending message: {"temperature":20.58973782791757,"humidity":64.26617781160455}
message sent
Sending message: {"temperature":30.606218900428686,"humidity":70.57593844111695}
message sent
Sending message: {"temperature":28.483527184983096,"humidity":70.6114779074598}
message sent
Sending message: {"temperature":22.59206680932097,"humidity":64.17248829566937}
message sent
Sending message: {"temperature":26.87009185480146,"humidity":74.26954368191852}
message sent
Sending message: {"temperature":29.076348302755488,"humidity":66.80639697254125}
message sent
Sending message: {"temperature":33.30596531178733,"humidity":62.94948728532511}
message sent
Sending message: {"temperature":29.192259036616043,"humidity":67.84580308477204}
message sent
Sending message: {"temperature":21.75793794687162,"humidity":73.65655420261736}
message sent
```

- 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:\ \\ - 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, 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)

7/29/2020 • 6 minutes to read • [Edit Online](#)

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](#) quickstart 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 available only in the standard tier of IoT Hub. For more information about the basic and standard/free IoT Hub tiers, see [Choose the right IoT Hub tier](#).

This tutorial builds on [Send telemetry from a device to an 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.

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 [Send telemetry from a device to an IoT hub](#), which connects to your IoT hub and receives cloud-to-device messages.
- **SendCloudToDeviceMessage.py**, which sends cloud-to-device messages to the simulated device app through IoT Hub.

NOTE

IoT Hub has SDK support for many device platforms and languages (including C, Java, Javascript, and Python) through [Azure IoT device SDKs](#). For instructions on how to use Python to connect your device to this tutorial's code, and generally to Azure IoT Hub, see the [Azure IoT Python SDK](#).

Prerequisites

- An active Azure account. (If you don't have an account, you can create a [free account](#) in just a couple of minutes.)
- [Python version 3.7 or later](#) is recommended. 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. For other versions of Python supported, see [Azure IoT Device Features](#) in the SDK documentation. If you choose to use Python 2.7, you may need to [install or upgrade pip](#), the Python package management system.
- Make sure that port 8883 is open in your firewall. The device sample in this article uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

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. From a command prompt in your working directory, install the **Azure IoT Hub Device SDK for Python**:

```
pip install azure-iot-device
```

2. Using a text editor, create a file named **SimulatedDevice.py**.

3. Add the following `import` statements and variables at the start of the **SimulatedDevice.py** file:

```
import threading
import time
from azure.iot.device import IoTHubDeviceClient

RECEIVED_MESSAGES = 0
```

4. 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 [Send telemetry from a device to an IoT hub](#) quickstart:

```
CONNECTION_STRING = "{deviceConnectionString}"
```

5. Add the following function to print received messages to the console:

```
def message_listener(client):
    global RECEIVED_MESSAGES
    while True:
        message = client.receive_message()
        RECEIVED_MESSAGES += 1
        print("\nMessage received:")

        #print data and both system and application (custom) properties
        for property in vars(message).items():
            print ("    {0}".format(property))

        print( "Total calls received: {}".format(RECEIVED_MESSAGES))
        print()
```

6. Add the following code to initialize the client and wait to receive the cloud-to-device message:

```
def iothub_client_sample_run():
    try:
        client = IoTHubDeviceClient.create_from_connection_string(CONNECTION_STRING)

        message_listener_thread = threading.Thread(target=message_listener, args=(client,))
        message_listener_thread.daemon = True
        message_listener_thread.start()

        while True:
            time.sleep(1000)

    except KeyboardInterrupt:
        print ( "IoT Hub C2D Messaging device sample stopped" )
```

7. Add the following main function:

```

if __name__ == '__main__':
    print ( "Starting the Python IoT Hub C2D Messaging device sample..." )
    print ( "Waiting for C2D messages, press Ctrl-C to exit" )

    iothub_client_sample_run()

```

- Save and close the `SimulatedDevice.py` file.

Get the IoT hub connection string

In this article, you create a backend service to send cloud-to-device messages through the IoT hub you created in [Send telemetry from a device to an IoT hub](#). To send cloud-to-device messages, your service needs the **service connect** permission. By default, every IoT Hub is created with a shared access policy named **service** that grants this permission.

To get the IoT Hub connection string for the **service** policy, follow these steps:

- In the [Azure portal](#), select **Resource groups**. Select the resource group where your hub is located, and then select your hub from the list of resources.
- On the left-side pane of your IoT hub, select **Shared access policies**.
- From the list of policies, select the **service** policy.
- Under **Shared access keys**, select the copy icon for the **Connection string -- primary key** and save the value.

The screenshot shows the Azure portal interface for managing an IoT hub's shared access policies. On the left, the navigation pane includes options like Overview, Activity log, Access control (IAM), Tags, Events, Settings, Shared access policies (which is highlighted with a red box), Pricing and scale, IP Filter, Certificates, Built-in endpoints, Manual failover (preview), Properties, Locks, and Export template. The main content area displays the 'Shared access policies' page for the 'contoso-hub-1' IoT hub. It shows a table with two rows: 'iothubowner' and 'service'. The 'service' row has 'registry write, service connect' permissions. In the details pane on the right, under the 'service' policy, the 'Access policy name' is 'service', and the 'Permissions' section shows 'Registry read' (unchecked), 'Service connect' (checked), and 'Device connect' (unchecked). The 'Shared access keys' section contains fields for 'Primary key' and 'Secondary key', both of which have copy icons. The 'Connection string—primary key' field is also highlighted with a red box.

For more information about IoT Hub shared access policies and permissions, see [Access control and permissions](#).

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 [Send telemetry from a device to an IoT hub](#) quickstart. You also need the IoT hub connection string you copied previously in [Get the IoT hub connection string](#).

- In your working directory, open a command prompt and install the **Azure IoT Hub Service SDK for Python**.

```
pip install azure-iot-hub
```

2. Using a text editor, create a file named **SendCloudToDeviceMessage.py**.
3. Add the following `import` statements and variables at the start of the **SendCloudToDeviceMessage.py** file:

```
import random
import sys
from azure.iot.hub import IoTHubRegistryManager

MESSAGE_COUNT = 2
AVG_WIND_SPEED = 10.0
MSG_TXT = "{\"service client sent a message\": %.2f}"
```

4. Add the following code to **SendCloudToDeviceMessage.py** file. Replace the `{iot hub connection string}` and `{device id}` placeholder values with the IoT hub connection string and device ID you noted previously:

```
CONNECTION_STRING = "{IoTHubConnectionString}"
DEVICE_ID = "{deviceId}"
```

5. Add the following code to send messages to your device:

```
def iothub.messaging_sample_run():
    try:
        # Create IoTHubRegistryManager
        registry_manager = IoTHubRegistryManager(CONNECTION_STRING)

        for i in range(0, MESSAGE_COUNT):
            print ( 'Sending message: {0}'.format(i) )
            data = MSG_TXT % (AVG_WIND_SPEED + (random.random() * 4 + 2))

            props={}
            # optional: assign system properties
            props.update(messageId = "message_%d" % i)
            props.update(correlationId = "correlation_%d" % i)
            props.update(contentType = "application/json")

            # optional: assign application properties
            prop_text = "PropMsg_%d" % i
            props.update(testProperty = prop_text)

            registry_manager.send_c2d_message(DEVICE_ID, data, properties=props)

    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")

    except Exception as ex:
        print ( "Unexpected error {0}" % ex )
        return
    except KeyboardInterrupt:
        print ( "IoT Hub C2D Messaging service sample stopped" )
```

6. Add the following main function:

```
if __name__ == '__main__':
    print ( "Starting the Python IoT Hub C2D Messaging service sample..." )

    iothub.messaging_sample_run()
```

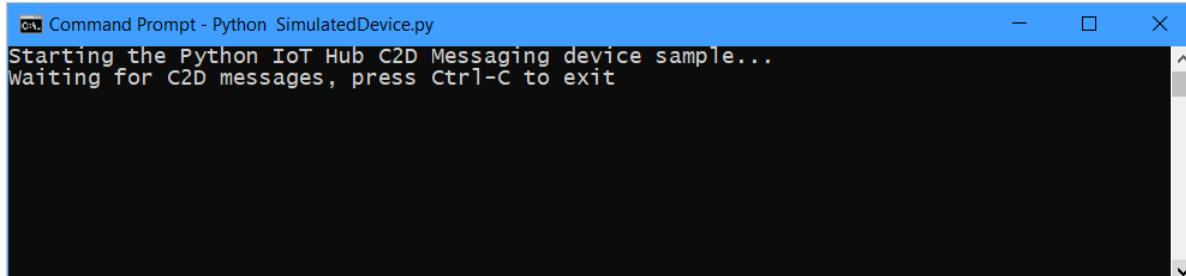
7. Save and close `SendCloudToDeviceMessage.py` file.

Run the applications

You are now ready to run the applications.

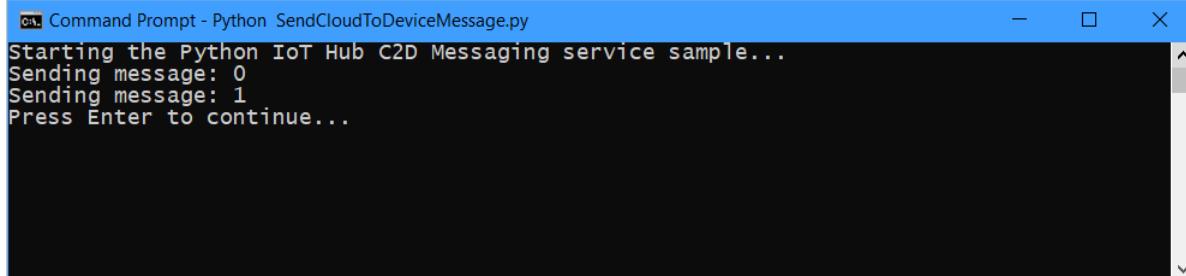
1. At the command prompt in your working directory, run the following command to listen for cloud-to-device messages:

```
python SimulatedDevice.py
```



2. Open a new command prompt in your working directory and run the following command to send cloud-to-device messages:

```
python SendCloudToDeviceMessage.py
```



3. Note the messages received by the device.

```
Command Prompt - Python SimulatedDevice.py
Starting the Python IoT Hub C2D Messaging device sample...
Waiting for C2D messages, press Ctrl-C to exit

Message received:
('data', b'{"service client sent a message": 13.00}')
('custom_properties', {'testProperty': 'PropMsg_0'})
('message_id', 'message_0')
('expiry_time_utc', None)
('correlation_id', 'correlation_0')
('user_id', None)
('content_encoding', None)
('content_type', 'application/json')
('output_name', None)
('_iothub_interface_id', None)
Total calls received: 1

Message received:
('data', b'{"service client sent a message": 14.11}')
('custom_properties', {'testProperty': 'PropMsg_1'})
('message_id', 'message_1')
('expiry_time_utc', None)
('correlation_id', 'correlation_1')
('user_id', None)
('content_encoding', None)
('content_type', 'application/json')
('output_name', None)
('_iothub_interface_id', None)
Total calls received: 2
```

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)

4/21/2020 • 6 minutes to read • [Edit Online](#)

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](#) quickstart 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 tutorial 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 acknowledgment (*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 acknowledgment.

NOTE

IoT Hub has SDK support for many device platforms and languages (including C, Java, Python, 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](#).

Prerequisites

- 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](#).
- Make sure that port 8883 is open in your firewall. The device sample in this article uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

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"
```

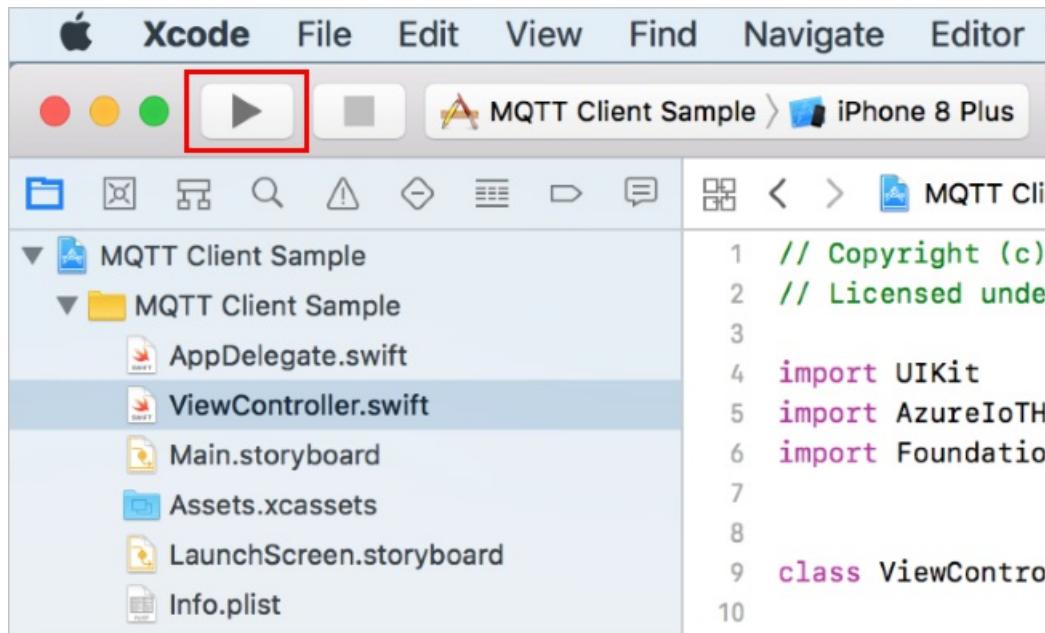
3. Expand the **MQTT Client Sample** project and then folder of the same name.

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**.



Get the IoT hub connection string

In this article you create a backend service to send cloud-to-device messages through the IoT hub you created in [Send telemetry from a device to an IoT hub](#). To send cloud-to-device messages, your service needs the **service connect** permission. By default, every IoT Hub is created with a shared access policy named **service** that grants this permission.

To get the IoT Hub connection string for the **service** policy, follow these steps:

1. In the [Azure portal](#), select **Resource groups**. Select the resource group where your hub is located, and then select your hub from the list of resources.
2. On the left-side pane of your IoT hub, select **Shared access policies**.
3. From the list of policies, select the **service** policy.
4. Under **Shared access keys**, select the copy icon for the **Connection string -- primary key** and save the value.

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

For more information about IoT Hub shared access policies and permissions, see [Access control and permissions](#).

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
```

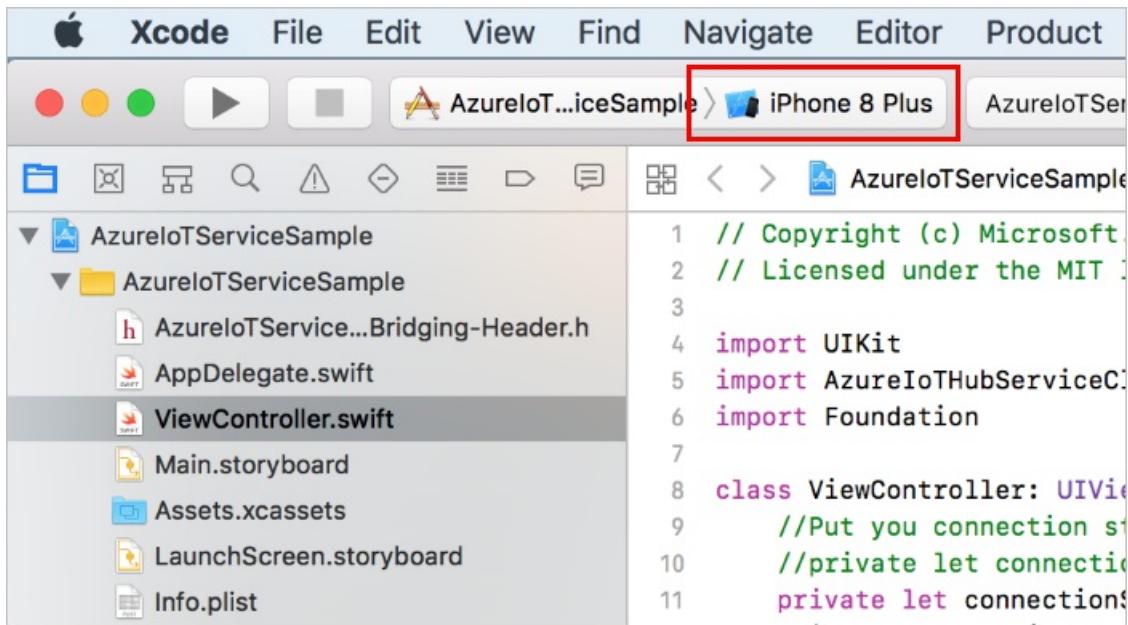
Along with installing the pods required for your project, the installation command also created an XCode workspace file that is already configured to use the pods for dependencies.

Run the sample service application

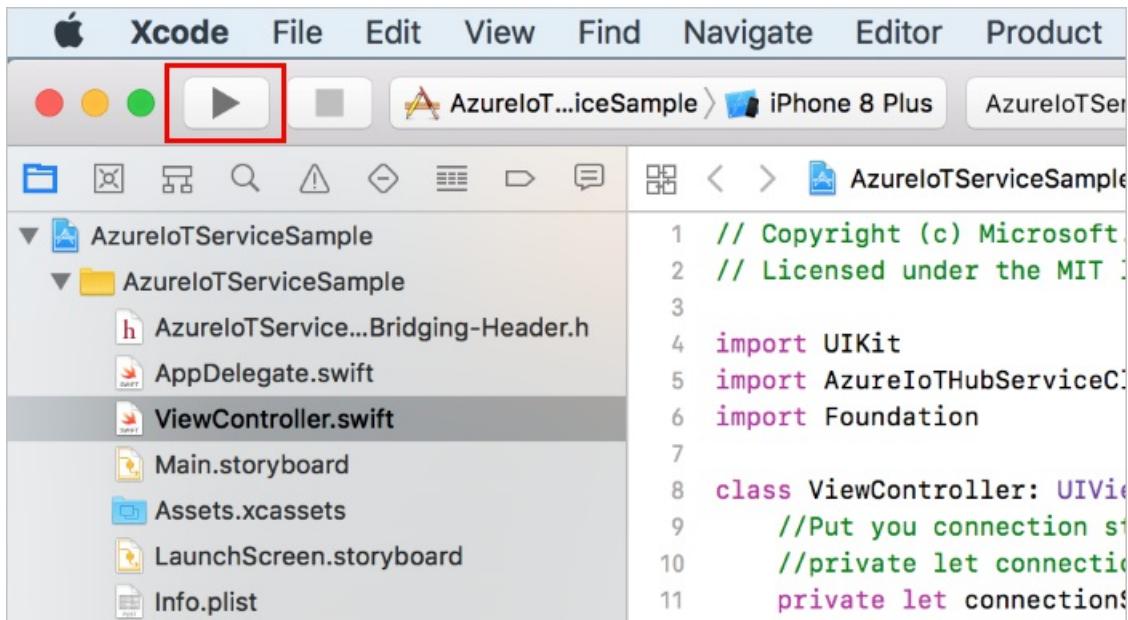
1. Open the sample workspace in XCode.

```
open AzureIoTServiceSample.xcworkspace
```

2. Expand the **AzureIoTServiceSample** 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 service connection string that you copied previously in [Get the IoT hub connection string](#).
5. Save your changes.
6. 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.



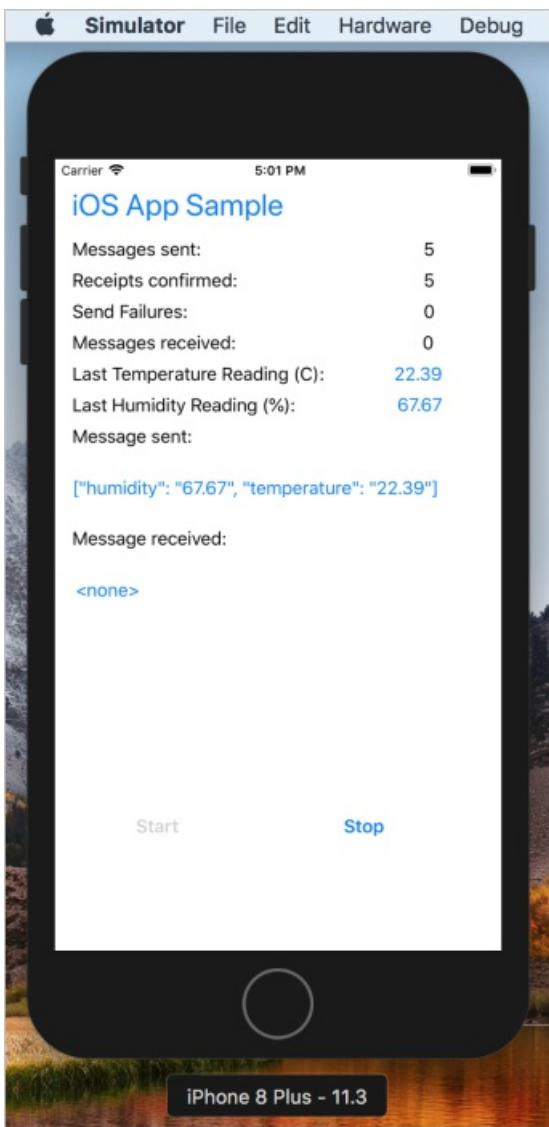
7. Run the project in the device emulator with the Build and run button or the key combo Command + r.



Send a cloud-to-device message

You are now ready to use the two applications to send and receive cloud-to-device messages.

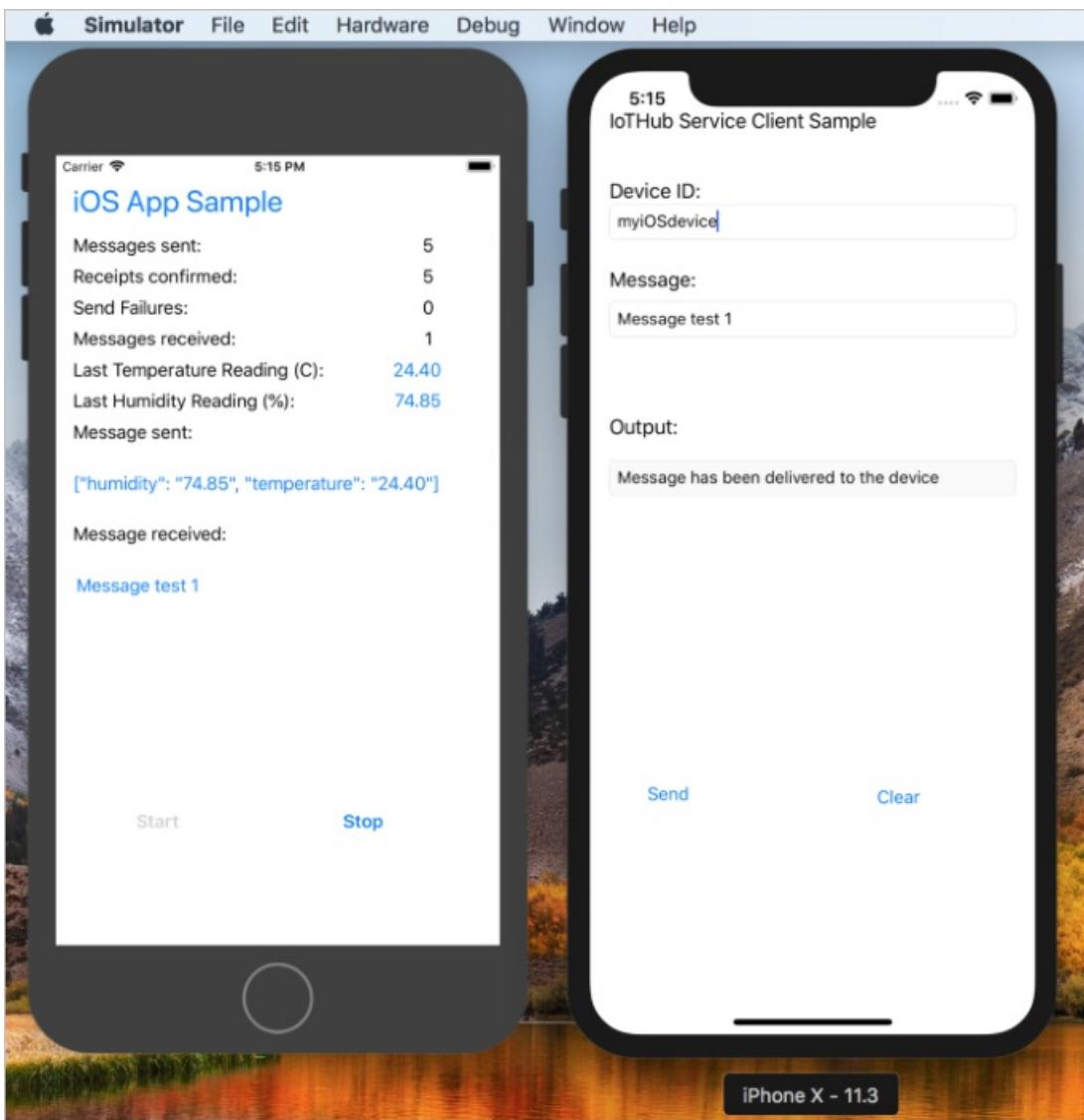
1. In the **iOS App Sample** app running on the simulated IoT device, click **Start**. The application starts sending device-to-cloud messages, but also starts listening for cloud-to-device messages.



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 (.NET)

4/21/2020 • 6 minutes to read • [Edit Online](#)

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](#) quickstart and [Send cloud-to-device messages with IoT Hub](#) tutorial 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 Microsoft Azure Blob storage. However, in some scenarios you can't 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**. This app uploads a file to storage using a SAS URI provided by your IoT hub. It is a modified version of the app created in the [Send cloud-to-device messages with IoT Hub](#) tutorial.
- **ReadFileUploadNotification**. This app receives file upload notifications from your IoT hub.

NOTE

IoT Hub supports many device platforms and languages, including C, Java, Python, 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.

Prerequisites

- Visual Studio
- An active Azure account. If you don't have an account, you can create a [free account](#) in just a couple of minutes.
- Make sure that port 8883 is open in your firewall. The device sample in this article uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

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.

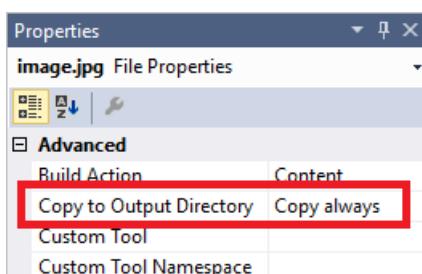
Follow the instructions in [Configure file uploads using the Azure portal](#). Make sure that a blob container is associated with your IoT hub and that file notifications are enabled.

The screenshot shows the Microsoft Azure portal interface. On the left, there's a sidebar with various service icons and a 'Create a resource' button. The main content area has a breadcrumb path: Home > All resources > iot-hub-contoso-one - File upload. The title is 'iot-hub-contoso-one - File upload'. Below the title, there's a 'Save' and 'Discard' button. The left navigation menu under 'Messaging' includes 'IoT device configuration', 'File upload' (which is selected and highlighted with a red box), 'Message routing', and other options like 'Security', 'Overview', 'Recommendations', etc. The right side of the screen shows configuration settings for 'File upload'. It includes sections for 'Storage container settings' (with 'Azure Storage Container' listed) and 'File notification settings' (with a toggle switch set to 'On'). There are also sliders for 'SAS TTL', 'Default TTL', and 'Maximum delivery count'. A note at the top says: 'Here you specify the storage container, file expiration, and retries for notifications when a file is uploaded.'

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 Solution Explorer, right-click the **SimulatedDevice** project, and select **Add > Existing Item**. Find an image file and include it in your project. This tutorial assumes the image is named `image.jpg`.
2. Right-click the image, and then select **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 line in the **Main** method, right before `Console.ReadLine()`:

```
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 [Transient fault handling](#).

Get the IoT hub connection string

In this article, you create a back-end service to receive file upload notification messages from the IoT hub you created in [Send telemetry from a device to an IoT hub](#). To receive file upload notification messages, your service needs the **service connect** permission. By default, every IoT Hub is created with a shared access policy named **service** that grants this permission.

To get the IoT Hub connection string for the **service** policy, follow these steps:

1. In the [Azure portal](#), select **Resource groups**. Select the resource group where your hub is located, and then select your hub from the list of resources.
2. On the left-side pane of your IoT hub, select **Shared access policies**.
3. From the list of policies, select the **service** policy.
4. Under **Shared access keys**, select the copy icon for the **Connection string -- primary key** and save the value.

The screenshot shows the 'Shared access policies' page for an IoT Hub named 'contoso-hub-1'. The left sidebar includes links for Overview, Activity log, Access control (IAM), Tags, Events, Settings, Shared access policies (which is selected and highlighted with a red box), Pricing and scale, IP Filter, Certificates, Built-in endpoints, Manual failover (preview), Properties, Locks, and Export template. The main area displays a table of shared access policies:

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

On the right, there's a 'service' configuration card with fields for Access policy name (set to 'service'), Permissions (Registry read, Registry write, Service connect checked, Device connect unchecked), and Shared access keys (Primary key and Secondary key both shown with copy icons). Below these are Connection string—primary key and Connection string—secondary key, each also with copy icons. A red box highlights the 'Connection string—primary key' field.

For more information about IoT Hub shared access policies and permissions, see [Access control and permissions](#).

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, select **File > New > Project**. In **Create a new project**, select **Console App (.NET Framework)**, and then select **Next**.
2. Name the project *ReadFileUploadNotification*. Under **Solution**, select **Add to solution**. Select **Create** to create the project.

The screenshot shows the 'Configure your new project' dialog. It has tabs for Console App (.NET Framework), C#, Windows, and Console. The 'Project name' field is set to 'ReadFileUploadNotification' (highlighted with a red box). The 'Location' dropdown is set to 'C:\Code\sample\iot-hub\Quickstarts'. The 'Solution' dropdown is set to 'Add to solution' (highlighted with a red box). The 'Solution name' dropdown is set to 'Quickstarts'. The 'Framework' dropdown is set to '.NET Framework 4.7.2'. At the bottom right, there are 'Back' and 'Create' buttons, with the 'Create' button highlighted with a red box.

3. In Solution Explorer, right-click the **ReadFileUploadNotification** project, and select **Manage NuGet Packages**.
4. In **NuGet Package Manager**, select **Browse**. Search for and select **Microsoft.Azure.Devices**, and then

select **Install**.

This step downloads, installs, and adds a reference to the [Azure IoT service SDK NuGet package](#) in the **ReadFileUploadNotification** project.

5. In the **Program.cs** file for this project, add the following statement at the top of the file:

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

6. Add the following fields to the **Program** class. Replace the `{iot hub connection string}` placeholder value with the IoT hub connection string that you copied previously in [Get the IoT hub connection string](#):

```
static ServiceClient serviceClient;
static string connectionString = "{iot hub connection string}";
```

7. 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.

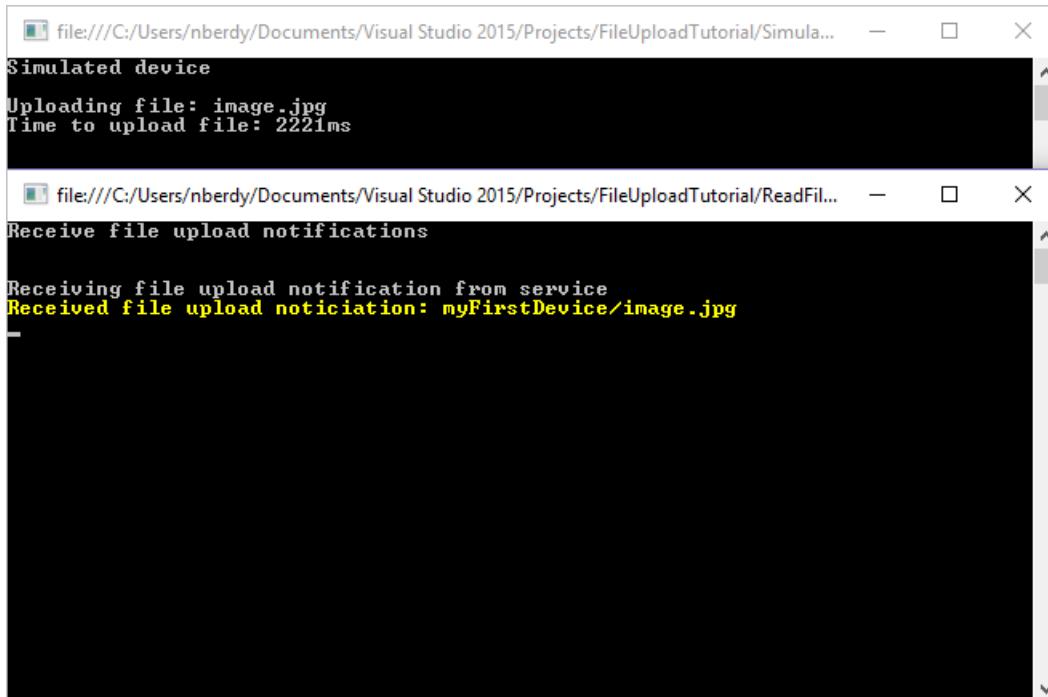
8. Finally, add the following lines to the **Main** method:

```
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.

1. In Solutions Explorer, right-click your solution, and select **Set StartUp Projects**.
2. In **Common Properties > Startup Project**, select **Multiple startup projects**, then select the **Start** action for **ReadFileUploadNotification** and **SimulatedDevice**. Select **OK** to save your changes.
3. 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.



The image shows two terminal windows side-by-side. The left window, titled 'Simulated device', displays the command 'Uploading file: image.jpg' and the time 'Time to upload file: 2221ms'. The right window, titled 'Receive file upload notifications', shows the message 'Receiving file upload notification from service' followed by 'Received file upload notification: myFirstDevice/image.jpg'.

```
file:///C:/Users/nberdy/Documents/Visual Studio 2015/Projects/FileUploadTutorial/Simula...
Simulated device
Uploading file: image.jpg
Time to upload file: 2221ms

file:///C:/Users/nberdy/Documents/Visual Studio 2015/Projects/FileUploadTutorial/ReadFil...
Receive file upload notifications

Receiving file upload notification from service
Received file upload notification: myFirstDevice/image.jpg
```

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 (Java)

7/29/2020 • 7 minutes to read • [Edit Online](#)

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 from a device to an IoT hub](#) quickstart and [Send cloud-to-device messages with IoT Hub](#) tutorial 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.

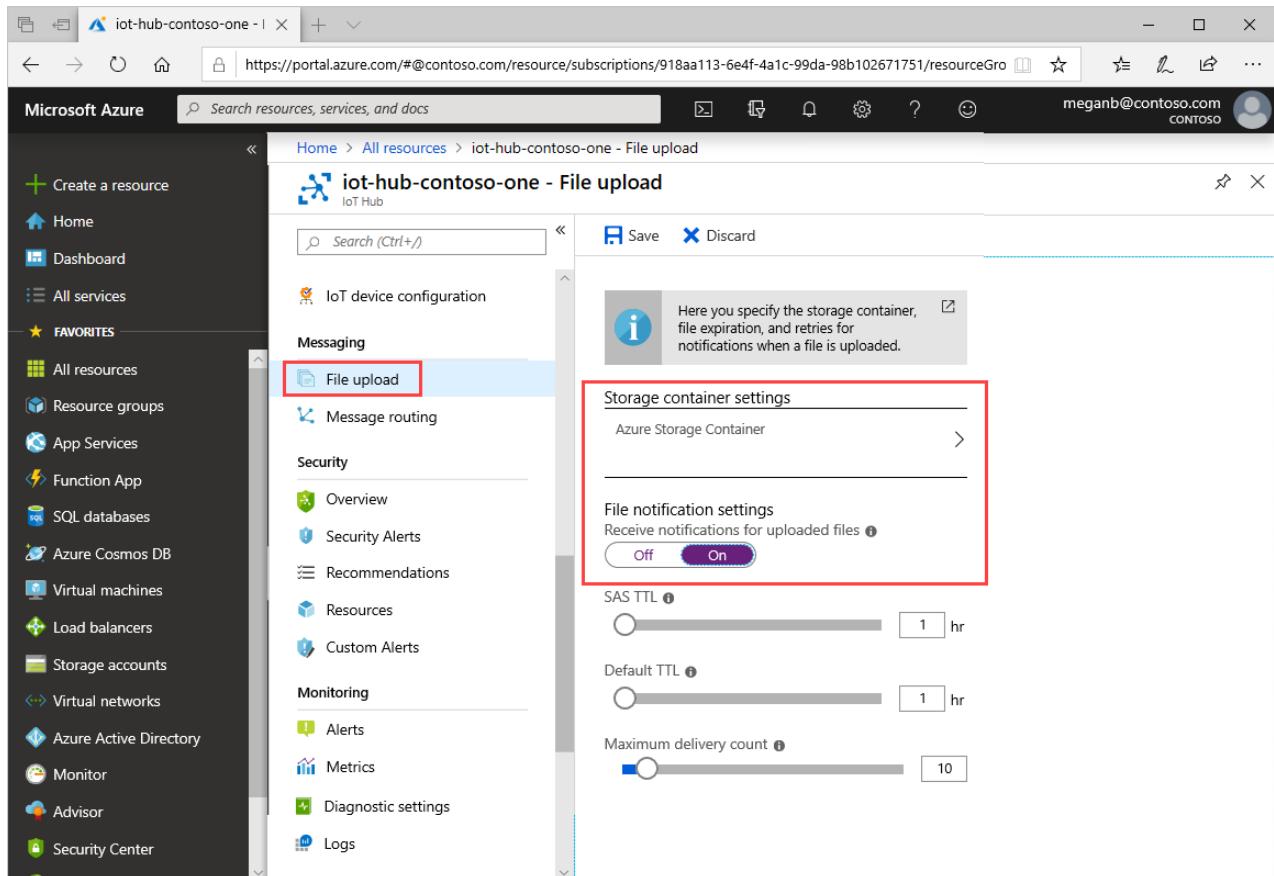
Prerequisites

- [Java SE Development Kit 8](#). Make sure you select **Java 8** under **Long-term support** to get to downloads for JDK 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.)
- Make sure that port 8883 is open in your firewall. The device sample in this article uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

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 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
```

Get the IoT hub connection string

In this article you create a backend service to receive file upload notification messages from the IoT hub you created in [Send telemetry from a device to an IoT hub](#). To receive file upload notification messages, your service needs the **service connect** permission. By default, every IoT Hub is created with a shared access policy named **service** that grants this permission.

To get the IoT Hub connection string for the **service** policy, follow these steps:

1. In the [Azure portal](#), select **Resource groups**. Select the resource group where your hub is located, and then select your hub from the list of resources.
2. On the left-side pane of your IoT hub, select **Shared access policies**.
3. From the list of policies, select the **service** policy.

- Under **Shared access keys**, select the copy icon for the **Connection string -- primary key** and save the value.

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

Shared access keys

- Primary key [Copy](#)
- Secondary key [Copy](#)
- Connection string—primary key [Copy](#) **Connection string—primary key**
- Connection string—secondary key [Copy](#)

For more information about IoT Hub shared access policies and permissions, see [Access control and permissions](#).

Receive a file upload notification

In this section, you create a Java console app that receives file upload notification messages from IoT Hub.

- 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
```

- At your command prompt, navigate to the new `read-file-upload-notification` folder.
- 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](#).

- Save and close the `pom.xml` file.
- Using a text editor, open the `read-file-upload-notification\src\main\java\com\mycompany\app\App.java` file.
- 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. Replace the `{Your IoT Hub connection string}` placeholder value with the IoT hub connection string that you copied previously in [Get the IoT hub connection string](#):

```
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: Name: myimage.png, Modified: 27/06/2017 03:22:38.

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 (Node.js)

4/21/2020 • 6 minutes to read • [Edit Online](#)

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 from a device to an 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. For example:

- Large files that contain images
- Videos
- Vibration data sampled at high frequency
- Some form of pre-processed 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.

Prerequisites

- Node.js version 10.0.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.)
- Make sure that port 8883 is open in your firewall. The device sample in this article uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

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 Microsoft Azure portal interface. The left sidebar lists various services like Home, Dashboard, All services, and Resource groups. The main content area is titled 'iot-hub-contoso-one - File upload' under 'IoT Hub'. On the left, a navigation menu includes 'File upload' (which is selected and highlighted in red), 'Message routing', 'Security', 'Monitoring', 'Resources', and 'Logs'. The right side contains several configuration sections. One section, 'Storage container settings', has a sub-section for 'Azure Storage Container' which is also highlighted with a red box. Another section, 'File notification settings', has a toggle switch set to 'On'. Below these 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 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`.

Get the IoT hub connection string

In this article you create a backend service to receive file upload notification messages from the IoT hub you created in [Send telemetry from a device to an IoT hub](#). To receive file upload notification messages, your service needs the **service connect** permission. By default, every IoT Hub is created with a shared access policy named **service** that grants this permission.

To get the IoT Hub connection string for the **service** policy, follow these steps:

1. In the [Azure portal](#), select **Resource groups**. Select the resource group where your hub is located, and then select your hub from the list of resources.
2. On the left-side pane of your IoT hub, select **Shared access policies**.
3. From the list of policies, select the **service** policy.
4. Under **Shared access keys**, select the copy icon for the **Connection string -- primary key** and save the value.

The screenshot shows the 'Shared access policies' blade for an IoT hub named 'contoso-hub-1'. The left sidebar includes links for Overview, Activity log, Access control (IAM), Tags, Events, Settings, and Shared access policies (which is selected and highlighted with a red box). The main area displays a table of shared access policies:

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

On the right, there are sections for 'Permissions' (with 'Service connect' checked) and 'Shared access keys' (showing Primary key, Secondary key, Connection string—primary key, and Connection string—secondary key, all with copy icons). A red box highlights the 'Connection string—primary key' field.

For more information about IoT Hub shared access policies and permissions, see [Access control and permissions](#).

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 the `{iothubconnectionstring}` placeholder value with the IoT hub connection string that you copied previously in [Get the IoT hub connection string](#):

```
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, under 'myiothubfileupload' containers, 'uploadcontainer' is selected. On the right, the 'myFirstDeviceAgain' folder is shown. It has an 'Upload' button and a search bar. The table lists one item: 'myimage.png' (modified 27/06/2017 03:22).

| 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 (Python)

7/29/2020 • 5 minutes to read • [Edit Online](#)

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 from a device to an 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.

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 has SDK support for many device platforms and languages (including C, Java, Javascript, and Python) through [Azure IoT device SDKs](#). For instructions on how to use Python to connect your device to this tutorial's code, and generally to Azure IoT Hub, see the [Azure IoT Python SDK](#).

Prerequisites

- An active Azure account. (If you don't have an account, you can create a [free account](#) in just a couple of minutes.)
- [Python version 3.7 or later](#) is recommended. 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. For other versions of Python supported, see [Azure IoT Device Features](#) in the SDK documentation.

IMPORTANT

Because the device code in this article uses the asynchronous API, you cannot use Python 2.7.

- Make sure that port 8883 is open in your firewall. The device sample in this article uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

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 portal interface for managing an IoT hub named 'iot-hub-contoso-one'. The left sidebar lists various service categories like Home, Dashboard, All services, and Favorites. The main content area is titled 'iot-hub-contoso-one - File upload' under the 'IoT Hub' category. The 'File upload' tab is active, indicated by a red box. Below it, other tabs include 'IoT device configuration', 'Messaging', 'Message routing', 'Security', 'Recommendations', 'Resources', 'Custom Alerts', 'Monitoring', 'Logs', and 'Alerts'. A large callout box points to the 'File upload' tab. On the right, there's a descriptive text about specifying storage container settings and file expiration/retries for notifications. A red box highlights the 'Storage container settings' section, which includes a link to 'Azure Storage Container'. Another red box highlights the 'File notification settings' section, which contains a toggle switch labeled 'On' for receiving notifications for uploaded files. Other settings shown include SAS TTL (1 hr), Default TTL (1 hr), and Maximum delivery count (10).

Upload a file from a device app

In this section, you create the device app to upload a file to IoT hub.

- At your command prompt, run the following command to install the `azure-iot-device` package. You use this package to coordinate the file upload with your IoT hub.

```
pip install azure-iot-device
```

- At your command prompt, run the following command to install the `azure.storage.blob` package. You use this package to perform the file upload.

```
pip install azure.storage.blob
```

- Create a test file that you'll upload to blob storage.
- Using a text editor, create a `FileUpload.py` file in your working folder.
- Add the following `import` statements and variables at the start of the `FileUpload.py` file.

```
import os
import asyncio
from azure.iot.device.aio import IoTHubDeviceClient
from azure.core.exceptions import AzureError
from azure.storage.blob import BlobClient

CONNECTION_STRING = "[Device Connection String]"
PATH_TO_FILE = r"[Full path to local file]"
```

6. In your file, replace `[Device Connection String]` with the connection string of your IoT hub device. Replace `[Full path to local file]` with the path to the test file that you created or any file on your device that you want to upload.

7. Create a function to upload the file to blob storage:

```
async def store_blob(blob_info, file_name):
    try:
        sas_url = "https://{}.{}/{}{}".format(
            blob_info["hostName"],
            blob_info["containerName"],
            blob_info["blobName"],
            blob_info["sasToken"]
        )

        print("\nUploading file: {} to Azure Storage as blob: {} in container {}\n".format(file_name,
blob_info["blobName"], blob_info["containerName"]))

        # Upload the specified file
        with BlobClient.from_blob_url(sas_url) as blob_client:
            with open(file_name, "rb") as f:
                result = blob_client.upload_blob(f, overwrite=True)
            return (True, result)

    except FileNotFoundError as ex:
        # catch file not found and add an HTTP status code to return in notification to IoT Hub
        ex.status_code = 404
        return (False, ex)

    except AzureError as ex:
        # catch Azure errors that might result from the upload operation
        return (False, ex)
```

This function parses the `blob_info` structure passed into it to create a URL that it uses to initialize an [azure.storage.blob.BlobClient](#). Then it uploads your file to Azure blob storage using this client.

8. Add the following code to connect the client and upload the file:

```

async def main():
    try:
        print ( "IoT Hub file upload sample, press Ctrl-C to exit" )

        conn_str = CONNECTION_STRING
        file_name = PATH_TO_FILE
        blob_name = os.path.basename(file_name)

        device_client = IoTHubDeviceClient.create_from_connection_string(conn_str)

        # Connect the client
        await device_client.connect()

        # Get the storage info for the blob
        storage_info = await device_client.get_storage_info_for_blob(blob_name)

        # Upload to blob
        success, result = await store_blob(storage_info, file_name)

        if success == True:
            print("Upload succeeded. Result is: \n")
            print(result)
            print()

            await device_client.notify_blob_upload_status(
                storage_info["correlationId"], True, 200, "OK: {}".format(file_name)
            )

        else :
            # If the upload was not successful, the result is the exception object
            print("Upload failed. Exception is: \n")
            print(result)
            print()

            await device_client.notify_blob_upload_status(
                storage_info["correlationId"], False, result.status_code, str(result)
            )

    except Exception as ex:
        print("\nException:")
        print(ex)

    except KeyboardInterrupt:
        print ( "\nIoTHubDeviceClient sample stopped" )

    finally:
        # Finally, disconnect the client
        await device_client.disconnect()

if __name__ == "__main__":
    asyncio.run(main())
    #loop = asyncio.get_event_loop()
    #loop.run_until_complete(main())
    #loop.close()

```

This code creates an asynchronous **IoTHubDeviceClient** and uses the following APIs to manage the file upload with your IoT hub:

- **get_storage_info_for_blob** gets information from your IoT hub about the linked Storage Account you created previously. This information includes the hostname, container name, blob name, and a SAS token. The storage info is passed to the **store_blob** function (created in the previous step), so the **BlobClient** in that function can authenticate with Azure storage. The **get_storage_info_for_blob** method also returns a correlation_id, which is used in the

`notify_blob_upload_status` method. The `correlation_id` is IoT Hub's way of marking which blob you're working on.

- `notify_blob_upload_status` notifies IoT Hub of the status of your blob storage operation. You pass it the `correlation_id` obtained by the `get_storage_info_for_blob` method. It's used by IoT Hub to notify any service that might be listening for a notification on the status of the file upload task.

9. Save and close the `UploadFile.py` file.

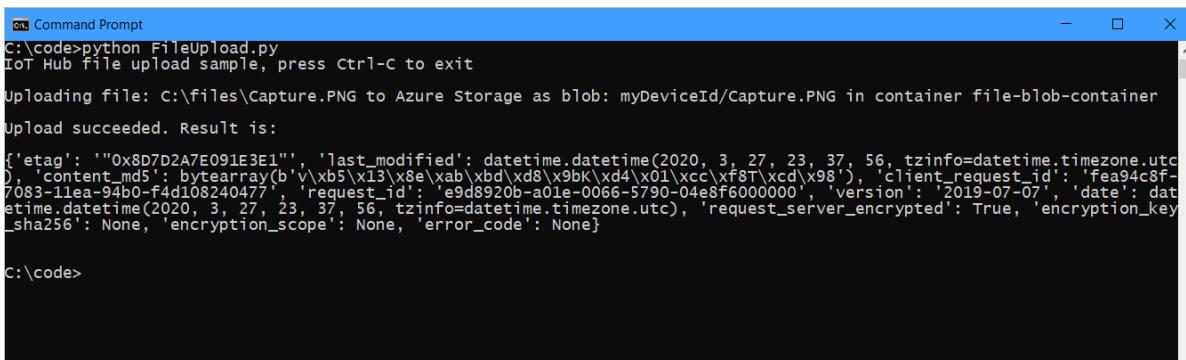
Run the application

Now you're 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:



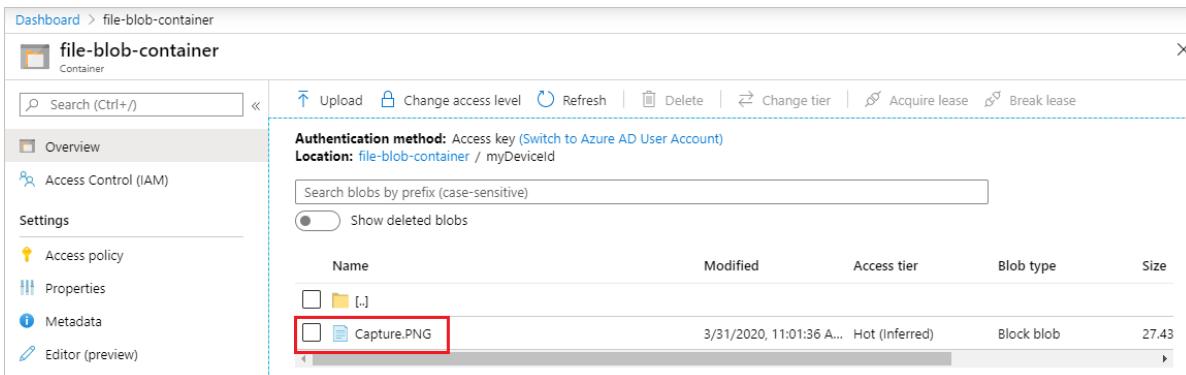
```
C:\Code>python FileUpload.py
IoT Hub file upload sample, press Ctrl-C to exit

Uploading file: C:\files\Capture.PNG to Azure Storage as blob: myDeviceId/Capture.PNG in container file-blob-container
Upload succeeded. Result is:

{'etag': "'0x8D2A7E091E3E1'", 'last_modified': datetime.datetime(2020, 3, 27, 23, 37, 56, tzinfo=datetime.timezone.utc), 'content_md5': b'\xb5\x13\x8e\xab\xbd\xd8\x9b\xd4\x01\xcc\xf8\xcd\x98'}, 'client_request_id': 'fea94c8f-7083-11ea-94b0-f4d108240477', 'request_id': 'e9d8920b-a01e-0066-5790-04e8f6000000', 'version': '2019-07-07', 'date': datetime.datetime(2020, 3, 27, 23, 37, 56, tzinfo=datetime.timezone.utc), 'request_server_encrypted': True, 'encryption_key_sha256': None, 'encryption_scope': None, 'error_code': None}

C:\code>
```

3. You can use the portal to view the uploaded file in the storage container you configured:



| Name | Modified | Access tier | Blob type | Size |
|-------------|--------------------------|----------------|------------|-------|
| Capture.PNG | 3/31/2020, 11:01:36 A... | Hot (Inferred) | Block blob | 27.43 |

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](#)

Learn more about Azure Blob Storage with the following links:

- [Azure Blob Storage documentation](#)
- [Azure Blob Storage for Python API documentation](#)

Get started with device twins (Node.js)

4/21/2020 • 11 minutes to read • [Edit Online](#)

Device twins are JSON documents that store device state information, including 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 available only in the standard tier of IoT Hub. For more information about the basic and standard/free IoT Hub tiers, see [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.

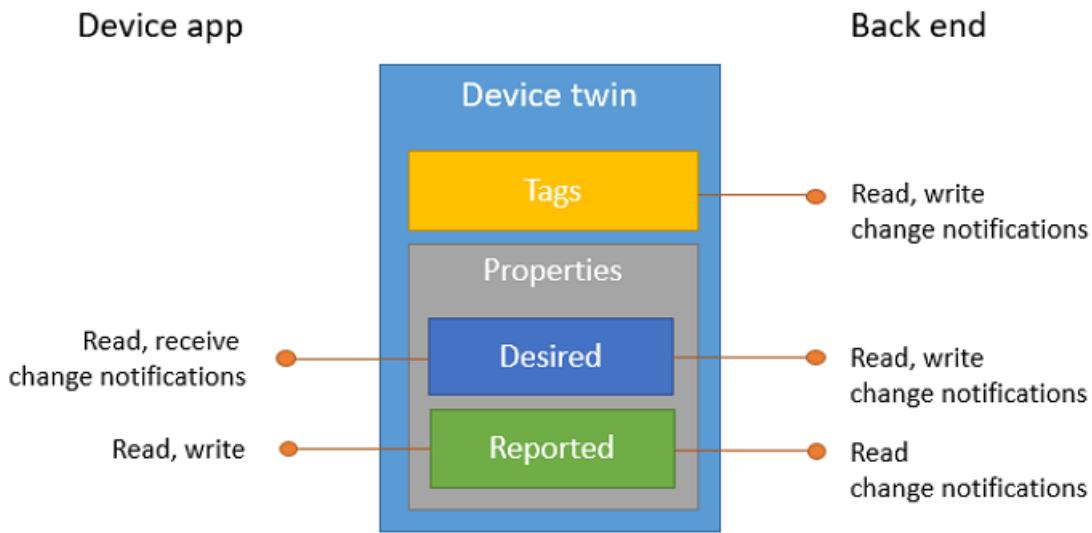
Device twins are designed for synchronization and for querying device configurations and conditions. More information on when to use device twins can be found in [Understand device twins](#).

Device twins are stored in an IoT hub and contain the following elements:

- **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.
- **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.

The following illustration shows device twin organization:



Additionally, the solution back end can query device twins based on all the above data. For more information about device twins, see [Understand device twins](#). For more information about querying, see [IoT Hub query language](#).

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.

Prerequisites

To complete this tutorial, you need:

- Node.js version 10.0.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.)
- Make sure that port 8883 is open in your firewall. The device sample in this article uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

Create an IoT hub

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

1. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.
 - **IoT Hub Name:** Enter a name for your hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

The screenshot shows the 'Basics' tab of the IoT hub creation wizard. A red box highlights the 'Subscription', 'Resource group', 'Region', and 'IoT hub name' fields. At the bottom, a red box highlights the 'Next: Size and scale >' button.

Home > New > IoT hub

IoT hub

Microsoft

Basics [Size and scale](#) [Tags](#) [Review + create](#)

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

Project details

Choose the subscription you'll use to manage deployments and costs. Use resource groups like folders to help you organize and manage resources.

Subscription * [Personal IoT items](#)

Resource group * [Create new](#)

Region * [East Asia](#)

IoT hub name * [Once your hub is created, this name can't be changed](#)

Review + create [< Previous](#) [Next: Size and scale >](#) [Automation options](#)

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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 [Determines how your IoT hub can scale. You can change this later if your needs increase.](#)

Azure Security Center [On](#) Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.
- If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure](#)

resources.

The screenshot shows the 'Tags' section of the IoT hub configuration. It displays a table with columns for Name, Value, and Resource. There are two entries: 'department' with value 'accounting' under 'IoT Hub', and an empty row for another IoT Hub entry. Navigation buttons at the bottom include 'Review + create', '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

| Name | Value | Resource |
|------------|------------|----------|
| department | accounting | IoT Hub |
| | | IoT Hub |

Review + create < Previous: Size and scale Next: Review + create > Automation options

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' page for the IoT hub. It lists configuration details under three sections: Basics, Size and scale, and Tags. The 'Review + create' tab is highlighted with a red box. The 'Create' button at the bottom left is also highlighted with a red box. Navigation buttons at the bottom include '< Previous: Tags', 'Next >', and 'Automation options'.

Basics

| | |
|----------------|------------------|
| Subscription | Personal testing |
| Resource group | iot-hubs |
| Region | West US 2 |
| IoT hub name | you-hub-name |

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 |
| Azure Security Center | 0.001 USD per device per month |

Tags

| | |
|------------|------------|
| department | accounting |
|------------|------------|

Create < Previous: Tags Next > Automation options

8. Select **Create** to create your new hub. Creating the hub takes a few minutes.

Register a new device in the IoT hub

In this section, you use the Azure CLI to create a device identity for this article. Device IDs are case sensitive.

1. Open [Azure Cloud Shell](#).
2. In Azure Cloud Shell, run the following command to install the Microsoft Azure IoT Extension for Azure CLI:

```
az extension add --name azure-iot
```

3. 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.

Get the IoT hub connection string

In this article, you create a back-end service that adds desired properties to a device twin and then queries the identity registry to find all devices with reported properties that have been updated accordingly. Your service needs the **service connect** permission to modify desired properties of a device twin, and it needs the **registry read** permission to query the identity registry. There is no default shared access policy that contains only these two permissions, so you need to create one.

To create a shared access policy that grants **service connect** and **registry read** permissions and get a connection string for this policy, follow these steps:

1. In the [Azure portal](#), select **Resource groups**. Select the resource group where your hub is located, and then select your hub from the list of resources.
2. On the left-side pane of your hub, select **Shared access policies**.
3. From the top menu above the list of policies, select **Add**.
4. Under **Add a shared access policy**, enter a descriptive name for your policy, such as `serviceAndRegistryRead`. Under **Permissions**, select **Registry read** and **Service connect**, and then select **Create**.

The screenshot shows the 'Shared access policies' page for an IoT hub. A new policy named 'serviceAndRegistryRead' is being created. The 'Permissions' section is highlighted with a red box, showing 'Registry read' checked. The 'Create' button is also highlighted with a red box.

5. Select your new policy from the list of policies.
6. Under **Shared access keys**, select the copy icon for the **Connection string -- primary key** and save the value.

The screenshot shows the 'Shared access policies' page with the 'serviceAndRegistryRead' policy selected. The 'Shared access keys' section is highlighted with a red box, showing the 'Primary key' and 'Connection string—primary key' fields. The 'Connection string—primary key' field has a copy icon next to it.

For more information about IoT Hub shared access policies and permissions, see [Access control and permissions](#).

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. The `--yes` parameter accepts all the defaults.

```
npm init --yes
```

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. Replace **{iot hub connection string}** with the IoT Hub connection string you copied in [Get the IoT hub connection string](#).

```
'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.

When the code creates the **query** object, it specifies the maximum number of returned documents in the second parameter. 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, the results of aggregation queries.

6. 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.

1. 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. The **--yes** parameter accepts all the defaults.

```
npm init --yes
```

- At your command prompt in the **reportconnectivity** folder, run the following command to install the **azure-iot-device**, and **azure-iot-device-mqtt** packages:

```
npm install azure-iot-device azure-iot-device-mqtt --save
```

- Using a text editor, create a new **ReportConnectivity.js** file in the **reportconnectivity** folder.

- Add the following code to the **ReportConnectivity.js** file. Replace `{device connection string}` with the device connection string you copied when you created the **myDeviceId** device identity in [Register a new device in the IoT hub](#).

```
'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.

- Run the device app

```
node ReportConnectivity.js
```

You should see the message `twin state reported`.

- 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)

7/29/2020 • 12 minutes to read • [Edit Online](#)

Device twins are JSON documents that store device state information, including 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 available only in the standard tier of IoT Hub. For more information about the basic and standard/free IoT Hub tiers, see [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.

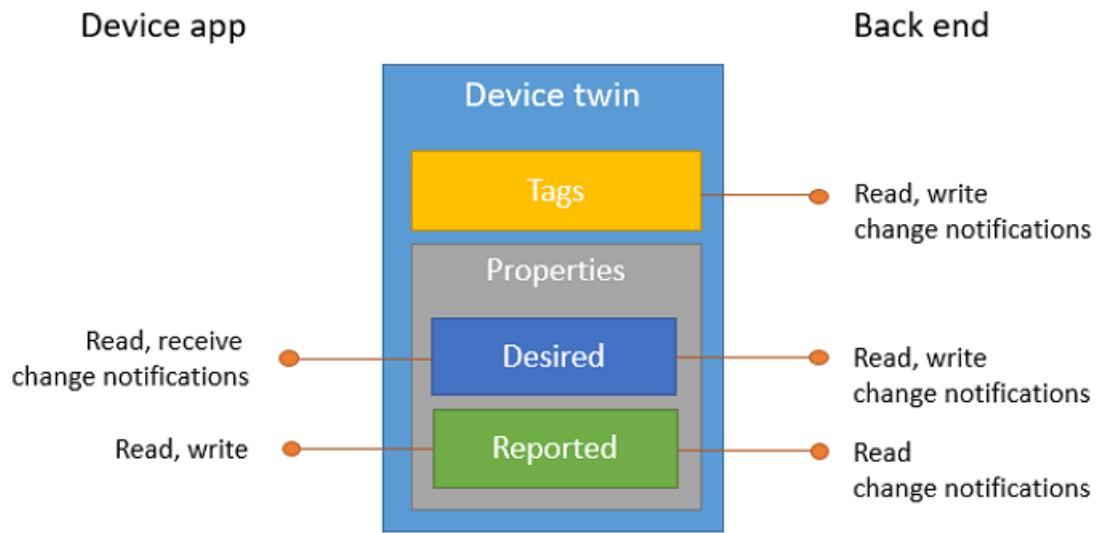
Device twins are designed for synchronization and for querying device configurations and conditions. More information on when to use device twins can be found in [Understand device twins](#).

Device twins are stored in an IoT hub and contain the following elements:

- **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.
- **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.

The following illustration shows device twin organization:



Additionally, the solution back end can query device twins based on all the above data. For more information about device twins, see [Understand device twins](#). For more information about querying, see [IoT Hub query language](#).

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 these .NET console apps:

- **AddTagsAndQuery**. This back-end app adds tags and queries device twins.
- **ReportConnectivity**. This device app 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.

Prerequisites

- Visual Studio.
- An active Azure account. If you don't have an account, you can create a [free account](#) in just a couple of minutes.
- Make sure that port 8883 is open in your firewall. The device sample in this article uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

Create an IoT hub

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

1. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.
 - **IoT Hub Name:** Enter a name for your hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

The screenshot shows the 'IoT hub' creation page in the Azure portal. The top navigation bar includes 'Home > New > IoT hub'. The main title is 'IoT hub' by Microsoft. Below the title, there are tabs for 'Basics', 'Size and scale', 'Tags', and 'Review + create'. The 'Basics' tab is selected. A descriptive text states: '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 asks to choose a subscription and resource group. It shows 'Subscription' set to 'Personal IoT items', 'Resource group' as a dropdown with 'Create new' option, 'Region' set to 'East Asia', and 'IoT hub name' as a dropdown with placeholder 'Once your hub is created, this name can't be changed'. A red box highlights the 'Basics' form fields. At the bottom, there are buttons for 'Review + create', '< Previous', 'Next: Size and scale >' (which is also highlighted with a red box), and 'Automation options'.

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics **Size and scale** Tags 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 [Determines how your IoT hub can scale. You can change this later if your needs increase.](#)

Azure Security Center [On](#) Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.
If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure](#)

resources.

The screenshot shows the 'Tags' tab of the IoT hub configuration page. It displays a table with one row of data:

| Name | Value | Resource | Actions |
|------------|------------|----------|-----------------------------|
| department | accounting | IoT Hub | ... Delete |
| | | IoT Hub | |

Below the table are navigation buttons: 'Review + create' (highlighted in blue), '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' page of the IoT hub creation process. The configuration details are as follows:

| Setting | Value |
|----------------------------|--------------------------------|
| Subscription | Personal testing |
| Resource group | iot-hubs |
| Region | West US 2 |
| IoT hub name | you-hub-name |
| 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 |
| Azure Security Center | 0.001 USD per device per month |
| Tags | |
| department | accounting |

At the bottom are buttons: 'Create' (highlighted in red), '< Previous: Tags', 'Next >', and 'Automation options'.

8. Select **Create** to create your new hub. Creating the hub takes a few minutes.

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 a hub

unless it has an entry in the identity registry. For more information, see the [IoT Hub developer guide](#).

1. In your IoT hub navigation menu, open **IoT Devices**, then select **New** to add a device in your IoT hub.

The screenshot shows the Azure IoT Hub management interface for the 'iot-hub-contoso-one' hub. The left sidebar contains a navigation menu with items such as Home, All resources, Overview, Activity log, Access control (IAM), Tags, Events, Settings (Shared access policies, Pricing and scale, IP Filter, Certificates, Built-in endpoints, Manual failover (preview), Properties, Locks, Export template), Explorers (Query explorer, IoT devices - highlighted with a red box), and Automatic Device Management. The main content area is titled 'iot-hub-contoso-one - IoT devices'. It features a search bar, a 'New' button (also highlighted with a red box), Refresh, and Delete buttons. A query editor allows for filtering devices based on properties, operators, and values. A table below lists columns for DEVICE ID, STATUS, LAST ACTIVITY TIME (UTC), LAST STATUS UPDATE (UTC), AUTHENTICATION T..., and CLOUD The message 'No results' is displayed at the bottom of the table area.

2. In **Create a device**, provide a name for your new device, such as **myDeviceId**, and select **Save**. This action creates a device identity for your IoT hub.

Home > All resources > **iot-hub-contoso-one - IoT devices** > Create a device

Create a device

Find Certified for Azure IoT devices in the Device Catalog

*** Device ID** myDeviceId

Authentication type Symmetric key

*** Primary key** Enter your primary key

*** Secondary key** Enter your secondary key

Auto-generate keys

Connect this device to an IoT hub Enable

Parent device 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 **Primary Connection String** to use later.

Home > **iot-hub-contoso-one - IoT devices** > myDeviceId

myDeviceId
iot-hub-contoso-one

| | | |
|------------------------------|---|---------------------------------|
| Device ID | myDeviceId | <input type="button" value=""/> |
| Primary Key | H2Awv1PN3suNBkaiQU1UeEINB3j0= | <input type="button" value=""/> |
| Secondary Key | G7615rzcbq/WFzcfTlgmad55/GVa4I= | <input type="button" value=""/> |
| Primary Connection String | HostName=iot-hub-contoso-one.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=QdSim8l7cptUCeMYGVSeiRKOV2ZGFSJpbmykIVYM9df= | <input type="button" value=""/> |
| Secondary Connection String | HostName=iot-hub-contoso-one.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=q32joixuiffXbbqKYkjv8sF82qZnqzGZspkl2nqz= | <input type="button" value=""/> |
| Enable connection to IoT Hub | <input checked="" type="radio"/> Enable <input type="radio"/> Disable | <input type="button" value=""/> |
| Parent device | No parent device | <input type="button" value=""/> |

Module Identities **Configurations**

MODULE ID CONNECTION STATE CONNECTION STATE LAST UPDATED (UTC) LAST ACTIVITY TIME (UTC)

There are no module identities for this 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](#).

Get the IoT hub connection string

In this article, you create a back-end service that adds desired properties to a device twin and then queries the identity registry to find all devices with reported properties that have been updated accordingly. Your service needs the **service connect** permission to modify desired properties of a device twin, and it needs the **registry read** permission to query the identity registry. There is no default shared access policy that contains only these two permissions, so you need to create one.

To create a shared access policy that grants **service connect** and **registry read** permissions and get a connection string for this policy, follow these steps:

1. In the [Azure portal](#), select **Resource groups**. Select the resource group where your hub is located, and then select your hub from the list of resources.
2. On the left-side pane of your hub, select **Shared access policies**.
3. From the top menu above the list of policies, select **Add**.
4. Under **Add a shared access policy**, enter a descriptive name for your policy, such as *serviceAndRegistryRead*. Under **Permissions**, select **Registry read** and **Service connect**, and then select **Create**.

The screenshot shows the Azure portal interface for creating a new Shared Access Policy. On the left, the navigation bar is visible with 'contoso-hub-1 - Shared access policies' selected. The main area displays a list of existing policies: 'iothubowner' (permissions: registry write, service connect), 'service' (permissions: service connect), 'device' (permissions: device connect), 'registryRead' (permissions: registry read), and 'registryReadWrite' (permissions: registry write). A red box highlights the '+ Add' button. To the right, a modal window titled 'Add a shared access policy' is open. It shows the 'Access policy name' field set to 'serviceAndRegistryRead'. A red box highlights the 'Permissions' section, which contains two checked checkboxes: 'Registry read' and 'Service connect'. Another red box highlights the 'Create' button at the bottom right of the modal.

5. Select your new policy from the list of policies.
6. Under **Shared access keys**, select the copy icon for the **Connection string -- primary key** and save the value.

The screenshot shows the Azure IoT Hub Shared access policies page for the 'contoso-hub-1' hub. The left sidebar lists various settings like Overview, Activity log, Access control (IAM), Tags, Events, Properties, Locks, and Export template. The 'Shared access policies' option is selected. On the right, a modal window titled 'serviceAndRegistryRead' is open, showing the configuration for this policy. The 'Permissions' section includes 'Registry read' (checked) and 'Service connect' (checked). The 'Shared access keys' section shows the primary key and secondary key, both with copy and delete icons. A red box highlights the 'Connection string—primary key' field, which contains a long connection string starting with 'Endpoint'. Below it is another connection string for the secondary key.

For more information about IoT Hub shared access policies and permissions, see [Access control and permissions](#).

Create the service app

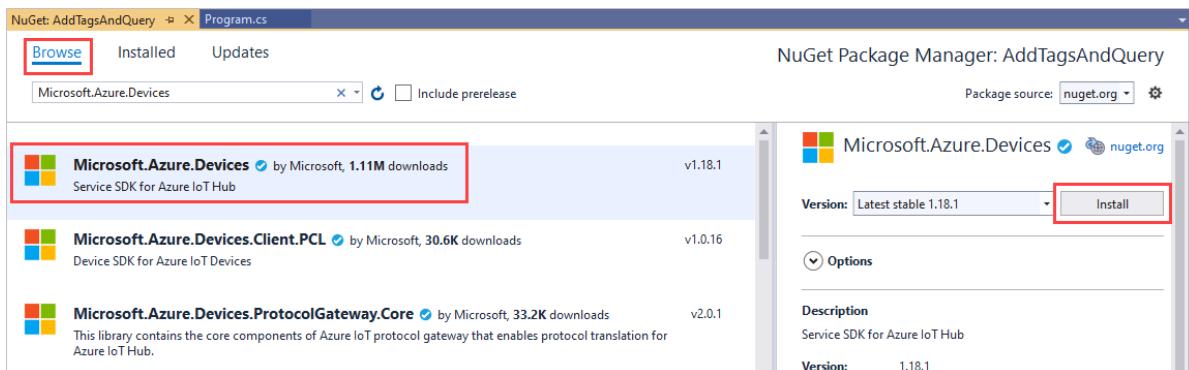
In this section, you create a .NET console app, using C#, that adds location metadata to the device twin associated with `myDeviceId`. It then queries the device twins stored in the IoT hub selecting the devices located in the US, and then the ones that reported a cellular connection.

1. In Visual Studio, select **Create a new project**. In **Create new project**, select **Console App (.NET Framework)**, and then select **Next**.
2. In **Configure your new project**, name the project **AddTagsAndQuery**.

The screenshot shows the 'Configure your new project' dialog for a 'Console App (.NET Framework)' in C#. The 'Project name' field is set to 'AddTagsAndQuery' and is highlighted with a red box. The 'Location' field shows the path 'C:\Code\IoTHubProjects'. The 'Solution name' field is also highlighted with a red box and contains 'AddTagsAndQuery'. The 'Place solution and project in the same directory' checkbox is unchecked. The 'Framework' dropdown is set to '.NET Framework 4.7.2'. At the bottom right, there are 'Back' and 'Create' buttons, with the 'Create' button highlighted with a red box.

3. In Solution Explorer, right-click the **AddTagsAndQuery** project, and then select **Manage NuGet Packages**.

4. Select **Browse** and search for and select **Microsoft.Azure.Devices**. Select **Install**.



This step downloads, installs, and adds a reference to the [Azure IoT service SDK](#) NuGet package and its dependencies.

5. Add the following `using` statements at the top of the **Program.cs** file:

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

6. Add the following fields to the **Program** class. Replace `{iot hub connection string}` with the IoT Hub connection string that you copied in [Get the IoT hub connection string](#).

```
static RegistryManager registryManager;
static string connectionString = "{iot hub connection string}";
```

7. 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.

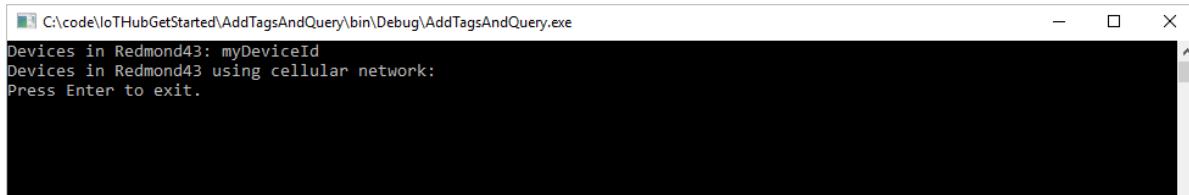
After updating, it executes two queries: the first selects only the device twins of devices located in the **Redmond43** plant, and the second refines the query to select only the devices that are also connected through cellular network.

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();
```

- 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.

- In Visual Studio, select **File > New > Project**. In **Create new project**, choose **Console App (.NET Framework)**, and then select **Next**.
- In **Configure your new project**, name the project **ReportConnectivity**. For **Solution**, choose **Add to solution**, and then select **Create**.
- In Solution Explorer, right-click the **ReportConnectivity** project, and then select **Manage NuGet Packages**.
- Select **Browse** and search for and choose **Microsoft.Azure.Devices.Client**. Select **Install**.

This step downloads, installs, and adds a reference to the [Azure IoT device SDK](#) NuGet package and its dependencies.

- 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;
```

- Add the following fields to the **Program** class. Replace `{device connection string}` with the device connection string that you noted in [Register a new device in the IoT hub](#).

```
static string DeviceConnectionString = "HostName=<yourIoTHubName>.azure-devices.net;DeviceId=<yourIoTDeviceName>;SharedAccessKey=<yourIoTDeviceAccessKey>";
static DeviceClient Client = null;
```

7. 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**.

8. 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 the reported property of **myDeviceId** with the connectivity information.

9. 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();

```

10. In Solution Explorer, right-click on your solution, and select **Set StartUp Projects**.
11. In **Common Properties > Startup Project**, select **Multiple startup projects**. For **ReportConnectivity**, select **Start** as the **Action**. Select **OK** to save your changes.
12. Run this app by right-clicking the **ReportConnectivity** project and selecting **Debug**, then **Start new instance**. You should see the app getting the twin information, and then sending connectivity as a *reported property*.

C:\code\IoTHubGetStarted\ReportConnectivity\bin\Debug\ReportConnectivity.exe
Connecting to hub
Retrieving twin
Sending connectivity data as reported property
Press Enter to exit.

After the device reported its connectivity information, it should appear in both queries.

13. Right-click the **AddTagsAndQuery** project and select **Debug > Start new instance** to run the queries again. This time, **myDeviceId** should appear in both query results.

C:\code\IoTHubGetStarted\AddTagsAndQuery\bin\Debug\AddTagsAndQuery.exe
Devices in Redmond43: myDeviceId
Devices in Redmond43 using cellular network: myDeviceId
Press Enter to exit.

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.

You can learn more from the following resources:

- To learn how to send telemetry from devices, see the [Send telemetry from a device to an IoT hub](#) tutorial.
- To learn how to configure devices using device twin's desired properties, see the [Use desired properties to configure devices](#) tutorial.
- To learn how to control devices interactively, such as turning on a fan from a user-controlled app, see the [Use direct methods](#) tutorial.

Get started with device twins (Java)

7/29/2020 • 14 minutes to read • [Edit Online](#)

Device twins are JSON documents that store device state information, including 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 available only in the standard tier of IoT Hub. For more information about the basic and standard/free IoT Hub tiers, see [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.

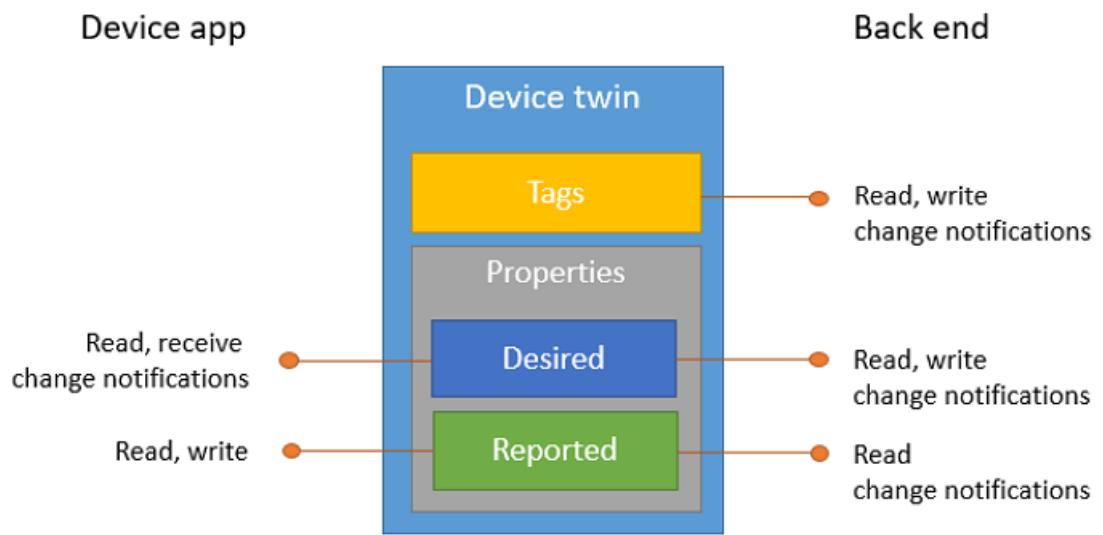
Device twins are designed for synchronization and for querying device configurations and conditions. More information on when to use device twins can be found in [Understand device twins](#).

Device twins are stored in an IoT hub and contain the following elements:

- **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.
- **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.

The following illustration shows device twin organization:



Additionally, the solution back end can query device twins based on all the above data. For more information about device twins, see [Understand device twins](#). For more information about querying, see [IoT Hub query language](#).

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.

Prerequisites

- [Java SE Development Kit 8](#). Make sure you select **Java 8** under **Long-term support** to get to downloads for JDK 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.)
- Make sure that port 8883 is open in your firewall. The device sample in this article uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

Create an IoT hub

This section describes how to create an IoT hub using the Azure portal.

1. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.
 - **IoT Hub Name:** Enter a name for your hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

The screenshot shows the 'Create IoT hub' wizard in the Azure portal. The title bar says 'Home > New > IoT hub'. The main heading is 'IoT hub' by Microsoft. Below it, there are tabs: 'Basics' (which is underlined), 'Size and scale', 'Tags', and 'Review + create'. A sub-section titled 'Project details' asks to choose a subscription and resource group. The 'Subscription' dropdown is set to 'Personal IoT items'. The 'Resource group' dropdown has a red border around it, and the 'Create new' button is visible below it. The 'Region' dropdown is set to 'East Asia'. The 'IoT hub name' input field contains 'Once your hub is created, this name can't be changed'. At the bottom, there are buttons for 'Review + create' (blue), '< Previous' (disabled), 'Next: Size and scale >' (highlighted with a red border), and 'Automation options'.

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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
 Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center On
 Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.

If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure](#)

resources.

The screenshot shows the 'Tags' section of the IoT hub creation wizard. It displays a table with one row of data:

| Name | Value | Resource |
|------------|------------|----------|
| department | accounting | IoT Hub |
| | | IoT Hub |

Below the table are navigation buttons: 'Review + create' (highlighted in blue), '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' page for the IoT hub. It lists the following configuration details:

| Category | Setting | Value |
|----------------|----------------------------|--------------------------------|
| Basics | Subscription | Personal testing |
| | Resource group | iot-hubs |
| | Region | West US 2 |
| | IoT hub name | you-hub-name |
| | | |
| 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 |
| | Azure Security Center | 0.001 USD per device per month |
| | | |
| Tags | department | accounting |

At the bottom are buttons: 'Create' (highlighted in red), '< Previous: Tags', 'Next >', and 'Automation options'.

8. Select **Create** to create your new hub. Creating the hub takes a few minutes.

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 a hub

unless it has an entry in the identity registry. For more information, see the [IoT Hub developer guide](#).

1. In your IoT hub navigation menu, open **IoT Devices**, then select **New** to add a device in your IoT hub.

The screenshot shows the Azure IoT Hub management interface for the 'iot-hub-contoso-one' hub. The left sidebar contains a navigation menu with items such as Home, All resources, Overview, Activity log, Access control (IAM), Tags, Events, Settings (with Shared access policies, Pricing and scale, IP Filter, Certificates, Built-in endpoints, Manual failover (preview), Properties, Locks, and Export template), Explorers (Query explorer), and Automatic Device Management (which is also highlighted with a red box). The main content area is titled 'iot-hub-contoso-one - IoT devices'. It features a search bar at the top with a '+ New' button (highlighted with a red box) and a 'Refresh' and 'Delete' button. Below the search bar is a query editor with fields for 'Field' (select or enter a property name), 'Operator' (=), and 'Value' (specify constraint value). A link 'Switch to query editor' is also present. The main table below the query editor shows columns: DEVICE ID, STATUS, LAST ACTIVITY TIME (UTC), LAST STATUS UPDATE (UTC), AUTHENTICATION T..., and CLOUD The table displays 'No results'.

2. In **Create a device**, provide a name for your new device, such as **myDeviceId**, and select **Save**. This action creates a device identity for your IoT hub.

Home > All resources > iot-hub-contoso-one - IoT devices > Create a device

Create a device

Find Certified for Azure IoT devices in the Device Catalog

*** Device ID** myDeviceId

Authentication type: Symmetric key

*** Primary key** Enter your primary key

*** Secondary key** Enter your secondary key

Auto-generate keys:

Connect this device to an IoT hub: Enable

Parent device: 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 **Primary Connection String** to use later.

Home > iot-hub-contoso-one - IoT devices > myDeviceId

myDeviceId
iot-hub-contoso-one

Save Message to Device Direct Method Add Module Identity Device Twin Manage keys Refresh

| | | |
|------------------------------|---|--|
| Device ID | myDeviceId | |
| Primary Key | H2Awv1PN3suIBkaiQU1UeEINB3j0= | |
| Secondary Key | G7615zcboqWFzcfTlgmad55/GVa4I= | |
| Primary Connection String | HostName=iot-hub-contoso-one.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=QdSim6i7cptUCeMYGVSeIRKOV2ZGFSJpbmykIVYM9df= | |
| Secondary Connection String | HostName=iot-hub-contoso-one.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=q32oiXuwffXbbqKYkjy8sF82qZInqzGZspqkl2nqz= | |
| Enable connection to IoT Hub | <input checked="" type="radio"/> Enable <input type="radio"/> Disable | |
| Parent device | No parent device | |

Module Identities Configurations

MODULE ID CONNECTION STATE CONNECTION STATE LAST UPDATED (UTC) LAST ACTIVITY TIME (UTC)

There are no module identities for this 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](#).

Get the IoT hub connection string

In this article, you create a back-end service that adds desired properties to a device twin and then queries the identity registry to find all devices with reported properties that have been updated accordingly. Your service needs the **service connect** permission to modify desired properties of a device twin, and it needs the **registry read** permission to query the identity registry. There is no default shared access policy that contains only these two permissions, so you need to create one.

To create a shared access policy that grants **service connect** and **registry read** permissions and get a connection string for this policy, follow these steps:

1. In the [Azure portal](#), select **Resource groups**. Select the resource group where your hub is located, and then select your hub from the list of resources.
2. On the left-side pane of your hub, select **Shared access policies**.
3. From the top menu above the list of policies, select **Add**.
4. Under **Add a shared access policy**, enter a descriptive name for your policy, such as *serviceAndRegistryRead*. Under **Permissions**, select **Registry read** and **Service connect**, and then select **Create**.

The screenshot shows the Azure portal interface for managing an IoT hub. On the left, the navigation menu includes options like Overview, Activity log, Access control (IAM), Tags, Events, Pricing and scale, IP Filter, Certificates, Built-in endpoints, Manual failover (preview), Properties, Locks, and Export template. The 'Shared access policies' option is highlighted with a red box. On the main page, there's a table listing existing policies: iothubowner (permissions: registry write, service connect), service (permissions: service connect), device (permissions: device connect), registryRead (permissions: registry read), and registryReadWrite (permissions: registry write). A modal window titled 'Add a shared access policy' is open, showing a form to create a new policy. The 'Access policy name' field contains 'serviceAndRegistryRead'. The 'Permissions' section is highlighted with a red box, showing checkboxes for 'Registry read' (checked), 'Registry write' (unchecked), 'Service connect' (checked), and 'Device connect' (unchecked). A large red box highlights the 'Create' button at the bottom right of the modal.

5. Select your new policy from the list of policies.
6. Under **Shared access keys**, select the copy icon for the **Connection string -- primary key** and save the value.

The screenshot shows the Azure IoT Hub Shared access policies page. A new policy named 'serviceAndRegistryRead' is being created. The 'Permissions' section includes 'Registry read' (checked) and 'Service connect' (checked). The 'Connection string—primary key' field is highlighted with a red box.

For more information about IoT Hub shared access policies and permissions, see [Access control and permissions](#).

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 named `iot-java-twin-getstarted`.
2. In the `iot-java-twin-getstarted` folder, create a Maven project named `add-tags-query` using the following command at your command prompt:

```
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>1.17.1</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 the IoT hub connection string you copied in [Get the IoT hub connection string](#).

```
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. Replace the code in the **main** method with the following code 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 named `simulated-device` using the following command at your command prompt:

```
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>1.17.5</version>
</dependency>
```

NOTE

You can check for the latest version of `iot-device-client` using [Maven search](#).

4. Add the following dependency to the `dependencies` node. This dependency configures a NOP for the Apache [SLF4J](#) logging facade, which is used by the device client SDK to implement logging. This configuration is optional, but, if you omit it, you may see a warning in the console when you run the app. For more information about logging in the device client SDK, see [Logging in the Samples for the Azure IoT device SDK for Java](#) readme file.

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-nop</artifactId>
  <version>1.7.28</version>
</dependency>
```

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 `simulated-device\src\main\java\com\mycompany\app\App.java` file.
8. 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;
```

9. Add the following class-level variables to the `App` class. Replace `{yourdeviceconnectionstring}` with the device connection string you copied in [Register a new device in the IoT hub](#).

```
private static String connString = "{yourdeviceconnectionstring}";
private static IoTHttpClientProtocol protocol = IoTHttpClientProtocol.MQTT;
private static String deviceId = "myDeviceId";
```

This sample app uses the `protocol` variable when it instantiates a `DeviceClient` object.

10. 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());
    }
}

```

11. Replace the code in the **main** method with the following code 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);
    }
};

```

12. 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...");
}

```

13. 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();

```

14. Modify the signature of the **main** method to include the exceptions as follows:

```

public static void main(String[] args) throws URISyntaxException, IOException

```

15. Save and close the **simulated-device\src\main\java\com\mycompany\app\App.java** file.
16. Build the **simulated-device** app and correct any errors. At your command prompt, navigate to the **simulated-device** folder and run the following command:

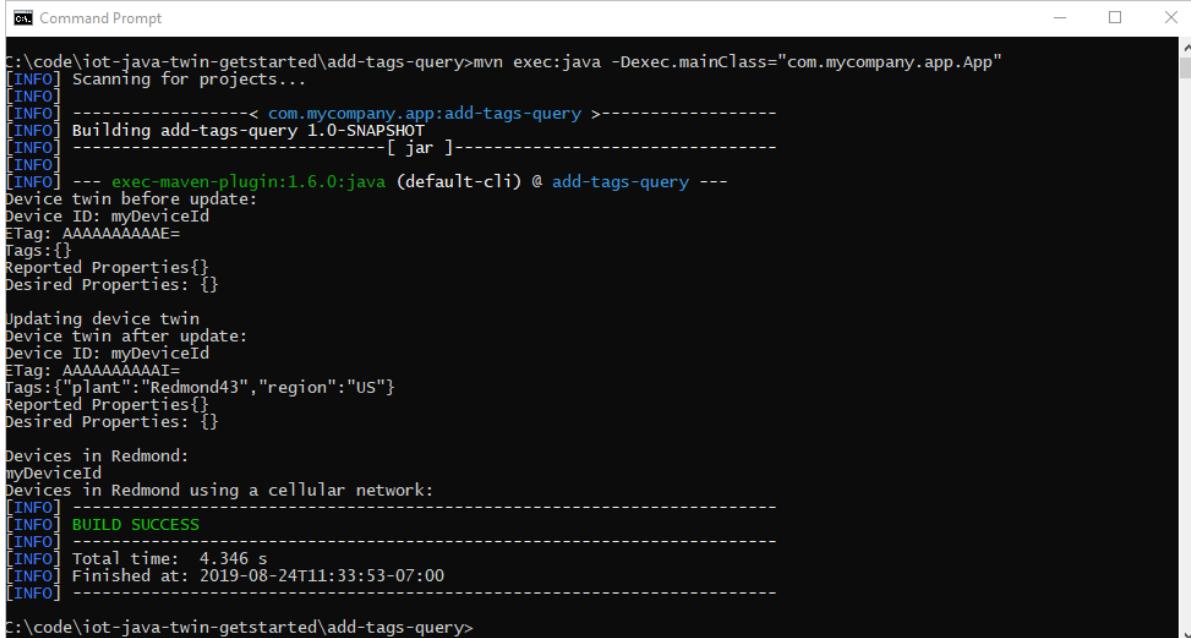
```
mvn clean package -DskipTests
```

Run the apps

You are now ready to run the console apps.

1. 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"
```



```
C:\code\iot-java-twin-getstarted\add-tags-query>mvn exec:java -Dexec.mainClass="com.mycompany.app.App"
[INFO] Scanning for projects...
[INFO] [INFO] -----< com.mycompany.app:add-tags-query >-----
[INFO] Building add-tags-query 1.0-SNAPSHOT
[INFO] [INFO] [jar]
[INFO] [INFO] --- exec-maven-plugin:1.6.0:java (default-cli) @ add-tags-query ---
Device twin before update:
Device ID: myDeviceId
ETag: AAAAAAAAEE=
Tags: {}
Reported Properties={}
Desired Properties: {}

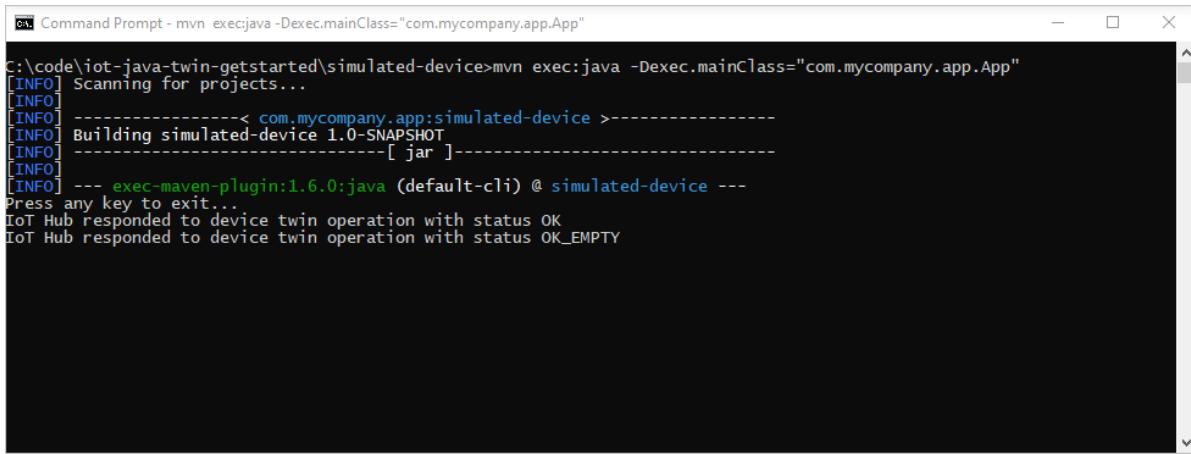
Updating device twin
Device twin after update:
Device ID: myDeviceId
ETag: AAAAAAAAII=
Tags: {"plant": "Redmond43", "region": "US"}
Reported Properties={}
Desired Properties: {}

Devices in Redmond:
myDeviceId
Devices in Redmond using a cellular network:
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 4.346 s
[INFO] Finished at: 2019-08-24T11:33:53-07:00
[INFO] -----
```

You can see the **plant** and **region** tags added to the device twin. The first query returns your device, but the second does not.

2. At a command prompt in the **simulated-device** folder, run the following command to add the **connectivityType** reported property to the device twin:

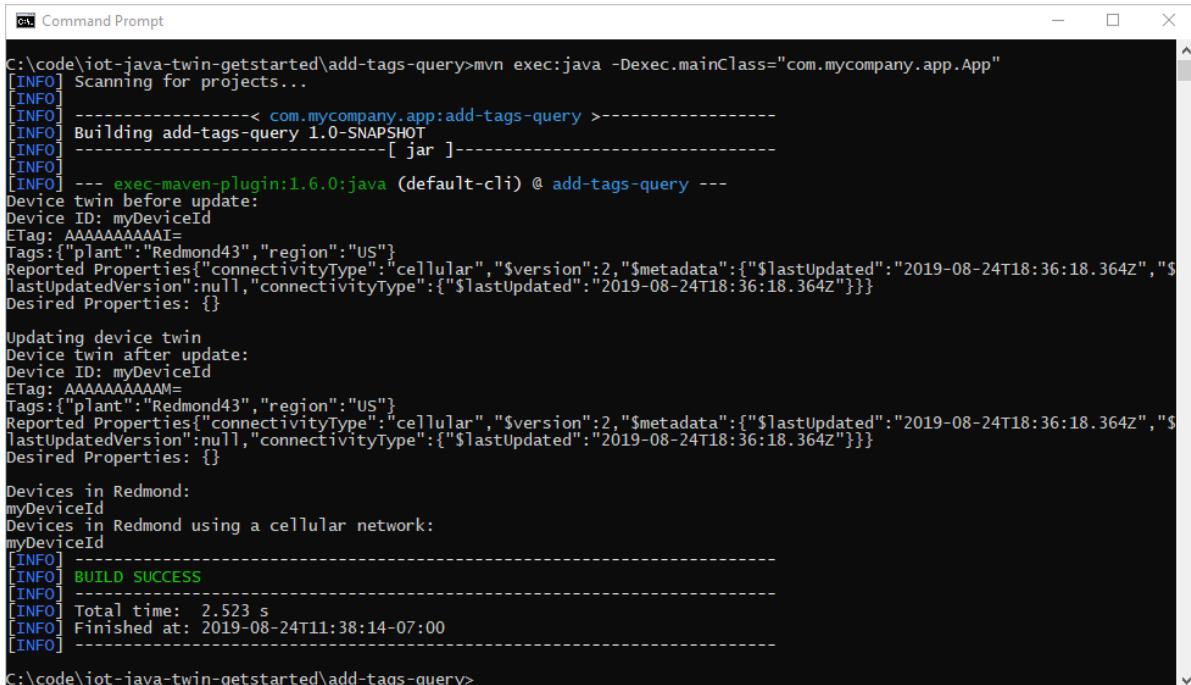
```
mvn exec:java -Dexec.mainClass="com.mycompany.app.App"
```



```
C:\code\iot-java-twin-getstarted\simulated-device>mvn exec:java -Dexec.mainClass="com.mycompany.app.App"
[INFO] Scanning for projects...
[INFO] -----< com.mycompany.app:simulated-device >-----
[INFO] Building simulated-device 1.0-SNAPSHOT
[INFO] [ jar ]-----
[INFO] --- exec-maven-plugin:1.6.0:java (default-cli) @ simulated-device ---
Press any key to exit...
IoT Hub responded to device twin operation with status OK
IoT Hub responded to device twin operation with status OK_EMPTY
```

3. 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:\code\iot-java-twin-getstarted\add-tags-query>mvn exec:java -Dexec.mainClass="com.mycompany.app.App"
[INFO] Scanning for projects...
[INFO] -----< com.mycompany.app:add-tags-query >-----
[INFO] Building add-tags-query 1.0-SNAPSHOT
[INFO] [ jar ]-----
[INFO] --- exec-maven-plugin:1.6.0:java (default-cli) @ add-tags-query ---
Device twin before update:
Device ID: myDeviceId
ETag: AAAAAAAA=
Tags:{"plant":"Redmond43","region":"US"}
Reported Properties{"connectivityType","cellular","$version":2,"$metadata":{"$lastUpdated":"2019-08-24T18:36:18.364Z","$lastUpdatedVersion":null,"connectivityType":{"$lastUpdated":"2019-08-24T18:36:18.364Z"}}}
Desired Properties: {}

Updating device twin
Device twin after update:
Device ID: myDeviceId
ETag: AAAAAAAAAM=
Tags:{"plant":"Redmond43","region":"US"}
Reported Properties{"connectivityType","cellular","$version":2,"$metadata":{"$lastUpdated":"2019-08-24T18:36:18.364Z","$lastUpdatedVersion":null,"connectivityType":{"$lastUpdated":"2019-08-24T18:36:18.364Z"}}}
Desired Properties: {}

Devices in Redmond:
myDeviceId
Devices in Redmond using a cellular network:
myDeviceId
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 2.523 s
[INFO] Finished at: 2019-08-24T11:38:14-07:00
[INFO] -----
```

Now that 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)

7/29/2020 • 12 minutes to read • [Edit Online](#)

Device twins are JSON documents that store device state information, including 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 available only in the standard tier of IoT Hub. For more information about the basic and standard/free IoT Hub tiers, see [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.

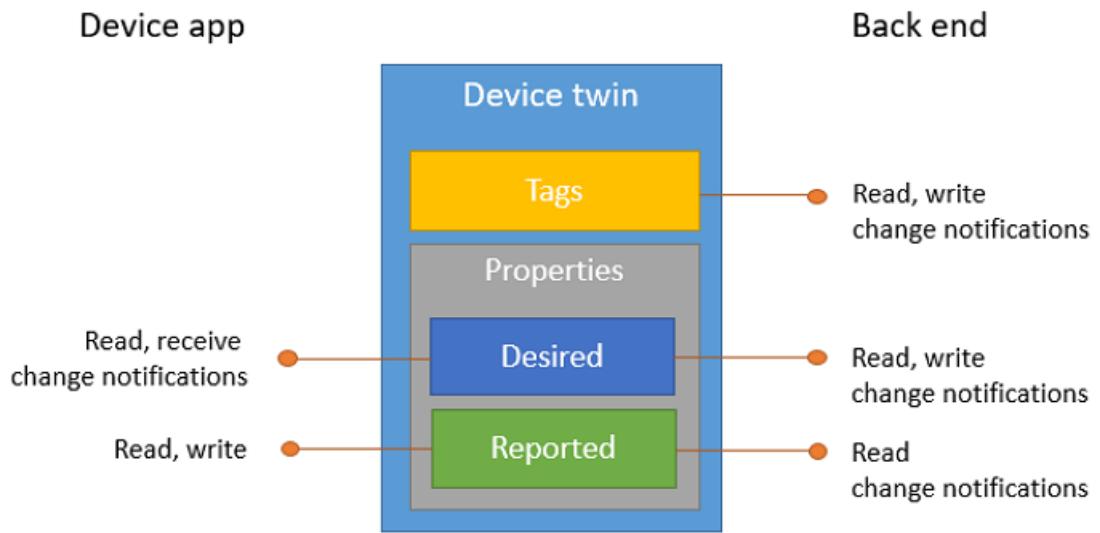
Device twins are designed for synchronization and for querying device configurations and conditions. More information on when to use device twins can be found in [Understand device twins](#).

Device twins are stored in an IoT hub and contain the following elements:

- **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.
- **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.

The following illustration shows device twin organization:



Additionally, the solution back end can query device twins based on all the above data. For more information about device twins, see [Understand device twins](#). For more information about querying, see [IoT Hub query language](#).

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

IoT Hub has SDK support for many device platforms and languages (including C, Java, Javascript, and Python) through [Azure IoT device SDKs](#). For instructions on how to use Python to connect your device to this tutorial's code, and generally to Azure IoT Hub, see the [Azure IoT Python SDK](#).

Prerequisites

- An active Azure account. (If you don't have an account, you can create a [free account](#) in just a couple of minutes.)
- [Python version 3.7 or later](#) is recommended. 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. For other versions of Python supported, see [Azure IoT Device Features](#) in the SDK documentation. If you choose to use Python 2.7, you may need to [install or upgrade pip](#), the Python package management system.
- Make sure that port 8883 is open in your firewall. The device sample in this article uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

Create an IoT hub

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

1. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.
 - **IoT Hub Name:** Enter a name for your hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

The screenshot shows the 'IoT hub' creation page in the Azure portal. At the top, the navigation bar shows 'Home > New > IoT hub'. The main title is 'IoT hub' by Microsoft. Below the title, there are tabs for 'Basics', 'Size and scale', 'Tags', and 'Review + create'. The 'Basics' tab is selected. A sub-header 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 fields for 'Subscription', 'Resource group', 'Region', and 'IoT hub name'. The 'Subscription' dropdown is set to 'Personal IoT items'. The 'Resource group' dropdown has 'Create new' selected. The 'Region' dropdown is set to 'East Asia'. The 'IoT hub name' input field contains the placeholder 'Once your hub is created, this name can't be changed'. At the bottom of the page, there are buttons for 'Review + create', '< Previous', 'Next: Size and scale >', and 'Automation options'.

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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
 Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center [On](#) Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.

If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure](#)

resources.

The screenshot shows the 'Tags' step in the IoT hub creation wizard. At the top, there are tabs for 'Basics', 'Size and scale', 'Tags' (which is underlined in blue), and 'Review + create'. Below the tabs, a note says: 'Tags are name/value pairs. To categorize resources and consolidate billing, apply the same tag to multiple resources and resource groups. Your tags will update automatically if you change your resources.' A 'Learn more' link is provided. A table lists two tags: 'department' with value 'accounting' and another tag row with empty fields. At the bottom are buttons for 'Review + create' (highlighted in blue), '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' step in the IoT hub creation wizard. At the top, there are tabs for 'Basics', 'Size and scale', 'Tags' (underlined in blue), and 'Review + create' (highlighted with a red box). Below the tabs, the configuration details are listed:

- Basics**
 - Subscription: Personal testing
 - Resource group: iot-hubs
 - Region: West US 2
 - IoT hub name: you-hub-name
- 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
 - Azure Security Center: 0.001 USD per device per month
- Tags**
 - department: accounting

At the bottom are buttons for 'Create' (highlighted with a red box), '< Previous: Tags', 'Next >', and 'Automation options'.

8. Select **Create** to create your new hub. Creating the hub takes a few minutes.

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 a

hub unless it has an entry in the identity registry. For more information, see the [IoT Hub developer guide](#).

1. In your IoT hub navigation menu, open **IoT Devices**, then select **New** to add a device in your IoT hub.

The screenshot shows the Azure IoT Hub management interface for the 'iot-hub-contoso-one' hub. The left sidebar contains a navigation menu with various options like Overview, Activity log, Access control (IAM), Tags, Events, Settings (Shared access policies, Pricing and scale, IP Filter, Certificates, Built-in endpoints, Manual failover (preview), Properties, Locks, Export template), Explorers (Query explorer), and Automatic Device Management (with 'IoT devices' highlighted). The main content area is titled 'iot-hub-contoso-one - IoT devices'. It features a search bar, a 'New' button (which is highlighted with a red box), and a 'Refresh' and 'Delete' button. Below these are sections for 'View, create, delete, and update devices in your IoT Hub.' and a 'Query devices' section with a query editor. The table below shows no results.

| DEVICE ID | STATUS | LAST ACTIVITY TIME (UTC) | LAST STATUS UPDATE (UTC) | AUTHENTICATION T... | CLOUD ... |
|------------|--------|--------------------------|--------------------------|---------------------|-----------|
| No results | | | | | |

2. In **Create a device**, provide a name for your new device, such as **myDeviceId**, and select **Save**. This action creates a device identity for your IoT hub.

Home > All resources > iot-hub-contoso-one - IoT devices > Create a device

Create a device

Find Certified for Azure IoT devices in the Device Catalog

*** Device ID** myDeviceId

Authentication type: Symmetric key

*** Primary key** Enter your primary key

*** Secondary key** Enter your secondary key

Auto-generate keys:

Connect this device to an IoT hub: Enable

Parent device: 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 **Primary Connection String** to use later.

Home > iot-hub-contoso-one - IoT devices > myDeviceId

myDeviceId

Device ID: myDeviceId

Primary Key: HZAwv1PN3suNBkaiQU1UeEiNB3j0=

Secondary Key: G7615rcbqyWFzcfTlgmad55lGVa4l=

Primary Connection String: HostName=iot-hub-contoso-one.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=QdSim6i7cptUcemyGVSeiRKOV2ZGFSJpbmykIVYM9df=

Secondary Connection String: HostName=iot-hub-contoso-one.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=q32joXuwIEbbqKYkj8sF82q2lnqzGZspqkl2nqz=

Enable connection to IoT Hub: Enable Disable

Parent device: No parent device

Module Identities **Configurations**

There are no module identities for this 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](#).

Get the IoT hub connection string

In this article, you create a back-end service that adds desired properties to a device twin and then queries the identity registry to find all devices with reported properties that have been updated accordingly. Your service needs the **service connect** permission to modify desired properties of a device twin, and it needs the **registry read** permission to query the identity registry. There is no default shared access policy that contains only these two permissions, so you need to create one.

To create a shared access policy that grants **service connect** and **registry read** permissions and get a connection string for this policy, follow these steps:

1. In the [Azure portal](#), select **Resource groups**. Select the resource group where your hub is located, and then select your hub from the list of resources.
2. On the left-side pane of your hub, select **Shared access policies**.
3. From the top menu above the list of policies, select **Add**.
4. Under **Add a shared access policy**, enter a descriptive name for your policy, such as *serviceAndRegistryRead*. Under **Permissions**, select **Registry read** and **Service connect**, and then select **Create**.

The screenshot shows the Azure portal interface for managing an IoT hub. On the left, the navigation menu includes options like Overview, Activity log, Access control (IAM), Tags, Events, Settings, Shared access policies (which is highlighted with a red box), Pricing and scale, IP Filter, Certificates, Built-in endpoints, Manual failover (preview), Properties, Locks, and Export template. The main content area shows the 'Shared access policies' blade for the 'contoso-hub-1' IoT hub. At the top right of this blade is a 'Create' button. Below it is a table listing existing policies: iothubowner (permissions: registry write, service connect), service (permissions: service connect), device (permissions: device connect), registryRead (permissions: registry read), and registryReadWrite (permissions: registry write). To the right of the table is a 'Add a shared access policy' dialog box. The 'Access policy name' field contains 'serviceAndRegistryRead'. The 'Permissions' section is expanded, showing checkboxes for 'Registry read' (checked), 'Registry write' (unchecked), 'Service connect' (checked), and 'Device connect' (unchecked). A red box highlights the 'Create' button at the bottom of the dialog.

5. Select your new policy from the list of policies.
6. Under **Shared access keys**, select the copy icon for the **Connection string -- primary key** and save the value.

The screenshot shows the Azure IoT Hub Shared access policies page. A new policy named "serviceAndRegistryRead" is being created. The policy is assigned to the "service" endpoint and grants "registry read, service connect" permissions. The "Primary key" and "Secondary key" connection strings are also displayed.

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

For more information about IoT Hub shared access policies and permissions, see [Access control and permissions](#).

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. In your working directory, open a command prompt and install the [Azure IoT Hub Service SDK for Python](#).

```
pip install azure-iot-hub
```

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
from time import sleep
from azure.iot.hub import IoTHubRegistryManager
from azure.iot.hub.models import Twin, TwinProperties, QuerySpecification, QueryResult
```

4. Add the following code. Replace `[IoTHub Connection String]` with the IoT hub connection string you copied in [Get the IoT hub connection string](#). Replace `[Device Id]` with the device ID you registered in [Register a new device in the IoT hub](#).

```
IOTHUB_CONNECTION_STRING = "[IoTHub Connection String]"
DEVICE_ID = "[Device Id]"
```

5. Add the following code to the `AddTagsAndQuery.py` file:

```

def iothub_service_sample_run():
    try:
        iothub_registry_manager = IoTHubRegistryManager(IOTHUB_CONNECTION_STRING)

        new_tags = {
            'location' : {
                'region' : 'US',
                'plant' : 'Redmond43'
            }
        }

        twin = iothub_registry_manager.get_twin(DEVICE_ID)
        twin_patch = Twin(tags=new_tags, properties= TwinProperties(desired={'power_level' : 1}))
        twin = iothub_registry_manager.update_twin(DEVICE_ID, twin_patch, twin.etag)

        # Add a delay to account for any latency before executing the query
        sleep(1)

        query_spec = QuerySpecification(query="SELECT * FROM devices WHERE tags.location.plant = "
'Redmond43"')
        query_result = iothub_registry_manager.query_iot_hub(query_spec, None, 100)
        print("Devices in Redmond43 plant: {}".format(', '.join([twin.device_id for twin in
query_result.items])))

        print()

        query_spec = QuerySpecification(query="SELECT * FROM devices WHERE tags.location.plant = "
'Redmond43' AND properties.reported.connectivity = 'cellular')
        query_result = iothub_registry_manager.query_iot_hub(query_spec, None, 100)
        print("Devices in Redmond43 plant using cellular network: {}".format(', '.join([twin.device_id
for twin in query_result.items])))

    except Exception as ex:
        print("Unexpected error {0}".format(ex))
        return
    except KeyboardInterrupt:
        print("IoT Hub Device Twin service sample stopped")

```

The **IoTHubRegistryManager** object exposes all the methods required to interact with device twins from the service. The code first initializes the **IoTHubRegistryManager** object, then updates the device twin for **DEVICE_ID**, 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 a 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 Python IoT Hub Device Twin service sample...")
    print()

    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.

```
C:\code\iot-python-twin-getstarted>python AddTagsAndQuery.py
Starting the Python IoT Hub Device Twin service sample...
Devices in Redmond43 plant: myDeviceId
Devices in Redmond43 plant using cellular network:
C:\code\iot-python-twin-getstarted>
```

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. From a command prompt in your working directory, install the **Azure IoT Hub Device SDK for Python**:

```
pip install azure-iot-device
```

2. Using a text editor, create a new **ReportConnectivity.py** file.

3. Add the following code to import the required modules from the device SDK:

```
import time
import threading
from azure.iot.device import IoTHubModuleClient
```

4. Add the following code. Replace the **[IoTHub Device Connection String]** placeholder value with the device connection string you copied in [Register a new device in the IoT hub](#).

```
CONNECTION_STRING = "[IoTHub Device Connection String]"
```

5. Add the following code to the **ReportConnectivity.py** file to implement the device twins functionality:

```

def twin_update_listener(client):
    while True:
        patch = client.receive_twin_desired_properties_patch() # blocking call
        print("Twin patch received:")
        print(patch)

def iothub_client_init():
    client = IoTHubModuleClient.create_from_connection_string(CONNECTION_STRING)
    return client

def iothub_client_sample_run():
    try:
        client = iothub_client_init()

        twin_update_listener_thread = threading.Thread(target=twin_update_listener, args=(client,))
        twin_update_listener_thread.daemon = True
        twin_update_listener_thread.start()

        # Send reported
        print ("Sending data as reported property...")
        reported_patch = {"connectivity": "cellular"}
        client.patch_twin_reported_properties(reported_patch)
        print ("Reported properties updated")

        while True:
            time.sleep(1000000)
    except KeyboardInterrupt:
        print ("IoT Hub Device Twin device sample stopped")

```

The **IoTHubModuleClient** object exposes all the methods you require to interact with device twins from the device. The previous code, after it initializes the **IoTHubModuleClient** 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 Python IoT Hub Device Twin device sample...")
    print ("IoTHubModuleClient waiting for commands, press Ctrl-C to exit")

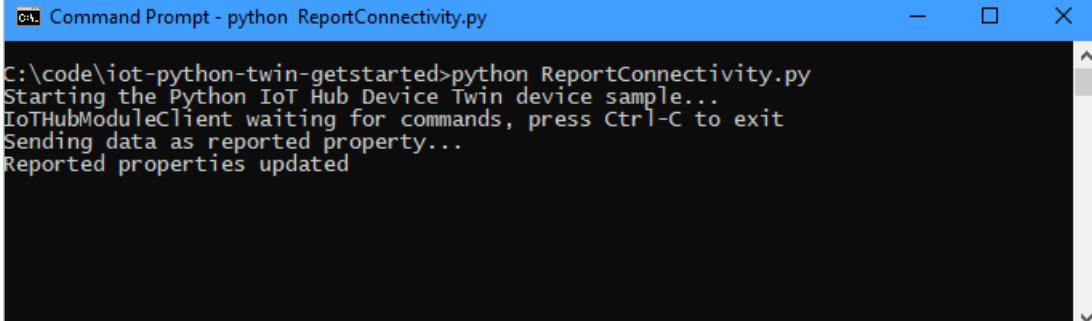
    iothub_client_sample_run()

```

7. Run the device app:

```
python ReportConnectivity.py
```

You should see confirmation the device twin reported properties were updated.



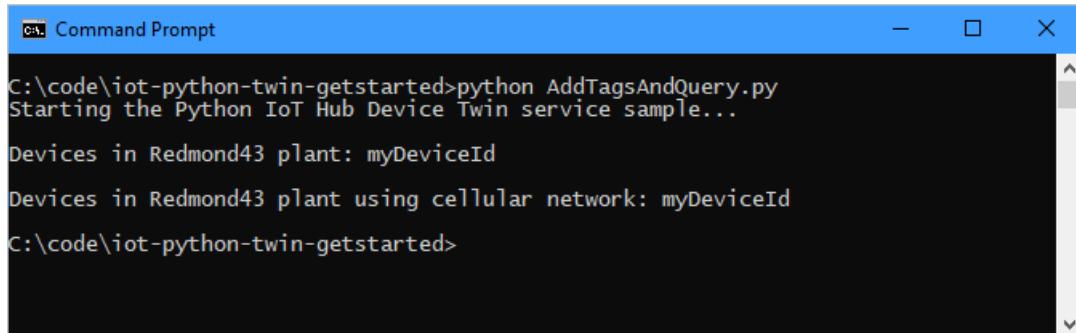
```
C:\code\iot-python-twin-getstarted>python ReportConnectivity.py
Starting the Python IoT Hub Device Twin device sample...
IoTHubModuleClient waiting for commands, press Ctrl-C to exit
Sending data as reported property...
Reported properties updated
```

8. Now that the device reported its connectivity information, it should appear in both queries. Go back and

run the queries again:

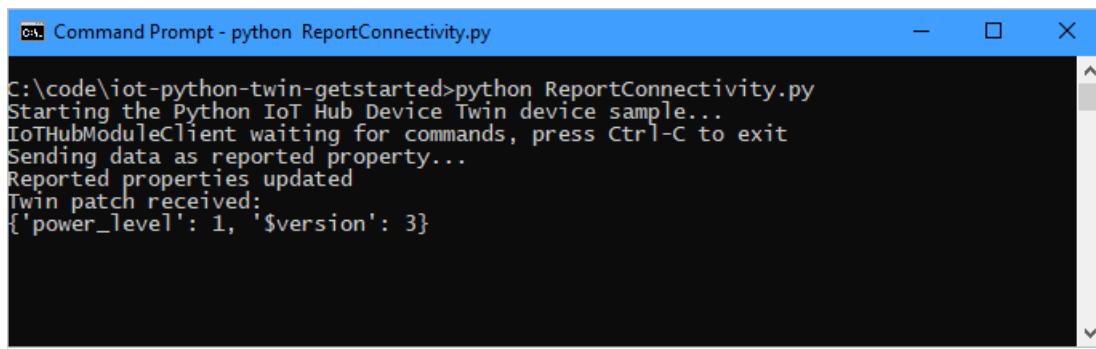
```
python AddTagsAndQuery.py
```

This time your {Device ID} should appear in both query results.



```
C:\code\iot-python-twin-getstarted>python AddTagsAndQuery.py
Starting the Python IoT Hub Device Twin service sample...
Devices in Redmond43 plant: myDeviceId
Devices in Redmond43 plant using cellular network: myDeviceId
C:\code\iot-python-twin-getstarted>
```

In your device app, you'll see confirmation that the desired properties twin patch sent by the service app was received.



```
C:\code\iot-python-twin-getstarted>python ReportConnectivity.py
Starting the Python IoT Hub Device Twin device sample...
IoTHubModuleClient waiting for commands, press Ctrl-C to exit
Sending data as reported property...
Reported properties updated
Twin patch received:
{'power_level': 1, '$version': 3}
```

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

4/21/2020 • 7 minutes to read • [Edit Online](#)

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 provide 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, module identities and module twins allow for isolated configuration and conditions for each component.

In this tutorial, you will learn:

- How to create a module identity in the portal.
- How to use a .NET device SDK to 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](#).

Prerequisites

- Visual Studio.
- An active Azure account. If you don't have an account, you can create a [free account](#) in just a couple of minutes.

Create a hub

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

1. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.

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

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

The screenshot shows the 'Basics' step of the IoT hub creation wizard. It includes fields for Subscription, Resource group, Region, and IoT hub name, all highlighted with a red box. Below the form are navigation buttons: 'Review + create', '< Previous', 'Next: Size and scale >', and 'Automation options'.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags Review + create

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

Project details

Choose the subscription you'll use to manage deployments and costs. Use resource groups like folders to help you organize and manage resources.

Subscription * ⓘ Personal IoT items

Resource group * ⓘ Create new

Region * ⓘ East Asia

IoT hub name * ⓘ Once your hub is created, this name can't be changed

[Review + create](#) < Previous [Next: Size and scale >](#) Automation options

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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](#)

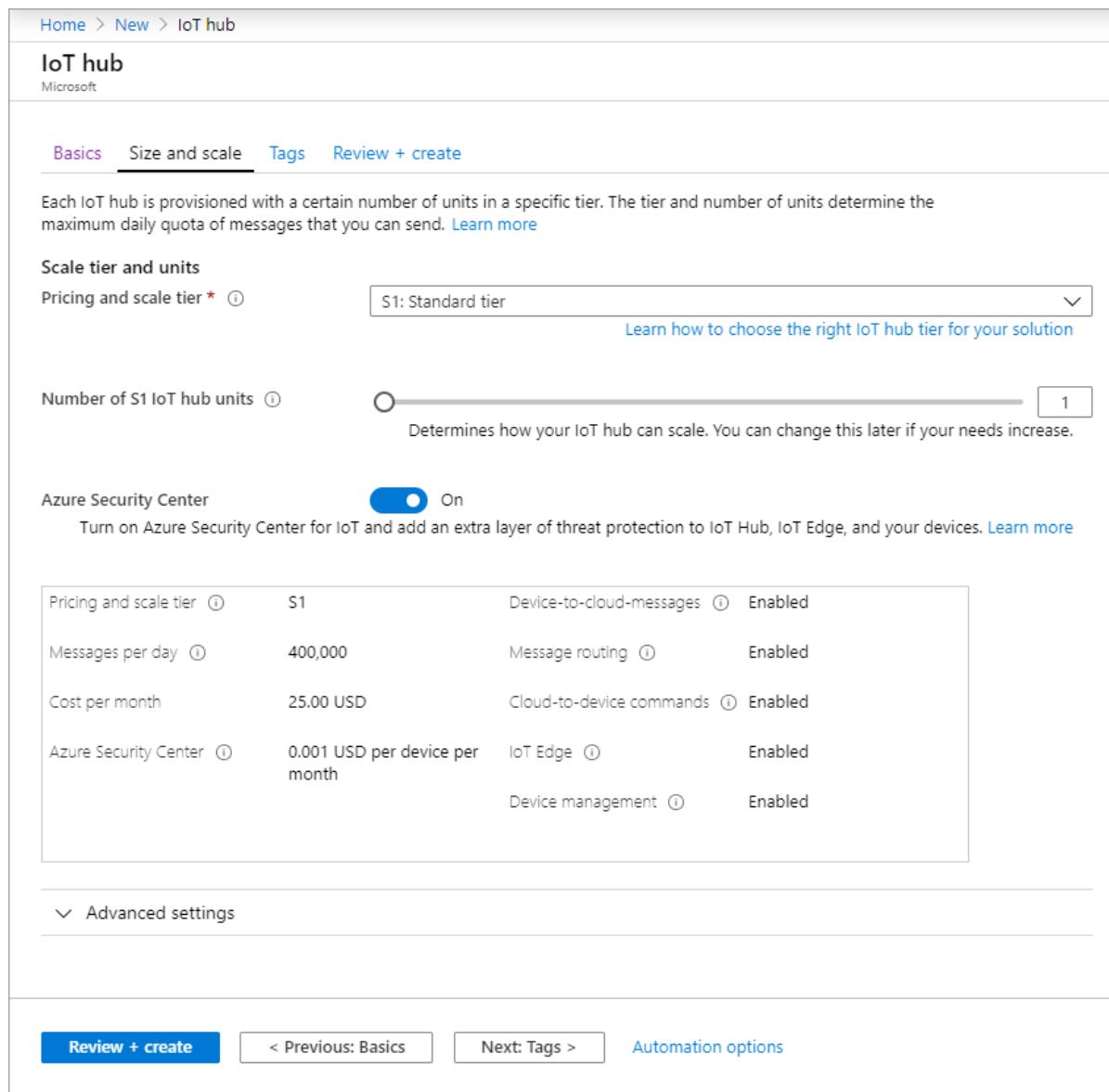
Number of S1 IoT hub units ⓘ 1
Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center On
Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)



You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.

If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure](#)

resources.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags **Review + create**

Tags are name/value pairs. To categorize resources and consolidate billing, apply the same tag to multiple resources and resource groups. Your tags will update automatically if you change your resources. [Learn more](#)

| Name ⓘ | Value ⓘ | Resource |
|------------|---------|--------------------|
| department | : | accounting IoT Hub |
| | : | IoT Hub |

Review + create < Previous: Size and scale Next: Review + create > Automation options

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags **Review + create**

Basics

| | |
|----------------|------------------|
| Subscription | Personal testing |
| Resource group | iot-hubs |
| Region | West US 2 |
| IoT hub name | you-hub-name |

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 |
| Azure Security Center | 0.001 USD per device per month |

Tags

| | |
|------------|------------|
| department | accounting |
|------------|------------|

Create < Previous: Tags Next > Automation options

8. Select **Create** to create your new hub. Creating the hub takes a few minutes.

Register a new device in the hub

In this section, you create a device identity in the identity registry in your IoT hub. A device cannot connect to a hub

unless it has an entry in the identity registry. For more information, see the [IoT Hub developer guide](#).

1. In your IoT hub navigation menu, open **IoT Devices**, then select **New** to add a device in your IoT hub.

The screenshot shows the Azure IoT Hub management interface for the 'iot-hub-contoso-one' hub. The top navigation bar includes 'Home', 'All resources', and the specific hub name. Below the navigation is a search bar and a toolbar with 'New' (highlighted), 'Refresh', and 'Delete' buttons. A message states: 'View, create, delete, and update devices in your IoT Hub.' To the right is a query editor with fields for 'Field' (set to 'select or enter a property name'), 'Operator' (set to '='), and 'Value' (set to 'specify constraint value'). Below the editor is a 'Query devices' button and a link to 'Switch to query editor'. The main table area has columns for 'DEVICE ID', 'STATUS', 'LAST ACTIVITY TIME (UTC)', 'LAST STATUS UPDATE (UTC)', 'AUTHENTICATION T...', and 'CLOUD ...'. A message 'No results' is displayed. The left sidebar contains sections for Overview, Activity log, Access control (IAM), Tags, Events, Settings (with Shared access policies, Pricing and scale, IP Filter, Certificates, Built-in endpoints, Manual failover (preview), Properties, Locks, Export template), Explorers (Query explorer, IoT devices - highlighted with a red box), and Automatic Device Management.

2. In **Create a device**, provide a name for your new device, such as **myDeviceId**, and select **Save**. This action creates a device identity for your IoT hub.

Home > All resources > iot-hub-contoso-one - IoT devices > Create a device

Create a device

Find Certified for Azure IoT devices in the Device Catalog

*** Device ID** myDeviceId

Authentication type: Symmetric key

*** Primary key**

*** Secondary key**

Auto-generate keys:

Connect this device to an IoT hub: Enable

Parent device: No 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 **Primary Connection String** to use later.

Home > iot-hub-contoso-one - IoT devices > myDeviceId

| Device ID | myDeviceId | | |
|--|---|-------------------------------------|--------------------------|
| Primary Key | H2Awv1PN3suNBkaiQU1UeEINB3j0= | | |
| Secondary Key | G7615rzcbyWFzcfIlgmad55lGVa4I= | | |
| Primary Connection String | HostName=iot-hub-contoso-one.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=QdSim6i7cptUCeMYGVSeIRKOV2ZGFSJpbmyklVYM9df= | | |
| Secondary Connection String | HostName=iot-hub-contoso-one.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=q32oiXuwifEXbbqKYkjv8sF82qZInqzGZspqkl2nqz= | | |
| Enable connection to IoT Hub | <input checked="" type="radio"/> Enable <input type="radio"/> Disable | | |
| Parent device | No parent device | | |
| Module Identities Configurations | | | |
| MODULE ID | CONNECTION STATE | CONNECTION STATE LAST UPDATED (UTC) | LAST ACTIVITY TIME (UTC) |
| There are no module identities for this 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. To add an identity, follow these steps:

1. For the device you created in the previous section, choose **Add Module Identity** to create your first module identity.
2. Enter the name *myFirstModule*. Save your module identity.

The screenshot shows two windows side-by-side. On the left is the 'Device Identity Details' page for 'myDeviceId'. It lists various connection strings and configuration options. On the right is the 'Add Module Identity' dialog. In the dialog, the 'Module Identity Name' field is set to 'myFirstModule' (highlighted with a red box). The 'Authentication type' dropdown is set to 'Symmetric key' (also highlighted with a red box). Below that, there are fields for 'Primary key' and 'Secondary key', both of which have 'Enter your primary key' placeholder text. A checked checkbox for 'Auto-generate keys' is also visible. At the bottom right of the dialog is a large blue 'Save' button (also highlighted with a red box).

Your new module identity appears at the bottom of the screen. Select it to see module identity details.

This screenshot shows the 'Module Identity Details' page for the 'myFirstModule' identity. It displays the module identity name ('myDeviceId/myFirstModule'), primary key, secondary key, and two connection strings. The second connection string is highlighted with a red box. Below the table, there are navigation arrows.

Save the **Connect string - primary key**. You use it in the next section to you set up your module on the device.

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.

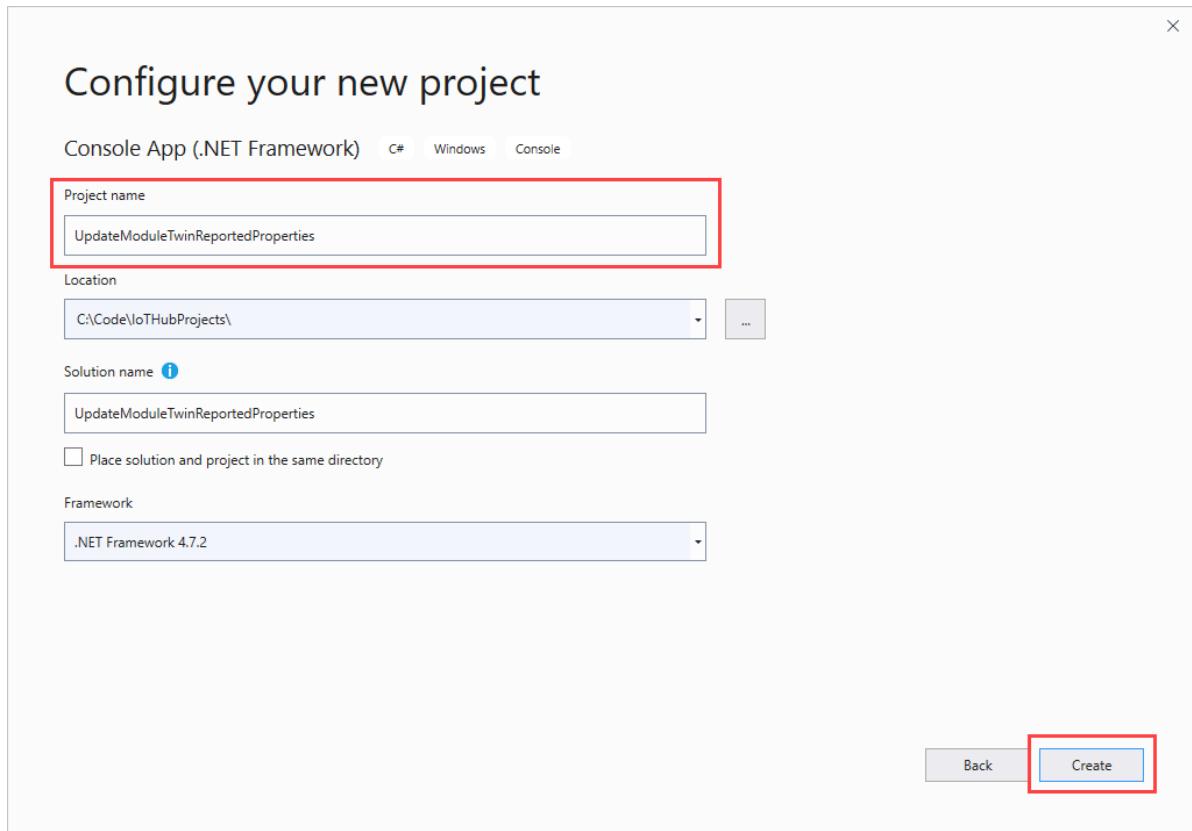
Create a Visual Studio project

To create an app that updates the module twin reported properties, follow these steps:

1. In Visual Studio, select **Create a new project**, then choose **Console App (.NET Framework)**, and select

Next.

2. In **Configure your new project**, enter *UpdateModuleTwinReportedProperties* as the **Project name**. Select **Create** to continue.



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 pre-release device SDKs. To install it, follow these steps:

1. In Visual Studio, open **Tools > NuGet Package Manager > Manage NuGet Packages for Solution**.
2. Select **Browse**, and then select **Include prerelease**. Search for *Microsoft.Azure.Devices.Client*. Select the latest version and install.



Now you have access to all the module features.

Get your module connection string

You need the module connection string for your console app. Follow these steps:

1. Sign in to the [Azure portal](#).
2. Navigate to your IoT hub and select **IoT Devices**. Open **myFirstDevice** and you see that **myFirstModule** was successfully created.

3. Select **myFirstModule** under **Module Identities**. In **Module Identity Details**, copy the **Connection string (primary key)**.

The screenshot shows the 'Module Identity Details' page for a module named 'myFirstModule'. The page includes fields for 'Module Identity Name' (set to 'myDeviceId/myFirstModule'), 'Primary key' (containing a long hex string), 'Secondary key' (containing a long hex string), 'Connection string (primary key)' (containing a URL with placeholders for Device ID and Module ID), and 'Connection string (secondary key)' (containing a similar URL). A red box highlights the 'Connection string (primary key)' field.

Create UpdateModuleTwinReportedProperties console app

To create your app, follow these steps:

1. 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;
```

2. 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;
```

3. 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);
}
```

4. Finally, replace the **Main** method with the following code:

```

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}");
}

```

You can build and run this app by using F5.

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.

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 (.NET)

4/21/2020 • 9 minutes to read • [Edit Online](#)

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 provide 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, module identities and module twins allow for isolated configuration and conditions for each component.

At the end of this tutorial, you have two .NET console apps:

- **CreateIdentities**. This app creates a device identity, a module identity, and associated security key to connect your device and module clients.
- **UpdateModuleTwinReportedProperties**. This app 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](#).

Prerequisites

- Visual Studio.
- An active Azure account. If you don't have an account, you can create a [free account](#) in just a couple of minutes.

Create a hub

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

1. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription**: Select the subscription to use for your hub.
 - **Resource Group**: Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region**: Select the region in which you want your hub to be located. Select the location closest to

you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.

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

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

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 that, the title 'IoT hub' is displayed with the Microsoft logo. A red box highlights the 'Project details' section, which includes fields for Subscription, Resource group, Region, and IoT hub name. The 'Subscription' field is set to 'Personal IoT items'. The 'Resource group' field has a dropdown menu with 'Create new' option. The 'Region' field is set to 'East Asia'. The 'IoT hub name' field contains the placeholder text 'Once your hub is created, this name can't be changed'. At the bottom of the form, there are navigation buttons: 'Review + create' (blue), '< Previous' (disabled), 'Next: Size and scale >' (highlighted with a red box), and 'Automation options'.

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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 Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center On Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.
If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure](#)

resources.

The screenshot shows the 'Tags' tab of the IoT hub configuration page. It displays a table with one row of data:

| Name | Value | Resource | Actions |
|------------|------------|----------|-----------------------------|
| department | accounting | IoT Hub | ... Delete |
| | | IoT Hub | |

Below the table are navigation buttons: 'Review + create' (highlighted), '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' step of the IoT hub creation wizard. It displays the following configuration details:

| Setting | Value |
|----------------------------|--------------------------------|
| Subscription | Personal testing |
| Resource group | iot-hubs |
| Region | West US 2 |
| IoT hub name | you-hub-name |
| 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 |
| Azure Security Center | 0.001 USD per device per month |
| Tags | |
| department | accounting |

At the bottom are buttons: 'Create' (highlighted), '< Previous: Tags', 'Next >', and 'Automation options'.

8. Select **Create** to create your new hub. Creating the hub takes a few minutes.

Get the IoT hub connection string

In this article, you create a back-end service that adds a device in the identity registry and then adds a module to

that device. Your service requires the **registry write** permission. By default, every IoT hub is created with a shared access policy named **registryReadWrite** that grants this permission.

To get the IoT Hub connection string for the **registryReadWrite** policy, follow these steps:

1. In the [Azure portal](#), select **Resource groups**. Select the resource group where your hub is located, and then select your hub from the list of resources.
2. On the left-side pane of your hub, select **Shared access policies**.
3. From the list of policies, select the **registryReadWrite** policy.
4. Under **Shared access keys**, select the copy icon for the **Connection string -- primary key** and save the value.

The screenshot shows the Azure portal interface for managing an IoT hub. On the left, the navigation pane includes options like Overview, Activity log, Access control (IAM), Tags, Events, Settings, and Shared access policies (which is highlighted with a red box). The main content area displays the 'iot-hub-contoso-one - Shared access policies' page. It lists several policies: iothubowner, service, device, registryRead, and registryReadWrite (also highlighted with a red box). To the right, the 'registryReadWrite' policy details are shown, including its permissions (Registry read and Registry write checked) and its shared access keys. The 'Connection string--primary key' is specifically highlighted with a red box.

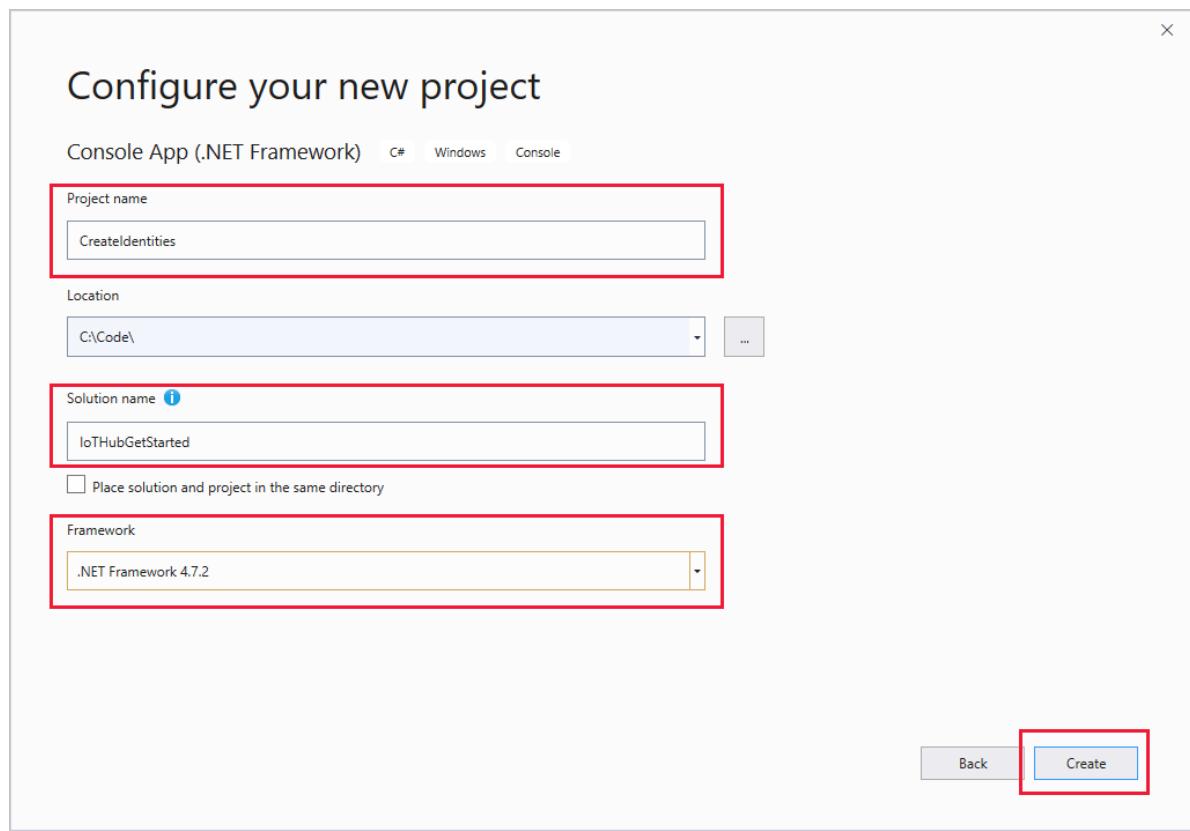
For more information about IoT Hub shared access policies and permissions, see [Access control and permissions](#).

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 hub. A device or module can't connect to hub unless it has an entry in the identity registry. For more information, see the [Identity Registry section of the IoT Hub developer guide](#).

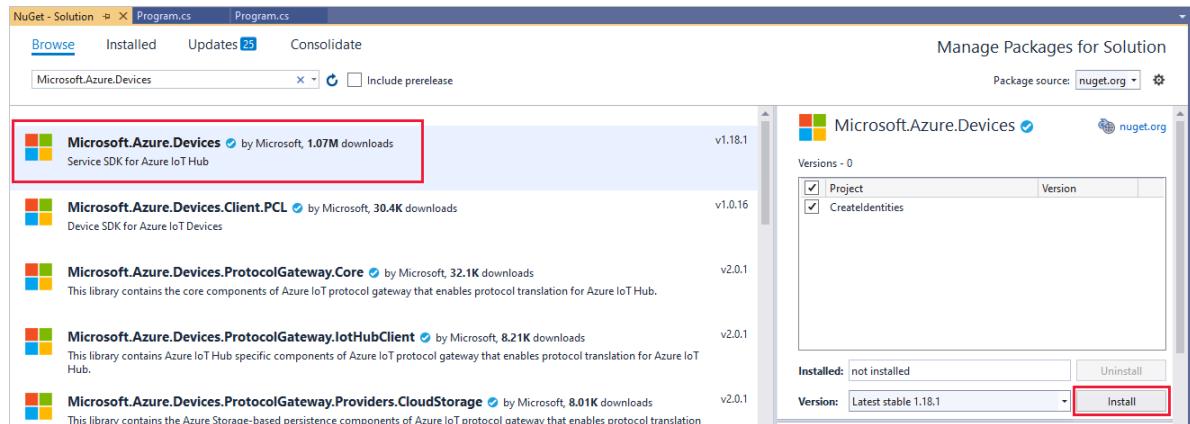
When you run this console app, it generates a unique ID and key for both device and module. Your device and module use these values to identify themselves when it sends device-to-cloud messages to IoT Hub. The IDs are case-sensitive.

1. Open Visual Studio, and select **Create a new project**.
2. In **Create a new project**, select **Console App (.NET Framework)**.
3. Select **Next** to open **Configure your new project**. Name the project *CreateIdentities* and name the solution *IoTHubGetStarted*. Make sure the .NET Framework version is 4.6.1 or later.



4. In Visual Studio, open Tools > NuGet Package Manager > Manage NuGet Packages for Solution. Select the **Browse** tab.

5. Search for **Microsoft.Azure.Devices**. Select it and then select **Install**.



6. Add the following `using` statements at the top of the **Program.cs** file:

```
using Microsoft.Azure.Devices;
using Microsoft.Azure.Devices.Common.Exceptions;
```

7. 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";
```

8. Add the following code to the **Main** class.

```
static void Main(string[] args)
{
    AddDeviceAsync().Wait();
    AddModuleAsync().Wait();
}
```

9. 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 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 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.

10. Run this app, and make a note of the device key and module key.

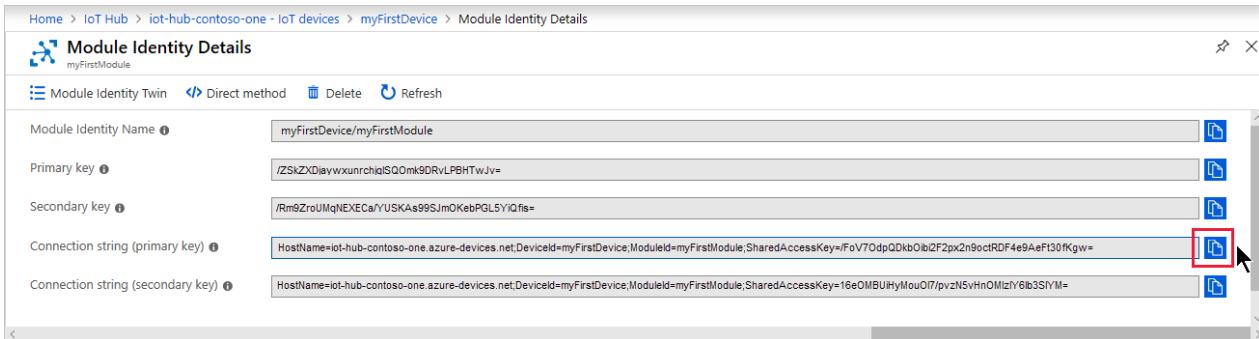
NOTE

The IoT Hub identity registry only stores device and module identities to enable secure access to the 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 app 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](#).

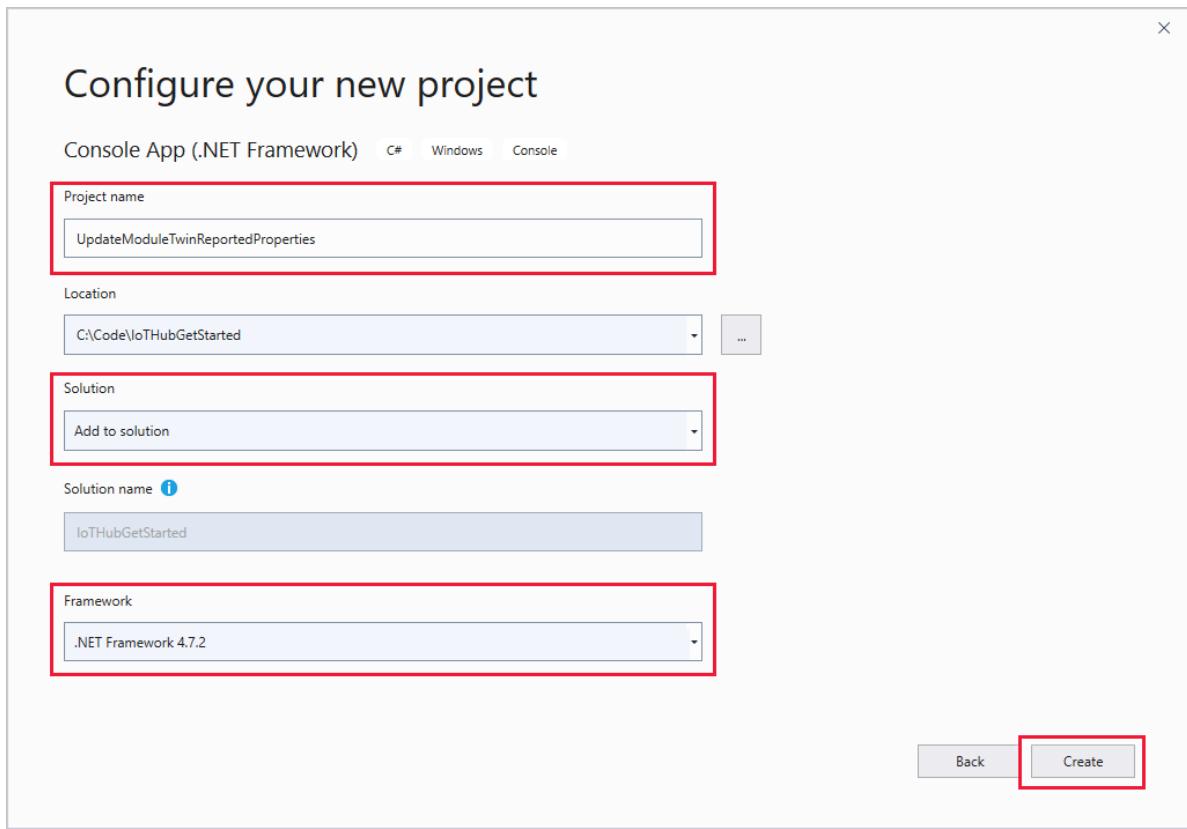
Update the module twin using .NET device SDK

In this section, you create a .NET console app on your simulated device that updates the module twin reported properties.

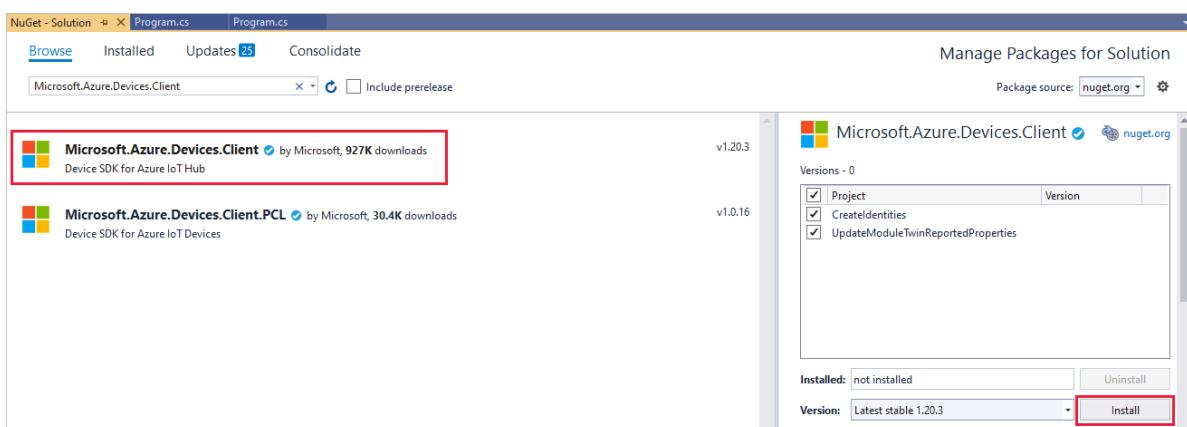
Before you begin, get your module connection string. Sign in to the [Azure portal](#). Navigate to your hub and select **IoT Devices**. Find **myFirstDevice**. Select **myFirstDevice** to open it, and then select **myFirstModule** to open it. In **Module Identity Details**, copy the **Connection string (primary key)** when needed in the following procedure.



1. In Visual Studio, add a new project to your solution by selecting **File > New > Project**. In Create a new project, select **Console App (.NET Framework)**, and select **Next**.
2. Name the project *UpdateModuleTwinReportedProperties*. For Solution, select **Add to solution**. Make sure the .NET Framework version is 4.6.1 or later.



3. Select **Create** to create your project.
4. In Visual Studio, open Tools > NuGet Package Manager > Manage NuGet Packages for Solution. Select the **Browse** tab.
5. Search for and select **Microsoft.Azure.Devices.Client**, and then select **Install**.



6. 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;
```

7. 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);
}

```

8. 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);
}

```

9. 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.

10. Optionally, you can add these statements to the **Main** method to send an event to IoT Hub from your module. Place these lines below the `try catch` block.

```
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 can now run the apps.

1. In Visual Studio, in **Solution Explorer**, right-click your solution, and then select **Set StartUp projects**.
2. Under **Common Properties**, select **Startup Project**.
3. Select **Multiple startup projects**, and then select **Start** as the action for the apps, and **OK** to accept your changes.
4. Press **F5** to start the apps.

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 (Python)

7/29/2020 • 10 minutes to read • [Edit Online](#)

NOTE

Module identities and module twins are similar to Azure IoT Hub device identities and device twins, but provide finer granularity. While Azure IoT Hub device identities and device twins enable a back-end application to configure a device and provide visibility on the device's conditions, module identities and module twins provide these capabilities for individual components of a device. On capable devices with multiple components, such as operating system based devices or firmware devices, they allow for isolated configuration and conditions for each component.

At the end of this tutorial, you have three Python apps:

- **CreateModule**, which creates a device identity, a module identity, and associated security keys to connect your device and module clients.
- **UpdateModuleTwinDesiredProperties**, which sends updated module twin desired properties to your IoT Hub.
- **ReceiveModuleTwinDesiredPropertiesPatch**, which receives the module twin desired properties patch on your device.

NOTE

IoT Hub has SDK support for many device platforms and languages (including C, Java, Javascript, and Python) through [Azure IoT device SDKs](#). For instructions on how to use Python to connect your device to this tutorial's code, and generally to Azure IoT Hub, see the [Azure IoT Python SDK](#).

Prerequisites

- An active Azure account. (If you don't have an account, you can create a [free account](#) in just a couple of minutes.)
- [Python version 3.7 or later](#) is recommended. 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. For other versions of Python supported, see [Azure IoT Device Features](#) in the SDK documentation. If you choose to use Python 2.7, you may need to [install or upgrade pip](#), the Python package management system.

Create an IoT hub

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

1. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.

4. On the Basics tab, complete the fields as follows:

- **Subscription:** Select the subscription to use for your hub.
- **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
- **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.
- **IoT Hub Name:** Enter a name for your hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

The screenshot shows the 'Basics' tab of the IoT hub creation wizard. At the top, there's a breadcrumb navigation: Home > New > IoT hub. Below that is the title 'IoT hub' and the Microsoft logo. The 'Basics' tab is selected, indicated by a underline. There are four tabs in total: Basics (selected), Size and scale, Tags, and Review + create. A descriptive text below the tabs 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 asks to choose a subscription and resource group, and specifies a region and IoT hub name. The 'Subscription' field is set to 'Personal IoT items'. The 'Resource group' field has a dropdown menu with 'Create new' option highlighted. The 'Region' field is set to 'East Asia'. The 'IoT hub name' field contains 'Once your hub is created, this name can't be changed'. A red box highlights the 'Subscription', 'Resource group', 'Region', and 'IoT hub name' fields. At the bottom of the page, there are navigation links: 'Review + create' (in blue), '< Previous', 'Next: Size and scale >' (which is highlighted with a red box), and 'Automation options'.

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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
 Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center On
 Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.

If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure](#)

resources.

The screenshot shows the 'Tags' tab of the IoT hub configuration page. It displays a table with one row of data:

| Name | Value | Resource |
|------------|------------|----------|
| department | accounting | IoT Hub |
| | | IoT Hub |

Below the table are navigation buttons: 'Review + create' (highlighted in blue), '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' page for the IoT hub. The configuration details are as follows:

| Setting | Value |
|----------------------------|--------------------------------|
| Subscription | Personal testing |
| Resource group | iot-hubs |
| Region | West US 2 |
| IoT hub name | you-hub-name |
| 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 |
| Azure Security Center | 0.001 USD per device per month |
| Tags | |
| department | accounting |

At the bottom are buttons: 'Create' (highlighted in red), '< Previous: Tags', 'Next >', and 'Automation options'.

8. Select **Create** to create your new hub. Creating the hub takes a few minutes.

Get the IoT hub connection string

In this article, you create a back-end service that adds a device in the identity registry and then adds a module to

that device. This service requires the **registry write** permission (which also includes **registry read**). You also create a service that adds desired properties to the module twin for the newly created module. This service needs the **service connect** permission. Although there are default shared access policies that grant these permissions individually, in this section, you create a custom shared access policy that contains both of these permissions.

To create a shared access policy that grants **service connect** and **registry write** permissions and to get a connection string for this policy, follow these steps:

1. In the [Azure portal](#), select **Resource groups**. Select the resource group where your hub is located, and then select your hub from the list of resources.
2. On the left-side pane of your hub, select **Shared access policies**.
3. From the top menu above the list of policies, select **Add**.
4. Under **Add a shared access policy**, enter a descriptive name for your policy, such as `serviceAndRegistryReadWrite`. Under **Permissions**, select **Registry write** and **Service connect**, and then select **Create**. (The **Registry read** permission is included automatically when you select **Registry write**.)

The screenshot shows the Azure IoT Hub interface. On the left, the 'Shared access policies' blade is open, showing a list of existing policies: iothubowner (permissions: registry write, service connect), service (permissions: service connect), device (permissions: device connect), registryRead (permissions: registry read), and registryReadWrite (permissions: registry write). A red box highlights the 'Shared access policies' link in the sidebar. On the right, a modal dialog titled 'Add a shared access policy' is displayed. It has fields for 'Access policy name' (set to 'serviceAndRegistryReadWrite') and a 'Permissions' section. The 'Permissions' section contains four checkboxes: 'Registry read' (unchecked), 'Registry write' (checked), 'Service connect' (checked), and 'Device connect' (unchecked). A red box highlights the 'Permissions' section. At the bottom right of the dialog is a 'Create' button, which is also highlighted with a red box.

5. Select your new policy from the list of policies.
6. Under **Shared access keys**, select the copy icon for the **Connection string -- primary key** and save the value.

The screenshot shows the Azure IoT Hub Shared access policies blade. On the left, there's a navigation menu with options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Events, Settings, Shared access policies, Pricing and scale, IP Filter, Certificates, Built-in endpoints, Failover, Properties, Locks, and Export template. The Shared access policies item is selected. In the main area, there's a search bar and a list of existing policies: iothubowner, service, device, registryRead, registryReadWrite, and serviceAndRegistryReadWrite (which is highlighted with a red box). To the right, a detailed view of the 'serviceAndRegistryReadWrite' policy is shown. It has an 'Access policy name' of 'serviceAndRegistryReadWrite'. Under 'Permissions', 'Registry read' and 'Registry write' are checked, while 'Service connect' and 'Device connect' are unchecked. Under 'Shared access keys', both 'Primary key' and 'Secondary key' are listed with their respective connection strings, which are also highlighted with red boxes.

For more information about IoT Hub shared access policies and permissions, see [Access control and permissions](#).

Create a device identity and a module identity in IoT Hub

In this section, you create a Python service app that creates a device identity and a module identity in the identity registry in your IoT hub. A device or module can't connect to IoT hub unless it has an entry in the identity registry. For more information, see [Understand the identity registry in your IoT hub](#). 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. At your command prompt, run the following command to install the `azure-iot-hub` package:

```
pip install azure-iot-hub
```

2. At your command prompt, run the following command to install the `msrest` package. You need this package to catch `HTTPOperationError` exceptions.

```
pip install msrest
```

3. Using a text editor, create a file named `CreateModule.py` in your working directory.
4. Add the following code to your Python file. Replace `YourIoTHubConnectionString` with the connection string you copied in [Get the IoT hub connection string](#).

```

import sys
from msrest.exceptions import HttpOperationError
from azure.iot.hub import IoTHubRegistryManager

CONNECTION_STRING = "YourIoTHubConnectionString"
DEVICE_ID = "myFirstDevice"
MODULE_ID = "myFirstModule"

try:
    # RegistryManager
    iothub_registry_manager = IoTHubRegistryManager(CONNECTION_STRING)

    try:
        # CreateDevice - let IoT Hub assign keys
        primary_key = ""
        secondary_key = ""
        device_state = "enabled"
        new_device = iothub_registry_manager.create_device_with_sas(
            DEVICE_ID, primary_key, secondary_key, device_state
        )
    except HttpOperationError as ex:
        if ex.response.status_code == 409:
            # 409 indicates a conflict. This happens because the device already exists.
            new_device = iothub_registry_manager.get_device(DEVICE_ID)
        else:
            raise

    print("device <" + DEVICE_ID +
          "> has primary key = " + new_device.authentication.symmetric_key.primary_key)

    try:
        # CreateModule - let IoT Hub assign keys
        primary_key = ""
        secondary_key = ""
        managed_by = ""
        new_module = iothub_registry_manager.create_module_with_sas(
            DEVICE_ID, MODULE_ID, managed_by, primary_key, secondary_key
        )
    except HttpOperationError as ex:
        if ex.response.status_code == 409:
            # 409 indicates a conflict. This happens because the module already exists.
            new_module = iothub_registry_manager.get_module(DEVICE_ID, MODULE_ID)
        else:
            raise

    print("device/module <" + DEVICE_ID + "/" + MODULE_ID +
          "> has primary key = " + new_module.authentication.symmetric_key.primary_key)

except Exception as ex:
    print("Unexpected error {}".format(ex))
except KeyboardInterrupt:
    print("IoTHubRegistryManager sample stopped")

```

5. At your command prompt, run the following command:

```
python CreateModule.py
```

This app creates a device identity with ID **myFirstDevice** and a module identity with ID **myFirstModule** under device **myFirstDevice**. (If the device or module ID already exists in the identity registry, the code simply retrieves the existing device or module information.) The app displays the ID and primary key for each 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 [Understand the identity registry in your IoT hub](#).

Update the module twin using Python service SDK

In this section, you create a Python service app that updates the module twin desired properties.

1. At your command prompt, run the following command to install the **azure-iot-hub** package. You can skip this step if you installed the **azure-iot-hub** package in the previous section.

```
pip install azure-iot-hub
```

2. Using a text editor, create a file named **UpdateModuleTwinDesiredProperties.py** in your working directory.
3. Add the following code to your Python file. Replace *YourIoTHubConnectionString* with the connection string you copied in [Get the IoT hub connection string](#).

```
import sys
from azure.iot.hub import IoTHubRegistryManager
from azure.iot.hub.models import Twin, TwinProperties

CONNECTION_STRING = "YourIoTHubConnectionString"
DEVICE_ID = "myFirstDevice"
MODULE_ID = "myFirstModule"

try:
    # RegistryManager
    iothub_registry_manager = IoTHubRegistryManager(CONNECTION_STRING)

    module_twin = iothub_registry_manager.get_module_twin(DEVICE_ID, MODULE_ID)
    print ( "" )
    print ( "Module twin properties before update      :" )
    print ( "{0}".format(module_twin.properties) )

    # Update twin
    twin_patch = Twin()
    twin_patch.properties = TwinProperties(desired={"telemetryInterval": 122})
    updated_module_twin = iothub_registry_manager.update_module_twin(
        DEVICE_ID, MODULE_ID, twin_patch, module_twin.etag
    )
    print ( "" )
    print ( "Module twin properties after update      :" )
    print ( "{0}".format(updated_module_twin.properties) )

except Exception as ex:
    print ( "Unexpected error {0}.".format(ex) )
except KeyboardInterrupt:
    print ( "IoTHubRegistryManager sample stopped" )
```

Get updates on the device side

In this section, you create a Python app to get the module twin desired properties update on your device.

1. Get your module connection string. In [Azure portal](#), navigate to your IoT Hub and select **IoT devices** in the left pane. Select **myFirstDevice** from the list of devices and open it. Under **Module identities**, select **myFirstModule**. Copy the module connection string. You need it in a following step.

2. At your command prompt, run the following command to install the **azure-iot-device** package:

```
pip install azure-iot-device
```

3. Using a text editor, create a file named **ReceiveModuleTwinDesiredPropertiesPatch.py** in your working directory.
4. Add the following code to your Python file. Replace *YourModuleConnectionString* with the module connection string you copied in step 1.

```
import time
import threading
from azure.iot.device import IoTHubModuleClient

CONNECTION_STRING = "YourModuleConnectionString"

def twin_update_listener(client):
    while True:
        patch = client.receive_twin_desired_properties_patch()  # blocking call
        print("")
        print("Twin desired properties patch received:")
        print(patch)

def iothub_client_sample_run():
    try:
        module_client = IoTHubModuleClient.create_from_connection_string(CONNECTION_STRING)

        twin_update_listener_thread = threading.Thread(target=twin_update_listener, args=(module_client,))
        twin_update_listener_thread.daemon = True
        twin_update_listener_thread.start()

        while True:
            time.sleep(1000000)

    except KeyboardInterrupt:
        print("IoTHubModuleClient sample stopped")

if __name__ == '__main__':
    print( "Starting the IoT Hub Python sample..." )
    print( "IoTHubModuleClient waiting for commands, press Ctrl-C to exit" )

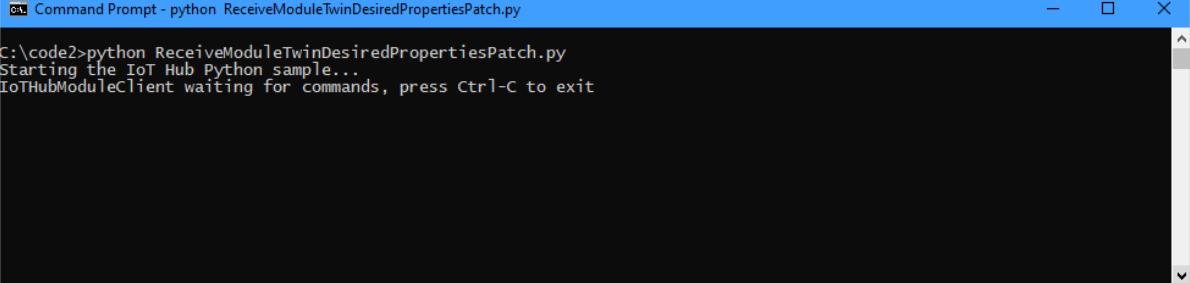
    iothub_client_sample_run()
```

Run the apps

In this section, you run the **ReceiveModuleTwinDesiredPropertiesPatch** device app and then run the **UpdateModuleTwinDesiredProperties** service app to update the desired properties of your module.

1. Open a command prompt and run the device app:

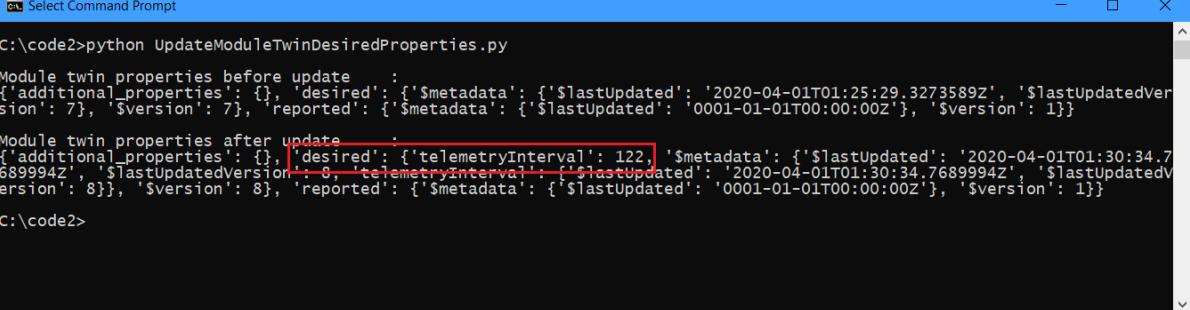
```
python ReceiveModuleTwinDesiredPropertiesPatch.py
```



2. Open a separate command prompt and run the service app:

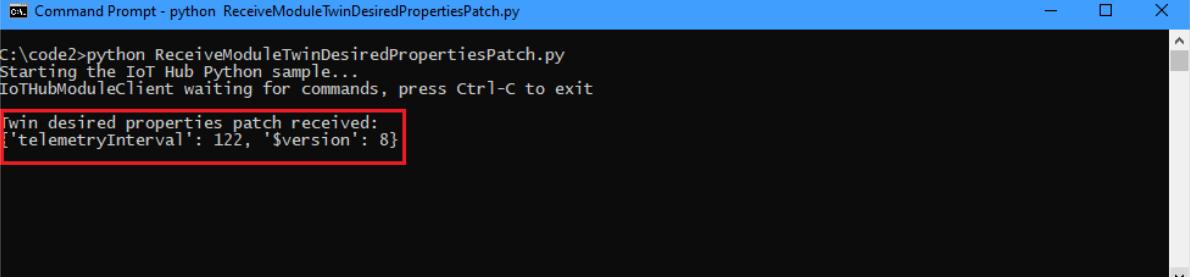
```
python UpdateModuleTwinDesiredProperties.py
```

Notice that the **TelemetryInterval** desired property appears in the updated module twin in your service app output:



```
C:\code2>python UpdateModuleTwinDesiredProperties.py
Module twin properties before update :
{"additional_properties": {}, "desired": {"$metadata": {"$lastUpdated": "2020-04-01T01:25:29.3273589Z", "$lastUpdatedVersion": 7}, "$version": 7}, "reported": {"$metadata": {"$lastUpdated": "0001-01-01T00:00:00Z"}, "$version": 1}}
Module twin properties after update :
{"additional_properties": {}, "desired": {"$metadata": {"$lastUpdated": "2020-04-01T01:30:34.7689994Z", "$lastUpdatedVersion": 8}, "telemetryInterval": 122, "version": 8}, "reported": {"$metadata": {"$lastUpdated": "0001-01-01T00:00:00Z"}, "version": 1}}
C:\code2>
```

The same property appears in the desired properties patch received in your device app output:



```
C:\code2>python ReceiveModuleTwinDesiredPropertiesPatch.py
Starting the IoT Hub Python sample...
IoTHubModuleClient waiting for commands, press Ctrl-C to exit
Twin desired properties patch received:
{'telemetryInterval': 122, '$version': 8}
```

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 (C)

4/21/2020 • 10 minutes to read • [Edit Online](#)

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](#).

Prerequisites

- An active Azure account. (If you don't have an account, you can create an [Azure free account](#) in just a couple of minutes.)
- The latest [Azure IoT C SDK](#).

Create an IoT hub

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

1. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription**: Select the subscription to use for your hub.
 - **Resource Group**: Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region**: Select the region in which you want your hub to be located. Select the location closest to

you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.

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

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

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 that is the title 'IoT hub' with the Microsoft logo. A red box highlights the 'Project details' section, which includes fields for Subscription, Resource group, Region, and IoT hub name. The 'Subscription' field is set to 'Personal IoT items'. The 'Resource group' field has a dropdown menu with 'Create new' option. The 'Region' field is set to 'East Asia'. The 'IoT hub name' field contains the placeholder text 'Once your hub is created, this name can't be changed'. At the bottom of the form, there are navigation buttons: 'Review + create' (blue), '< Previous' (disabled), 'Next: Size and scale >' (highlighted with a red box), and 'Automation options'.

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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
 Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center [On](#) Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.

If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure](#)

resources.

The screenshot shows the 'Tags' section of the IoT hub configuration. It displays a table with two rows. The first row has 'Name' as 'department' and 'Value' as 'accounting'. The second row is empty. Below the table are navigation buttons: 'Review + create' (highlighted), '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' page for the IoT hub. It lists configuration details under three sections: Basics, Size and scale, and Tags. The 'Review + create' tab is highlighted with a red box. At the bottom, the 'Create' button is also highlighted with a red box.

| Section | Setting | Value |
|----------------|----------------------------|--------------------------------|
| Basics | Subscription | Personal testing |
| | Resource group | iot-hubs |
| | Region | West US 2 |
| | IoT hub name | you-hub-name |
| 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 |
| | Azure Security Center | 0.001 USD per device per month |
| | Tags | |
| Tags | department:accounting | |

8. Select **Create** to create your new hub. Creating the hub takes a few minutes.

Get the IoT hub connection string

In this article, you create a back-end service that adds a device in the identity registry and then adds a module to

that device. Your service requires the **registry write** permission. By default, every IoT hub is created with a shared access policy named **registryReadWrite** that grants this permission.

To get the IoT Hub connection string for the **registryReadWrite** policy, follow these steps:

1. In the [Azure portal](#), select **Resource groups**. Select the resource group where your hub is located, and then select your hub from the list of resources.
2. On the left-side pane of your hub, select **Shared access policies**.
3. From the list of policies, select the **registryReadWrite** policy.
4. Under **Shared access keys**, select the copy icon for the **Connection string -- primary key** and save the value.

The screenshot shows the Azure portal interface for managing an IoT hub's shared access policies. On the left, the 'Shared access policies' section is selected. A specific policy, 'registryReadWrite', is highlighted with a red box. On the right, the details for this policy are shown, including its permissions (Registry read and Registry write checked) and its shared access keys. The 'Primary key' field is also highlighted with a red box, indicating it is the connection string to be copied.

For more information about IoT Hub shared access policies and permissions, see [Access control and permissions](#).

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.deviceId = deviceId;
    deviceCreateInfo.registrationType = IOTHUB_REGISTRY_DEVICE_REGISTRATION_TYPE_PRIMARY;
    deviceCreateInfo.authenticationMethod =
        IOTHUB_REGISTRY_DEVICE_AUTHENTICATION_METHOD_X509;
    deviceCreateInfo.x509Thumbprint =
        "AQAB..."; // replace with your thumbprint
    deviceCreateInfo.x509ThumbprintSize = 128;
}
```

```

    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_DEVICE_EXIST)
    {
        (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.

Module Identity Twin Direct Method Delete

| | |
|---------------------------------|-----------------------------|
| Module Identity Name | myFirstDevice/myFirstModule |
| Primary key | [REDACTED] |
| Secondary key | [REDACTED] |
| Connection string—primary key | [REDACTED] |
| Connection string—secondary key | [REDACTED] |

2. Create UpdateModuleTwinReportedProperties app

Add the following 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_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"

```

```

#ifndef _IOTHUB_MODULE_CLIENT_LL_H_
#define _IOTHUB_MODULE_CLIENT_LL_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

#ifdef SAMPLE_MQTT
    #include "iothubtransportmqtt.h"
#endif // SAMPLE_MQTT
#ifdef SAMPLE_MQTT_OVER_WEBSOCKETS
    #include "iothubtransportmqtt_websockets.h"
#endif // SAMPLE_MQTT_OVER_WEBSOCKETS
#ifdef SAMPLE_AMQP
    #include "iothubtransportamqp.h"
#endif // SAMPLE_AMQP
#ifdef SAMPLE_AMQP_OVER_WEBSOCKETS
    #include "iothubtransportamqp_websockets.h"
#endif // SAMPLE_AMQP_OVER_WEBSOCKETS
#ifdef 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",
           MU_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
#ifdef SAMPLE_MQTT
    protocol = MQTT_Protocol;
#endif // SAMPLE_MQTT
#ifdef SAMPLE_MQTT_OVER_WEBSOCKETS
    protocol = MQTT_WebSocket_Protocol;
#endif // SAMPLE_MQTT_OVER_WEBSOCKETS
#ifdef SAMPLE_AMQP
    protocol = AMQP_Protocol;

```

```

protocol = AMQP_Over_WebSockets;
#endif // SAMPLE_AMQP
#ifndef SAMPLE_AMQP_OVER_WEBSOCKETS
    protocol = AMQP_Protocol_over_WebSocketsTls;
#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 (Node.js)

4/21/2020 • 9 minutes to read • [Edit Online](#)

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](#).

Prerequisites

- Node.js version 10.0.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. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoTHub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription**: Select the subscription to use for your hub.
 - **Resource Group**: Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).

- **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.
- **IoT Hub Name:** Enter a name for your hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

The screenshot shows the 'Basics' tab of the IoT Hub creation wizard. It includes fields for Subscription, Resource group, Region, and IoT hub name, all of which are highlighted with a red border. At the bottom, there are navigation buttons: 'Review + create' (blue), '< Previous' (disabled), 'Next: Size and scale >' (highlighted with a red border), and 'Automation options'.

Home > New > IoT hub

IoT hub
Microsoft

Basics [Size and scale](#) [Tags](#) [Review + create](#)

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

Project details

Choose the subscription you'll use to manage deployments and costs. Use resource groups like folders to help you organize and manage resources.

Subscription * ⓘ

Resource group * ⓘ [Create new](#)

Region * ⓘ

IoT hub name * ⓘ

[Review + create](#) < Previous [Next: Size and scale >](#) Automation options

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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 [Determines how your IoT hub can scale. You can change this later if your needs increase.](#)

Azure Security Center [On](#) Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.
If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure](#)

resources.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags Review + create

Tags are name/value pairs. To categorize resources and consolidate billing, apply the same tag to multiple resources and resource groups. Your tags will update automatically if you change your resources. [Learn more](#)

| Name ⓘ | Value ⓘ | Resource |
|------------|--------------|----------|
| department | : accounting | IoT Hub |
| | : | IoT Hub |

[Review + create](#) < Previous: Size and scale Next: Review + create > Automation options

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags Review + create

Basics

Subscription Personal testing

Resource group iot-hubs

Region West US 2

IoT hub name you-hub-name

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

Azure Security Center 0.001 USD per device per month

Tags

department accounting

[Create](#) < Previous: Tags Next > Automation options

8. Select **Create** to create your new hub. Creating the hub takes a few minutes.

Get the IoT hub connection string

In this article, you create a back-end service that adds a device in the identity registry and then adds a module to

that device. Your service requires the **registry write** permission. By default, every IoT hub is created with a shared access policy named **registryReadWrite** that grants this permission.

To get the IoT Hub connection string for the **registryReadWrite** policy, follow these steps:

1. In the [Azure portal](#), select **Resource groups**. Select the resource group where your hub is located, and then select your hub from the list of resources.
2. On the left-side pane of your hub, select **Shared access policies**.
3. From the list of policies, select the **registryReadWrite** policy.
4. Under **Shared access keys**, select the copy icon for the **Connection string -- primary key** and save the value.

The screenshot shows the Azure portal interface for managing an IoT hub. On the left, the navigation pane includes options like Overview, Activity log, Access control (IAM), Tags, Events, Settings, and Shared access policies. The 'Shared access policies' option is highlighted with a red box. The main content area displays a table of policies and their permissions. One row is selected, showing the 'registryReadWrite' policy with the following details:

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

On the right, a detailed view of the 'registryReadWrite' policy shows the 'Access policy name' as 'registryReadWrite'. Under 'Permissions', 'Registry write' is checked. The 'Shared access keys' section contains two keys: 'Primary key' and 'Secondary key', each with a copy icon. The 'Connection string--primary key' copy icon is highlighted with a red box.

For more information about IoT Hub shared access policies and permissions, see [Access control and permissions](#).

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.

Run this using node add.js. It will give you a connection string for your device identity and another one for your module identity.

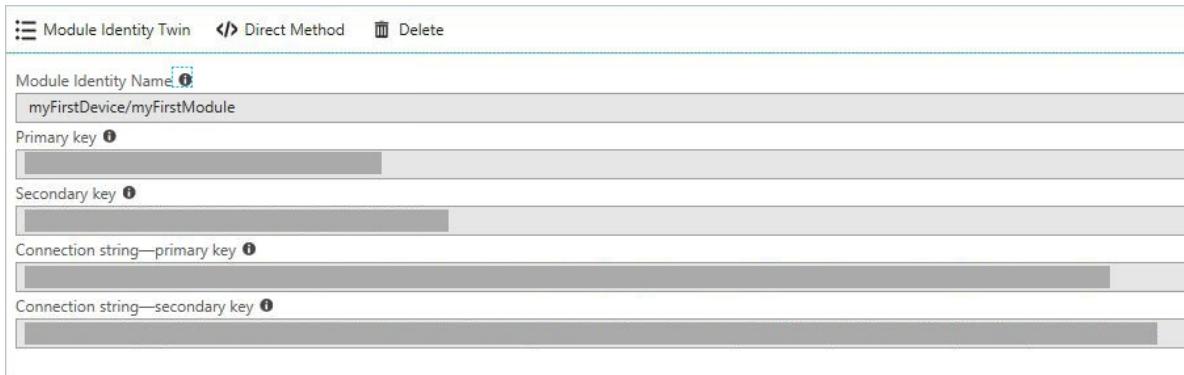
NOTE

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** -- Sign in to the [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
```

You will then see:

```

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.js)

4/21/2020 • 11 minutes to read • [Edit Online](#)

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 available only in the standard tier of IoT Hub. For more information about the basic and standard/free IoT Hub tiers, see [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.

Prerequisites

- Node.js version 10.0.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.)
- Make sure that port 8883 is open in your firewall. The device sample in this article uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT](#)

Hub (MQTT).

Create an IoT hub

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

1. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter **IoT Hub** in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.
 - **IoT Hub Name:** Enter a name for your hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

The screenshot shows the 'Basics' tab of the IoT hub creation wizard. It includes fields for Subscription (Personal IoT items), Resource group (Create new), Region (East Asia), and IoT hub name (Once your hub is created, this name can't be changed). The 'Review + create' and 'Automation options' buttons are at the bottom.

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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](#)

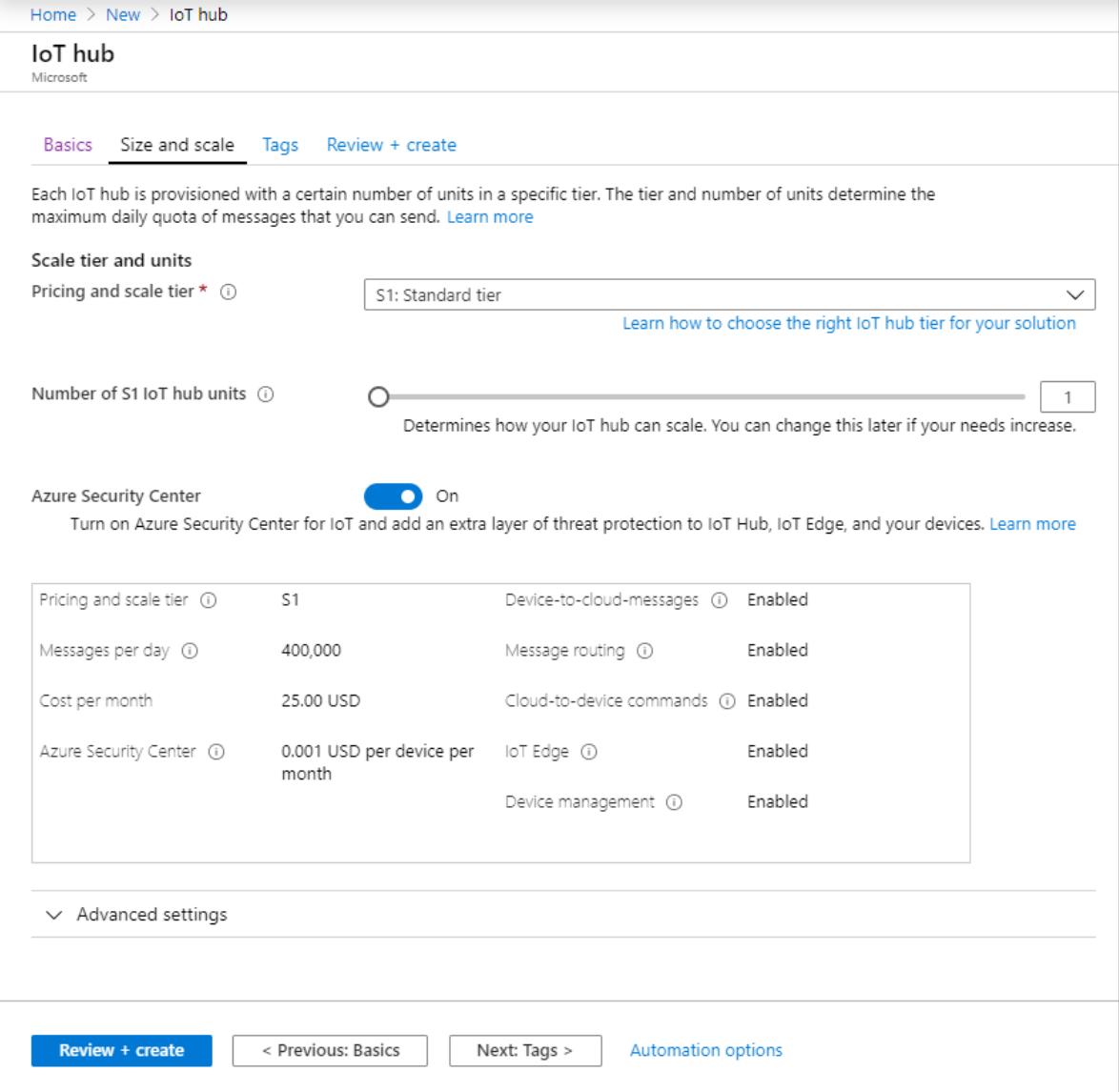
Number of S1 IoT hub units ⓘ 1
Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center On
Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)



You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.
If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure](#)

resources.

The screenshot shows the 'Tags' section of the Azure portal for creating an IoT hub. The table contains one tag entry:

| Name | Value | Resource |
|------------|------------|----------|
| department | accounting | IoT Hub |
| | | IoT Hub |

Below the table are navigation buttons: 'Review + create' (highlighted in blue), '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' screen for the IoT hub creation. The 'Review + create' tab is highlighted with a red box. The page displays the following configuration details:

| Basics | Size and scale | Tags |
|--------------------------------|-------------------------------|---|
| Subscription: Personal testing | Number of S1 IoT hub units: 1 | IoT hub name: you-hub-name |
| Resource group: iot-hubs | Pricing and scale tier: S1 | Region: West US 2 |
| Region: West US 2 | Messages per day: 400,000 | |
| IoT hub name: you-hub-name | Cost per month: 25.00 USD | Azure Security Center: 0.001 USD per device per month |
| Tags | department: accounting | |

At the bottom, the 'Create' button is highlighted with a red box.

8. Select **Create** to create your new hub. Creating the hub takes a few minutes.

Register a new device in the IoT hub

In this section, you use the Azure CLI to create a device identity for this article. Device IDs are case sensitive.

1. Open [Azure Cloud Shell](#).
2. In Azure Cloud Shell, run the following command to install the Microsoft Azure IoT Extension for Azure CLI:

```
az extension add --name azure-iot
```

3. 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
- 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 `{yourdeviceconnectionstring}` placeholder value with the device connection string you copied previously in [Register a new device in the IoT hub](#).

```
var connectionString = '{yourdeviceconnectionstring}';  
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 IotHub client');  
    } else {  
        console.log('Client opened. Waiting for reboot method.');//  
        client.onDeviceMethod('reboot', onReboot);  
    }  
});
```

8. 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](#).

Get the IoT hub connection string

In this article, you create a backend service that invokes a direct method on a device. To invoke a direct method on a device through IoT Hub, your service needs the **service connect** permission. By default, every IoT Hub is created with a shared access policy named **service** that grants this permission.

To get the IoT Hub connection string for the **service** policy, follow these steps:

1. In the [Azure portal](#), select **Resource groups**. Select the resource group where your hub is located, and then select your hub from the list of resources.
2. On the left-side pane of your IoT hub, select **Shared access policies**.
3. From the list of policies, select the **service** policy.
4. Under **Shared access keys**, select the copy icon for the **Connection string -- primary key** and save the value.

The screenshot shows the Azure portal interface for managing IoT Hub shared access policies. On the left, the navigation menu includes options like Overview, Activity log, Access control (IAM), Tags, Events, Settings, and Shared access policies (which is highlighted with a red box). The main content area displays the 'Shared access policies' page for the 'contoso-hub-1' IoT Hub. It lists several policies: 'iothubowner' (permissions: registry write, service connect), 'service' (selected, permissions: service connect), 'device' (permissions: device connect), 'registryRead' (permissions: registry read), and 'registryReadWrite' (permissions: registry read/write). On the right, a detailed view of the 'service' policy is shown, including its name ('service'), access policy name ('service'), and permissions (Registry read, Service connect, Device connect). Below this, the 'Shared access keys' section is displayed, showing the 'Primary key' and 'Connection string--primary key' (which is also highlighted with a red box). There are copy icons next to each key.

For more information about IoT Hub shared access policies and permissions, see [Access control and permissions](#).

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.

1. 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
```

2. 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
```

3. Using a text editor, create a **dmpatterns_getstarted_service.js** file in the **triggerrebootondevice** folder.
4. 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;
```

5. Add the following variable declarations and replace the `{iothubconnectionstring}` placeholder value with the IoT hub connection string you copied previously in [Get the IoT hub connection string](#):

```
var connectionString = '{iothubconnectionstring}';
var registry = Registry.fromConnectionString(connectionString);
var client = Client.fromConnectionString(connectionString);
var deviceToReboot = 'myDeviceId';
```

6. 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.");
        }
    });
};
```

7. 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.');
    });
};
```

8. 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);
```

9. Save and close the `dmpatterns_getstarted_service.js` file.

Run the apps

You're 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.

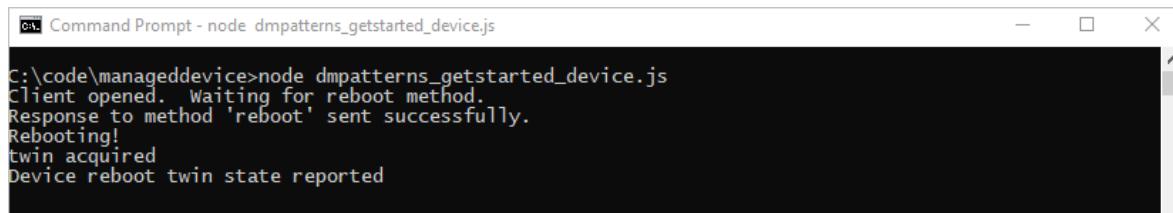
```
node dmpatterns_getstarted_device.js
```

- 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
```

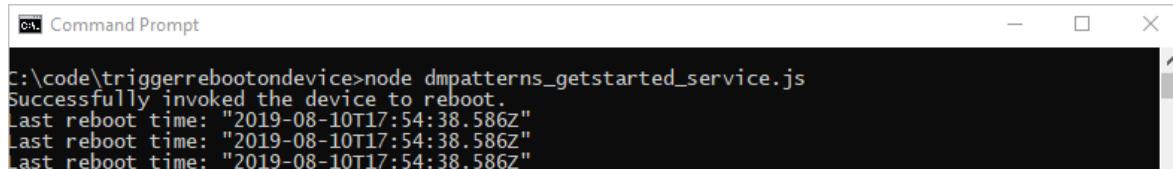
- You see the device response to the reboot direct method and the reboot status in the console.

The following shows the device response to the reboot direct method sent by the service:



```
C:\ Command Prompt - node dmpatterns_getstarted_device.js
C:\code\manageddevice>node dmpatterns_getstarted_device.js
Client opened. Waiting for reboot method.
Response to method 'reboot' sent successfully.
Rebooting!
twin acquired
Device reboot twin state reported
```

The following shows the service triggering the reboot and polling the device twin for the last reboot time:



```
C:\ Command Prompt
C:\code\triggerrebootondevice>node dmpatterns_getstarted_service.js
Successfully invoked the device to reboot.
Last reboot time: "2019-08-10T17:54:38.586Z"
Last reboot time: "2019-08-10T17:54:38.586Z"
Last reboot time: "2019-08-10T17:54:38.586Z"
```

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)

4/21/2020 • 12 minutes to read • [Edit Online](#)

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 available only in the standard tier of IoT Hub. For more information about the basic and standard/free IoT Hub tiers, see [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**. This app 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**. This app calls a direct method in the simulated device app, displays the response, and displays the updated reported properties.

Prerequisites

- Visual Studio.
- An active Azure account. If you don't have an account, you can create a [free account](#) in just a couple of minutes.
- Make sure that port 8883 is open in your firewall. The device sample in this article uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

Create an IoT hub

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

1. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.
 - **IoT Hub Name:** Enter a name for your hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

The screenshot shows the 'Basics' tab of the IoT hub creation wizard. At the top, there's a breadcrumb navigation: Home > New > IoT hub. The title 'IoT hub' is followed by 'Microsoft'. Below the tabs, there's a brief description: 'Create an IoT Hub to help you connect, monitor, and manage billions of your IoT assets.' A 'Learn more' link is provided. The 'Project details' section contains four required fields: 'Subscription *' (set to 'Personal IoT items'), 'Resource group *' (with a dropdown for 'Create new'), 'Region *' (set to 'East Asia'), and 'IoT hub name *' (set to 'Once your hub is created, this name can't be changed'). At the bottom of the page, there are navigation buttons: 'Review + create' (in a blue box), '< Previous', 'Next: Size and scale >' (highlighted with a red box), and 'Automation options'.

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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
Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center On
Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.

If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure](#)

resources.

The screenshot shows the 'Tags' step in the IoT hub creation wizard. It displays a table of resource tags:

| Name | Value | Resource |
|------------|------------|----------|
| department | accounting | IoT Hub |
| | | IoT Hub |

Below the table are navigation buttons: 'Review + create' (highlighted in blue), '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' step in the IoT hub creation wizard. The 'Review + create' tab is highlighted with a red box. The page displays the following configuration details:

| Category | Setting | Value |
|----------------|----------------------------|--------------------------------|
| Basics | Subscription | Personal testing |
| | Resource group | iot-hubs |
| | Region | West US 2 |
| | IoT hub name | you-hub-name |
| 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 |
| | Azure Security Center | 0.001 USD per device per month |
| | Tags | |
| Tags | department:accounting | |

At the bottom are buttons: 'Create' (highlighted with a red box), '< Previous: Tags', 'Next >', and 'Automation options'.

8. Select **Create** to create your new hub. Creating the hub takes a few minutes.

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 a hub

unless it has an entry in the identity registry. For more information, see the [IoT Hub developer guide](#).

1. In your IoT hub navigation menu, open **IoT Devices**, then select **New** to add a device in your IoT hub.

The screenshot shows the Azure IoT Hub management interface for the 'iot-hub-contoso-one' hub. The top navigation bar includes 'Home', 'All resources', and the specific hub name. Below the navigation is a search bar and a toolbar with 'New' (highlighted), 'Refresh', and 'Delete' buttons. A message states: 'View, create, delete, and update devices in your IoT Hub.' To the right is a query editor with fields for 'Field' (set to 'select or enter a property name'), 'Operator' (set to '='), and 'Value' (set to 'specify constraint value'). Below the editor is a 'Query devices' button and a link to 'Switch to query editor'. The main table area has columns for 'DEVICE ID', 'STATUS', 'LAST ACTIVITY TIME (UTC)', 'LAST STATUS UPDATE (UTC)', 'AUTHENTICATION T...', and 'CLOUD ...'. A message 'No results' is displayed. The left sidebar contains sections for Overview, Activity log, Access control (IAM), Tags, Events, Settings (with Shared access policies, Pricing and scale, IP Filter, Certificates, Built-in endpoints, Manual failover (preview), Properties, Locks, Export template), Explorers (Query explorer, IoT devices - highlighted with a red box), and Automatic Device Management.

2. In **Create a device**, provide a name for your new device, such as **myDeviceId**, and select **Save**. This action creates a device identity for your IoT hub.

Home > All resources > iot-hub-contoso-one - IoT devices > Create a device

Create a device

Find Certified for Azure IoT devices in the Device Catalog

*** Device ID** myDeviceId

Authentication type: Symmetric key

*** Primary key**

*** Secondary key**

Auto-generate keys:

Connect this device to an IoT hub: Enable

Parent device: No 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 **Primary Connection String** to use later.

Home > iot-hub-contoso-one - IoT devices > myDeviceId

| Device ID | myDeviceId | | |
|--|---|-------------------------------------|--------------------------|
| Primary Key | H2Awv1PN3suNBkaiQU1UeEINB3j0= | | |
| Secondary Key | G7615rzcbyWFzcfIlgmad55lGVa4I= | | |
| Primary Connection String | HostName=iot-hub-contoso-one.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=QdSim6i7cptUCeMYGVSeIRKOV2ZGFSJpbmyklVYM9df= | | |
| Secondary Connection String | HostName=iot-hub-contoso-one.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=q32oiXuwifEXbbqKYkjv8sF82qZInqzGZspqkl2nqz= | | |
| Enable connection to IoT Hub | <input checked="" type="radio"/> Enable <input type="radio"/> Disable | | |
| Parent device | No parent device | | |
| Module Identities Configurations | | | |
| MODULE ID | CONNECTION STATE | CONNECTION STATE LAST UPDATED (UTC) | LAST ACTIVITY TIME (UTC) |
| There are no module identities for this 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](#).

Get the IoT hub connection string

In this article, you create a backend service that invokes a direct method on a device. To invoke a direct method on a device through IoT Hub, your service needs the **service connect** permission. By default, every IoT Hub is created with a shared access policy named **service** that grants this permission.

To get the IoT Hub connection string for the **service** policy, follow these steps:

1. In the [Azure portal](#), select **Resource groups**. Select the resource group where your hub is located, and then select your hub from the list of resources.
2. On the left-side pane of your IoT hub, select **Shared access policies**.
3. From the list of policies, select the **service** policy.
4. Under **Shared access keys**, select the copy icon for the **Connection string -- primary key** and save the value.

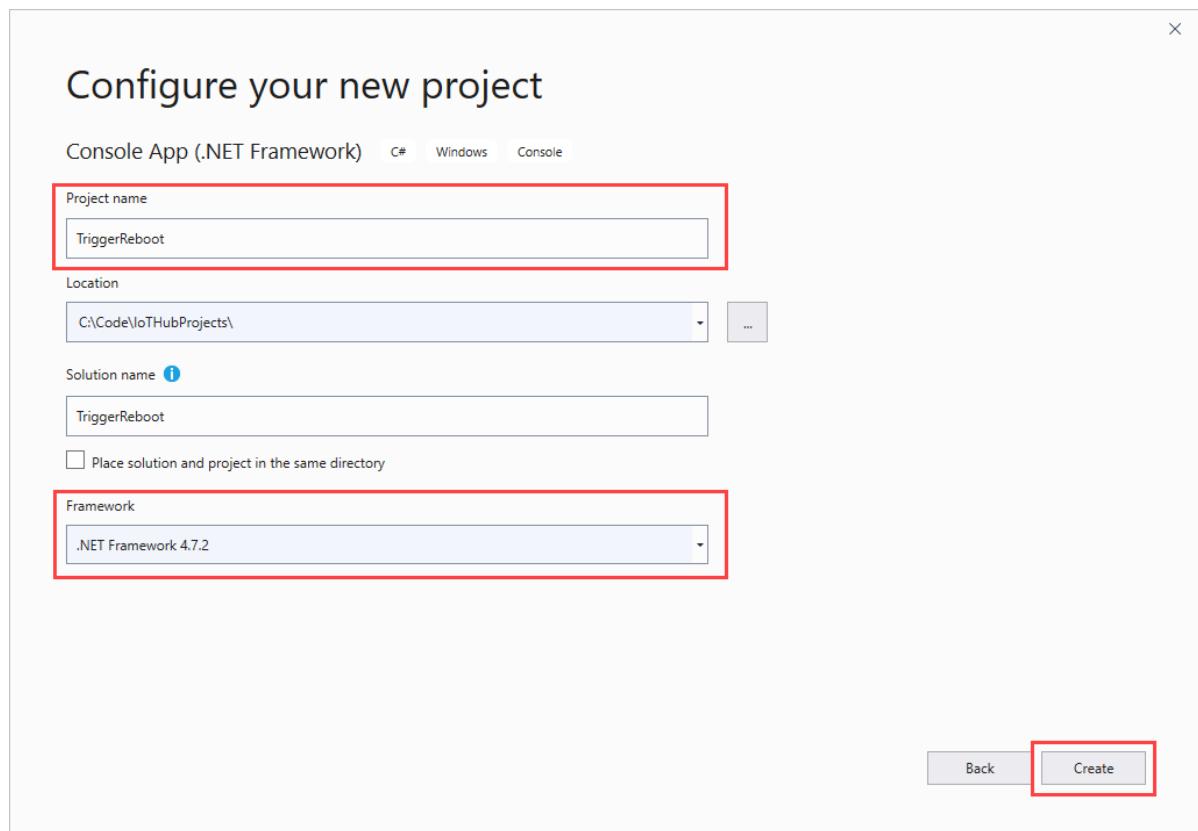
The screenshot shows the Azure portal interface for managing an IoT hub's shared access policies. On the left, there's a sidebar with various options like Overview, Activity log, Access control (IAM), Tags, Events, Pricing and scale, IP Filter, Certificates, Built-in endpoints, Manual failover (preview), Properties, Locks, and Export template. The 'Shared access policies' option is highlighted with a red box. The main content area shows a table of policies. The 'service' policy is selected and also highlighted with a red box. The table has two columns: POLICY and PERMISSIONS. The 'service' policy has 'registry write, service connect' permissions. Below the table, there's a section for 'Shared access keys' with fields for 'Primary key' and 'Secondary key', both with copy icons. The 'Connection string--primary key' field is also highlighted with a red box.

For more information about IoT Hub shared access policies and permissions, see [Access control and permissions](#).

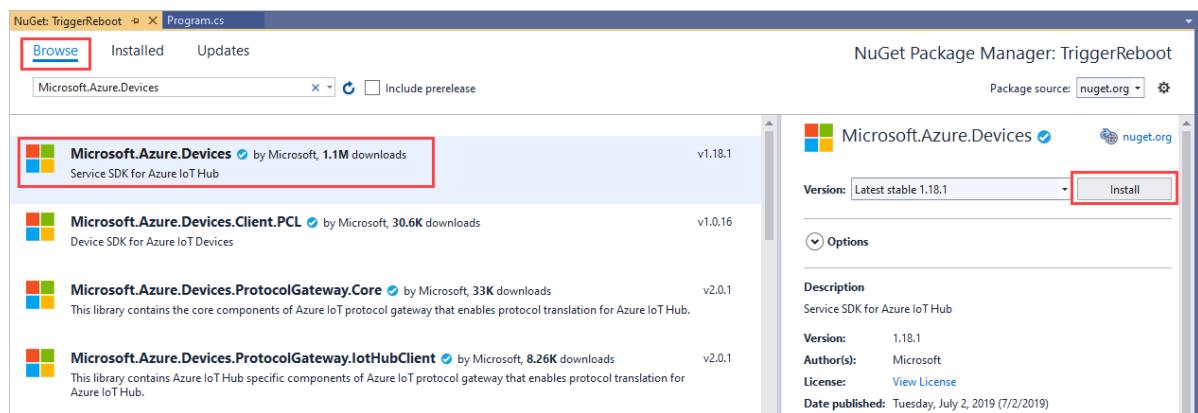
Trigger a remote reboot on the device using a direct method

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, select **Create a new project**.
2. In **Create a new project**, find and select the **Console App (.NET Framework)** project template, and then select **Next**.
3. In **Configure your new project**, name the project *TriggerReboot*, and select .NET Framework version 4.5.1 or later. Select **Create**.



4. In Solution Explorer, right-click the **TriggerReboot** project, and then select **Manage NuGet Packages**.
5. Select **Browse**, then search for and select **Microsoft.Azure.Devices**. Select **Install** to install the **Microsoft.Azure.Devices** package.



This step downloads, installs, and adds a reference to the [Azure IoT service SDK](#) NuGet package and its dependencies.

6. Add the following `using` statements at the top of the **Program.cs** file:

```
using Microsoft.Azure.Devices;
using Microsoft.Azure.Devices.Shared;
```

7. Add the following fields to the **Program** class. Replace the `{iot hub connection string}` placeholder value with the IoT Hub connection string you copied previously in [Get the IoT hub connection string](#).

```
static RegistryManager registryManager;
static string connString = "{iot hub connection string}";
static ServiceClient client;
static string targetDevice = "myDeviceId";
```

8. 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.ToString());
}
```

9. 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.");
}
```

10. 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();
```

11. Select **Build > Build 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

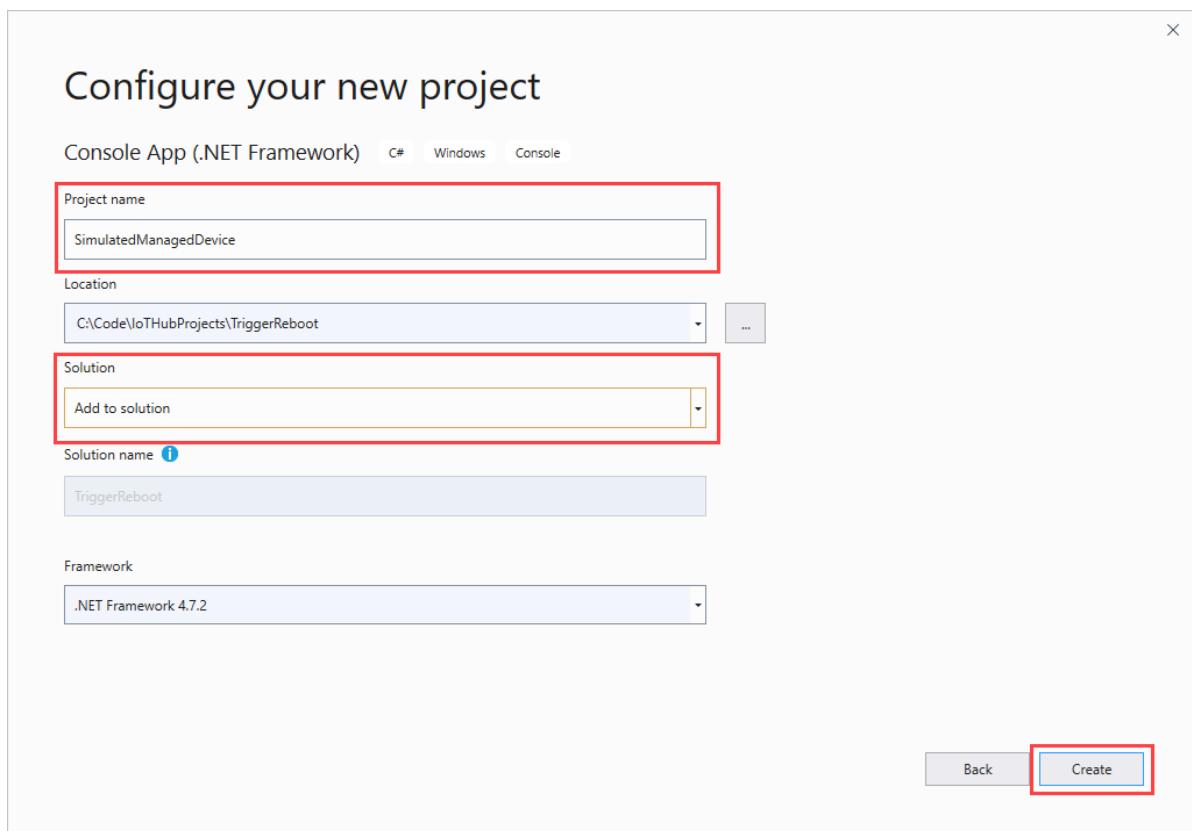
In this section, you:

- 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.

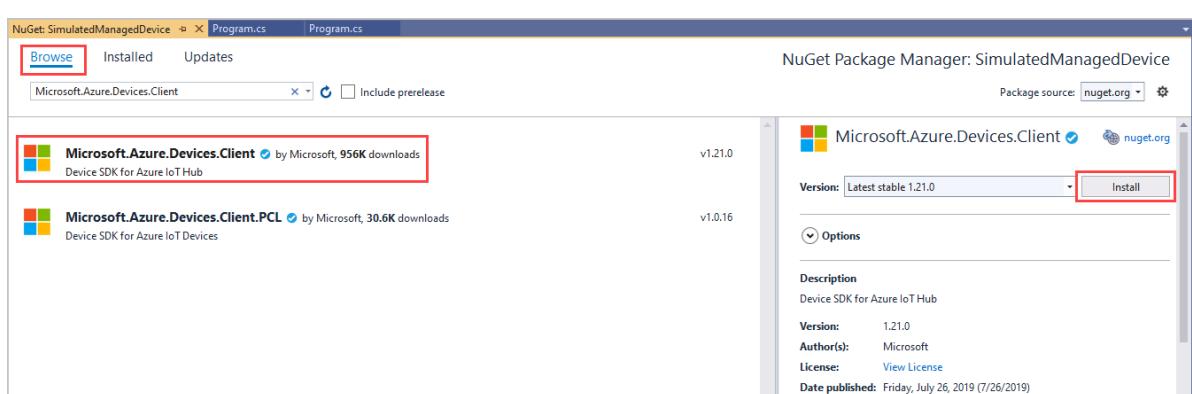
To create the simulated device app, follow these steps:

1. In Visual Studio, in the **TriggerReboot** solution you already created, select **File > New > Project**. In **Create a new project**, find and select the **Console App (.NET Framework)** project template, and then select **Next**.
2. In **Configure your new project**, name the project *SimulateManagedDevice*, and for **Solution**, select **Add**

to solution. Select **Create**.



3. In Solution Explorer, right-click the new **SimulateManagedDevice** project, and then select **Manage NuGet Packages**.
4. Select **Browse**, then search for and select **Microsoft.Azure.Devices.Client**. Select **Install**.



This step downloads, installs, and adds a reference to the [Azure IoT device SDK](#) NuGet package and its dependencies.

5. Add the following `using` statements at the top of the **Program.cs** file:

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

6. Add the following fields to the **Program** class. Replace the `{device connection string}` placeholder value with the device connection string that you noted previously in [Register a new device in the IoT hub](#).

```
static string DeviceConnectionString = "{device connection string}";
static DeviceClient Client = null;
```

7. 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 = @""{"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 Solution Explorer, right-click your solution, and then select **Set StartUp Projects**.
- For **Common Properties > Startup Project**, Select **Single startup project**, and then select the **SimulateManagedDevice** project. Select **OK** to save your changes.
- Select **Build > Build Solution**.

NOTE

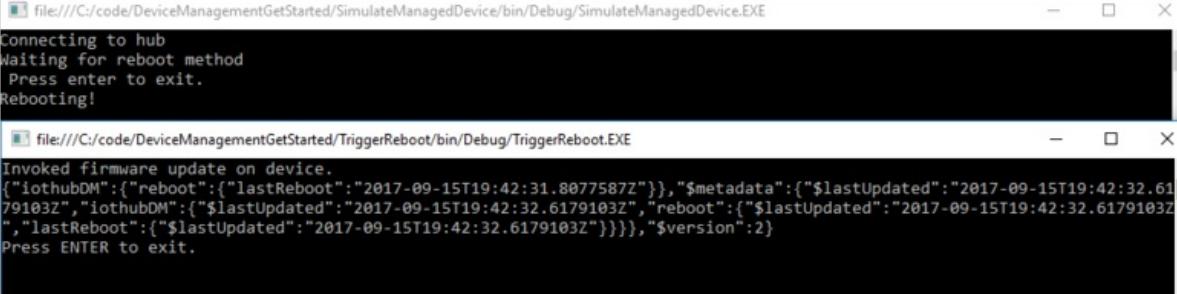
To keep things simple, this tutorial does not implement any retry policy. In production code, you should implement retry policies (such as an exponential backoff), as suggested in [Transient fault handling](#).

Run the apps

You're now ready to run the apps.

1. To run the .NET device app **SimulateManagedDevice**, in Solution Explorer, right-click the **SimulateManagedDevice** project, select **Debug**, and then select **Start new instance**. The app should start listening for method calls from your IoT hub.
2. After that the device is connected and waiting for method invocations, 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 left window, titled 'file:///C:/code/DeviceManagementGetStarted/SimulateManagedDevice/bin/Debug/SimulateManagedDevice.EXE', displays the following text:
Connecting to hub
Waiting for reboot method
Press enter to exit.
Rebooting!

The right window, titled 'file:///C:/code/DeviceManagementGetStarted/TriggerReboot/bin/Debug/TriggerReboot.EXE', displays the following text:
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)

7/29/2020 • 15 minutes to read • [Edit Online](#)

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 available only in the standard tier of IoT Hub. For more information about the basic and standard/free IoT Hub tiers, see [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](#).

Prerequisites

- [Java SE Development Kit 8](#). Make sure you select **Java 8** under **Long-term support** to get to downloads for JDK 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.)
- Make sure that port 8883 is open in your firewall. The device sample in this article uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

Create an IoT hub

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

1. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.
 - **IoT Hub Name:** Enter a name for your hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

Home > New > IoT hub

IoT hub

Microsoft

[X](#)

[Basics](#) [Size and scale](#) [Tags](#) [Review + create](#)

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

Project details

Choose the subscription you'll use to manage deployments and costs. Use resource groups like folders to help you organize and manage resources.

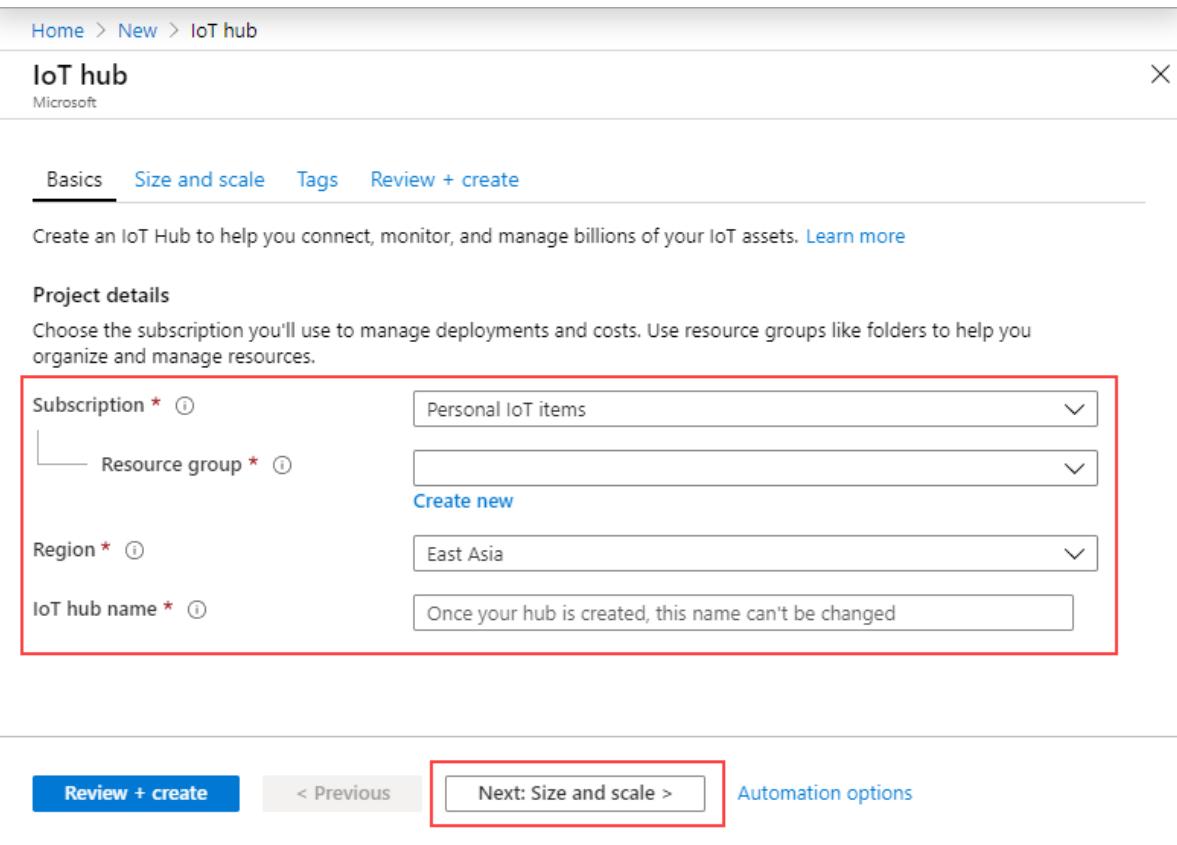
Subscription * ⓘ Personal IoT items

Resource group * ⓘ [Create new](#)

Region * ⓘ East Asia

IoT hub name * ⓘ Once your hub is created, this name can't be changed

[Review + create](#) [< Previous](#) [Next: Size and scale >](#) [Automation options](#)



5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics **Size and scale** Tags 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 [Determines how your IoT hub can scale. You can change this later if your needs increase.](#)

Azure Security Center [On](#) Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.
If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure](#)

resources.

The screenshot shows the 'Tags' tab selected in the top navigation bar. Below it, a note says: 'Tags are name/value pairs. To categorize resources and consolidate billing, apply the same tag to multiple resources and resource groups. Your tags will update automatically if you change your resources.' A table lists a single tag entry:

| Name | Value | Resource | Actions |
|------------|------------|----------|-----------------------------|
| department | accounting | IoT Hub | ... Delete |
| | | IoT Hub | |

At the bottom, there are buttons for 'Review + create', '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' step. It displays the configuration details for the IoT hub, including:

- Basics**:
 - Subscription: Personal testing
 - Resource group: iot-hubs
 - Region: West US 2
 - IoT hub name: you-hub-name
- 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
 - Azure Security Center: 0.001 USD per device per month
- Tags**:
 - department:accounting

At the bottom, there are buttons for 'Create' (highlighted with a red box), '< Previous: Tags', 'Next >', and 'Automation options'.

8. Select **Create** to create your new hub. Creating the hub takes a few minutes.

Register a new device in the IoT hub

In this section, you use the Azure CLI to create a device identity for this article. Device IDs are case sensitive.

1. Open [Azure Cloud Shell](#).
2. In Azure Cloud Shell, run the following command to install the Microsoft Azure IoT Extension for Azure CLI:

```
az extension add --name azure-iot
```

3. 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.

Get the IoT hub connection string

In this article, you create a backend service that invokes a direct method on a device. To invoke a direct method on a device through IoT Hub, your service needs the **service connect** permission. By default, every IoT Hub is created with a shared access policy named **service** that grants this permission.

To get the IoT Hub connection string for the **service** policy, follow these steps:

1. In the [Azure portal](#), select **Resource groups**. Select the resource group where your hub is located, and then select your hub from the list of resources.
2. On the left-side pane of your IoT hub, select **Shared access policies**.
3. From the list of policies, select the **service** policy.
4. Under **Shared access keys**, select the copy icon for the **Connection string -- primary key** and save the value.

The screenshot shows the Azure portal interface for managing shared access policies in an IoT hub. The left sidebar lists various settings like Overview, Activity log, Access control (IAM), Tags, Events, Pricing and scale, IP Filter, Certificates, Built-in endpoints, Manual failover (preview), Properties, Locks, and Export template. The 'Shared access policies' option is highlighted with a red box. The main content area shows a table of policies. The 'service' policy is selected and highlighted with a red box. The table columns are POLICY and PERMISSIONS. The rows show: iothubowner (registry write, service connect), device (device connect), registryRead (registry read), and registryReadWrite (registry write). To the right of the table, there's a modal dialog for the 'service' policy. It has fields for Access policy name (set to 'service'), Permissions (checkboxes for Registry read, Registry write, Service connect, Device connect, with 'Service connect' checked), and Shared access keys. The 'Primary key' field contains a long string of characters, and its copy icon is also highlighted with a red box. Below it is the 'Secondary key' field with its own copy icon. At the bottom of the dialog are Save, Discard, and More buttons.

For more information about IoT Hub shared access policies and permissions, see [Access control and permissions](#).

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:

```
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>1.17.1</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 the IoT Hub connection string you copied previously in [Get the IoT hub connection string](#):

```
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, replace the code in the `main` method with the following code:

```

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:

```
mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=simulated-device -
DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

- At your command prompt, navigate to the **simulated-device** folder.
- 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>1.17.5</version>
</dependency>
```

NOTE

You can check for the latest version of **iot-device-client** using [Maven search](#).

- Add the following dependency to the **dependencies** node. This dependency configures a NOP for the Apache [SLF4J](#) logging facade, which is used by the device client SDK to implement logging. This configuration is optional, but, if you omit it, you may see a warning in the console when you run the app. For more information about logging in the device client SDK, see [Logging](#) in the *Samples for the Azure IoT device SDK for Java* readme file.

```
<dependency>
<groupId>org.slf4j</groupId>
<artifactId>slf4j-nop</artifactId>
<version>1.7.28</version>
</dependency>
```

- 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>
```

- Save and close the **pom.xml** file.
- Using a text editor, open the **simulated-device\src\main\java\com\mycompany\app\App.java** source file.
- 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;
```

9. Add the following class-level variables to the **App** class. Replace `{yourdeviceconnectionstring}` with the device connection string you noted in the [Register a new device in the IoT hub](#) 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;
```

10. 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());
    }
}
```

11. 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());
    }
}
```

12. 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);
    }
}
```

13. 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());
        }
    }
}

```

14. 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 acknowledgment 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;
    }
}

```

15. Modify the signature of the **main** method to throw the following exceptions:

```

public static void main(String[] args) throws IOException, URISyntaxException

```

16. To instantiate a **DeviceClient**, replace the code in the **main** method with the following code:

```

System.out.println("Starting device client sample...");
client = new DeviceClient(connString, protocol);

```

17. 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...");
}

```

18. 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...");

```

19. Save and close the simulated-device\src\main\java\com\mycompany\app\App.java file.

20. 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're now ready to run the apps.

- 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"
```

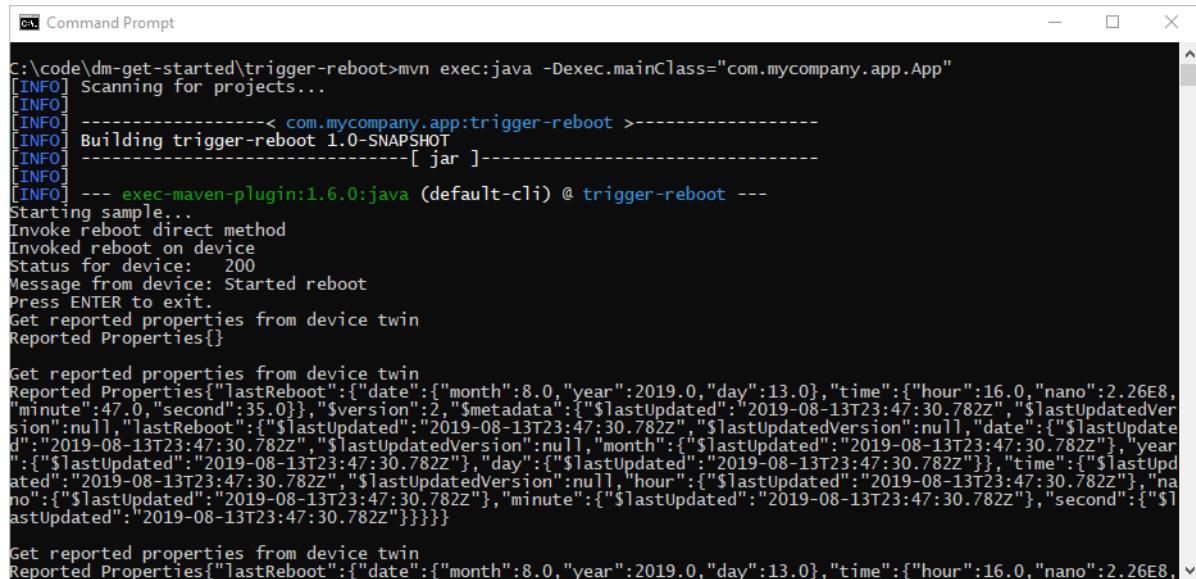
```

C:\code\dm-get-started\simulated-device>mvn exec:java -Dexec.mainClass="com.mycompany.app.App"
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.mycompany.app:simulated-device >-----
[INFO] Building simulated-device 1.0-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- exec-maven-plugin:1.6.0:java (default-cli) @ simulated-device ---
Starting device client sample...
Subscribed to direct methods and polling for reported properties. Waiting...
Press any key to exit...
IoT Hub responded to device method operation with status OK_EMPTY
IoT Hub responded to device twin operation with status OK

```

- 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:

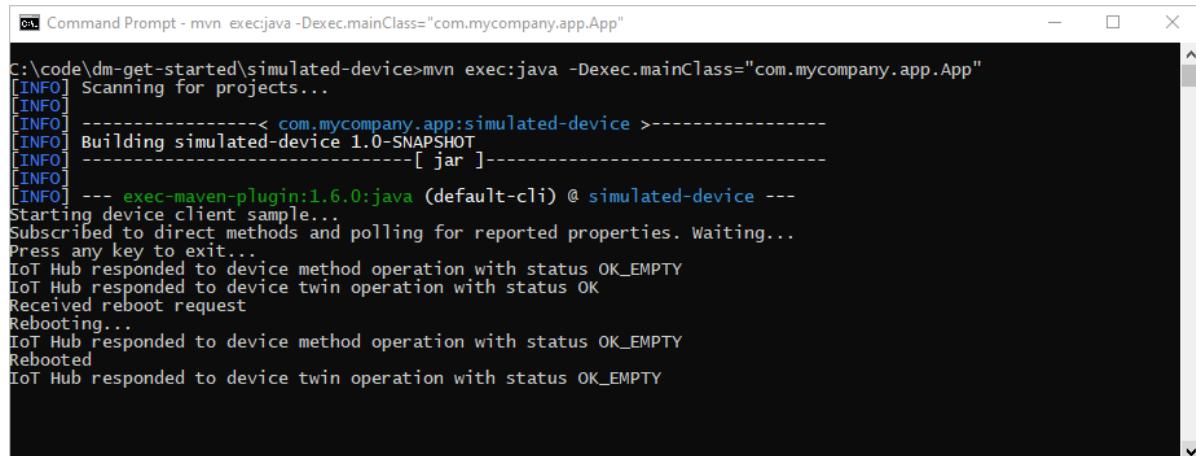
```
mvn exec:java -Dexec.mainClass="com.mycompany.app.App"
```



```
C:\code\dm-get-started\trigger-reboot>mvn exec:java -Dexec.mainClass="com.mycompany.app.App"
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.mycompany.app:trigger-reboot >-----
[INFO] Building trigger-reboot 1.0-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- exec-maven-plugin:1.6.0: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
Reported Properties{}

Get reported properties from device twin
Reported Properties{"lastReboot":{"date":{"month":8.0,"year":2019.0,"day":13.0},"time":{"hour":16.0,"nano":2.26E8,"minute":47.0,"second":35.0}},{"$metadata":{"$lastUpdated":2019-08-13T23:47:30.782Z,"$lastUpdatedVersion":null,"$lastReboot":{"$lastUpdated":2019-08-13T23:47:30.782Z,"$lastUpdatedVersion":null,"date":{"$lastUpdate":2019-08-13T23:47:30.782Z,"$lastUpdatedVersion":null,"month":{$lastUpdated":2019-08-13T23:47:30.782Z}, "year":{$lastUpdated":2019-08-13T23:47:30.782Z}, "day":{$lastUpdated":2019-08-13T23:47:30.782Z}}, "time":{"$lastUpdated":2019-08-13T23:47:30.782Z}, "hour":{$lastUpdated":2019-08-13T23:47:30.782Z}, "nano":{$lastUpdated":2019-08-13T23:47:30.782Z}}, "second":{$lastUpdated":2019-08-13T23:47:30.782Z}}}}
```

3. The simulated device responds to the reboot direct method call:



```
C:\code\dm-get-started\simulated-device>mvn exec:java -Dexec.mainClass="com.mycompany.app.App"
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.mycompany.app:simulated-device >-----
[INFO] Building simulated-device 1.0-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- exec-maven-plugin:1.6.0:java (default-cli) @ simulated-device ---
Starting device client sample...
Subscribed to direct methods and polling for reported properties. Waiting...
Press any key to exit...
IoT Hub responded to device method 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
```

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)

7/29/2020 • 11 minutes to read • [Edit Online](#)

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 available only in the standard tier of IoT Hub. For more information about the basic and standard/free IoT Hub tiers, see [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.

NOTE

IoT Hub has SDK support for many device platforms and languages (including C, Java, Javascript, and Python) through [Azure IoT device SDKs](#). For instructions on how to use Python to connect your device to this tutorial's code, and generally to Azure IoT Hub, see the [Azure IoT Python SDK](#).

Prerequisites

- An active Azure account. (If you don't have an account, you can create a [free account](#) in just a couple of minutes.)

- Python version 3.7 or later is recommended. 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. For other versions of Python supported, see [Azure IoT Device Features](#) in the SDK documentation. If you choose to use Python 2.7, you may need to [install or upgrade pip, the Python package management system](#).
- Make sure that port 8883 is open in your firewall. The device sample in this article uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

Create an IoT hub

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

1. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the + **Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.
 - **IoT Hub Name:** Enter a name for your hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

Home > New > IoT hub

IoT hub

Microsoft

X

Basics Size and scale Tags Review + create

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

Project details

Choose the subscription you'll use to manage deployments and costs. Use resource groups like folders to help you organize and manage resources.

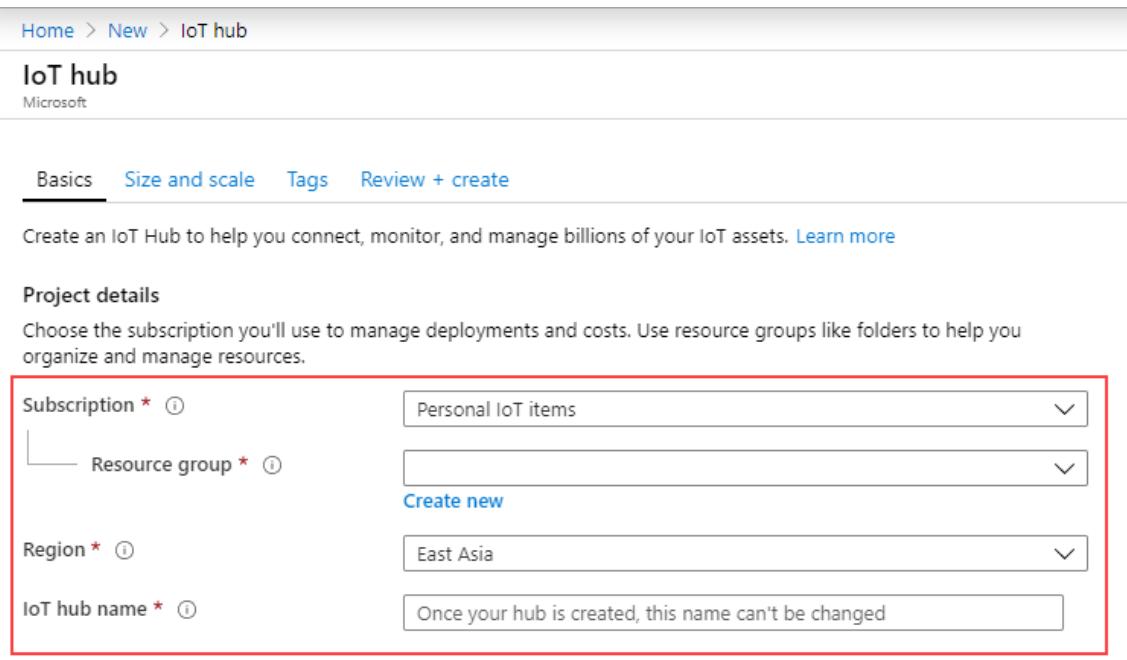
Subscription * ⓘ Personal IoT items

Resource group * ⓘ Create new

Region * ⓘ East Asia

IoT hub name * ⓘ Once your hub is created, this name can't be changed

Review + create < Previous **Next: Size and scale >** Automation options



5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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
Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center On
Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.
If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure](#)

resources.

The screenshot shows the 'Tags' step in the Azure portal for creating an IoT hub. At the top, there are tabs for 'Basics', 'Size and scale', 'Tags', and 'Review + create'. The 'Tags' tab is selected. Below it, a note says: 'Tags are name/value pairs. To categorize resources and consolidate billing, apply the same tag to multiple resources and resource groups. Your tags will update automatically if you change your resources.' A 'Learn more' link is provided. A table lists the tag: 'department' with value 'accounting'. The table has columns for 'Name', 'Value', and 'Resource'. There are two rows for IoT Hub. At the bottom, there are buttons for 'Review + create', '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' step in the Azure portal for creating an IoT hub. At the top, there are tabs for 'Basics', 'Size and scale', 'Tags', and 'Review + create'. The 'Review + create' tab is selected. Below it, there are sections for 'Basics', 'Size and scale', and 'Tags'. The 'Basics' section includes fields for Subscription (Personal testing), Resource group (iot-hubs), Region (West US 2), and IoT hub name (you-hub-name). The 'Size and scale' section includes fields for Pricing and scale tier (S1), Number of S1 IoT hub units (1), Messages per day (400,000), Cost per month (25.00 USD), and Azure Security Center (0.001 USD per device per month). The 'Tags' section includes a tag for department:accounting. At the bottom, there are buttons for 'Create' (highlighted with a red box), '< Previous: Tags', 'Next >', and 'Automation options'.

8. Select **Create** to create your new hub. Creating the hub takes a few minutes.

Register a new device in the IoT hub

In this section, you use the Azure CLI to create a device identity for this article. Device IDs are case sensitive.

1. Open [Azure Cloud Shell](#).
2. In Azure Cloud Shell, run the following command to install the Microsoft Azure IoT Extension for Azure CLI:

```
az extension add --name azure-iot
```

3. 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 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. At your command prompt, run the following command to install the `azure-iot-device` package:

```
pip install azure-iot-device
```

2. Using a text editor, create a file named `dmpatterns_getstarted_device.py` in your working directory.
3. Add the following `import` statements at the start of the `dmpatterns_getstarted_device.py` file.

```
import threading
import time
import datetime
from azure.iot.device import IoTHubDeviceClient, MethodResponse
```

4. Add the `CONNECTION_STRING` variable. Replace the `{deviceConnectionString}` placeholder value with your device connection string. You copied this connection string previously in [Register a new device in the IoT hub](#).

```
CONNECTION_STRING = "{deviceConnectionString}"
```

5. Add the following function callbacks to implement the direct method on the device.

```

def reboot_listener(client):
    while True:
        # Receive the direct method request
        method_request = client.receive_method_request("rebootDevice") # blocking call

        # Act on the method by rebooting the device...
        print( "Rebooting device" )
        time.sleep(20)
        print( "Device rebooted")

        # ...and patching the reported properties
        current_time = str(datetime.datetime.now())
        reported_props = {"rebootTime": current_time}
        client.patch_twin_reported_properties(reported_props)
        print( "Device twins updated with latest rebootTime")

        # Send a method response indicating the method request was resolved
        resp_status = 200
        resp_payload = {"Response": "This is the response from the device"}
        method_response = MethodResponse(method_request.request_id, resp_status, resp_payload)
        client.send_method_response(method_response)

```

6. Start the direct method listener and wait.

```

def iothub_client_init():
    client = IoTHubDeviceClient.create_from_connection_string(CONNECTION_STRING)
    return client

def iothub_client_sample_run():
    try:
        client = iothub_client_init()

        # Start a thread listening for "rebootDevice" direct method invocations
        reboot_listener_thread = threading.Thread(target=reboot_listener, args=(client,))
        reboot_listener_thread.daemon = True
        reboot_listener_thread.start()

        while True:
            time.sleep(1000)

    except KeyboardInterrupt:
        print ( "IoTHubDeviceClient sample stopped" )

if __name__ == '__main__':
    print ( "Starting the IoT Hub Python sample..." )
    print ( "IoTHubDeviceClient waiting for commands, press Ctrl-C to exit" )

    iothub_client_sample_run()

```

7. 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](#).

Get the IoT hub connection string

In this article, you create a backend service that invokes a direct method on a device. To invoke a direct method on a device through IoT Hub, your service needs the **service connect** permission. By default, every IoT Hub is created with a shared access policy named **service** that grants this permission.

To get the IoT Hub connection string for the **service** policy, follow these steps:

1. In the [Azure portal](#), select **Resource groups**. Select the resource group where your hub is located, and then select your hub from the list of resources.
2. On the left-side pane of your IoT hub, select **Shared access policies**.
3. From the list of policies, select the **service** policy.
4. Under **Shared access keys**, select the copy icon for the **Connection string -- primary key** and save the value.

The screenshot shows the Azure portal interface for managing IoT Hub shared access policies. The left sidebar lists various hub management options like Overview, Activity log, Access control (IAM), Tags, Events, Properties, Locks, and Export template. The 'Shared access policies' option is selected and highlighted with a red box. The main content area displays a table of existing policies. One row for 'service' is highlighted with a red box. The 'Permissions' column for the 'service' policy shows 'registry read, service connect'. The 'Shared access keys' section contains two keys: 'Primary key' and 'Connection string--primary key'. The 'Connection string--primary key' is specifically highlighted with a red box, indicating it's the one to be copied.

For more information about IoT Hub shared access policies and permissions, see [Access control and permissions](#).

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. At your command prompt, run the following command to install the `azure-iot-hub` package:

```
pip install azure-iot-hub
```

2. Using a text editor, create a file named `dmpatterns_getstarted_service.py` in your working directory.
3. Add the following `import` statements at the start of the `dmpatterns_getstarted_service.py` file.

```
import sys, time

from azure.iot.hub import IoTHubRegistryManager
from azure.iot.hub.models import CloudToDeviceMethod, CloudToDeviceMethodResult, Twin
```

4. Add the following variable declarations. Replace the `{IoTHubConnectionString}` placeholder value with the IoT hub connection string you copied previously in [Get the IoT hub connection string](#). Replace the `{deviceId}` placeholder value with the device ID you registered in [Register a new device in the IoT hub](#).

```

CONNECTION_STRING = "{IoTHubConnectionString}"
DEVICE_ID = "{deviceId}"

METHOD_NAME = "rebootDevice"
METHOD_PAYLOAD = "{\"method_number\":\"42\"}"
TIMEOUT = 60
WAIT_COUNT = 10

```

- 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:
        # Create IoTHubRegistryManager
        registry_manager = IoTHubRegistryManager(CONNECTION_STRING)

        print ( "" )
        print ( "Invoking device to reboot..." )

        # Call the direct method.
        deviceMethod = CloudToDeviceMethod(method_name=METHOD_NAME, payload=METHOD_PAYLOAD)
        response = registry_manager.invoke_device_method(DEVICE_ID, deviceMethod)

        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 = registry_manager.get_twin(DEVICE_ID)

            if twin_info.properties.reported.get("rebootTime") != None :
                print ("Last reboot time: " + twin_info.properties.reported.get("rebootTime"))
            else:
                print ("Waiting for device to report last reboot time...")

            time.sleep(5)
            status_counter += 1

    except Exception as ex:
        print ( "" )
        print ( "Unexpected error {0}".format(ex) )
        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()

```

- Save and close the `dmpatterns_getstarted_service.py` file.

Run the apps

You're now ready to run the apps.

- At the command prompt, run the following command to begin listening for the reboot direct method.

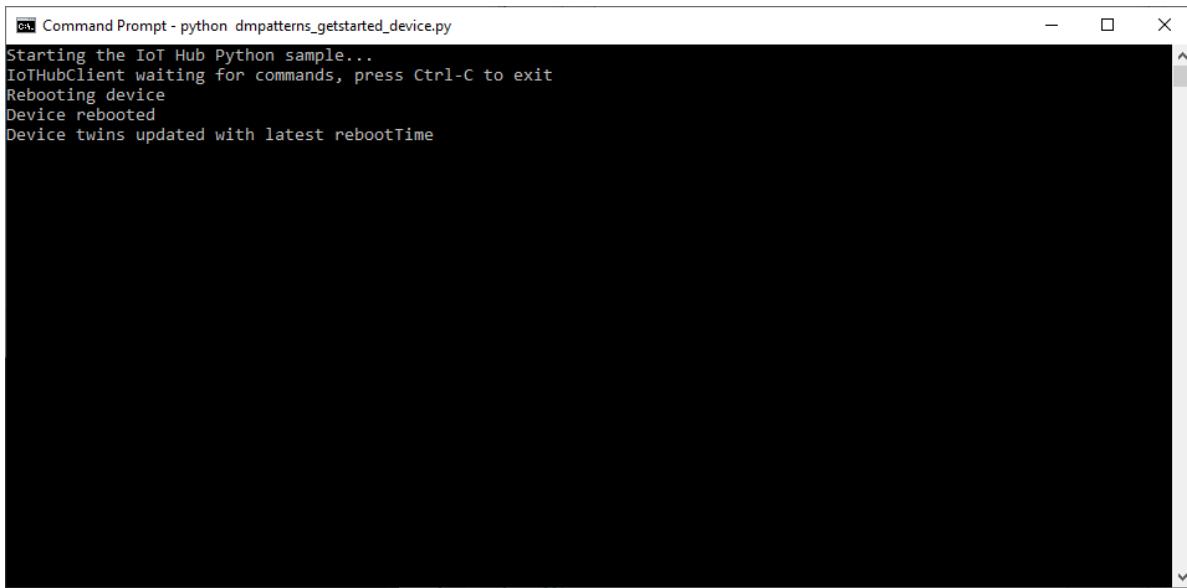
```
python dmpatterns_getstarted_device.py
```

- 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
```

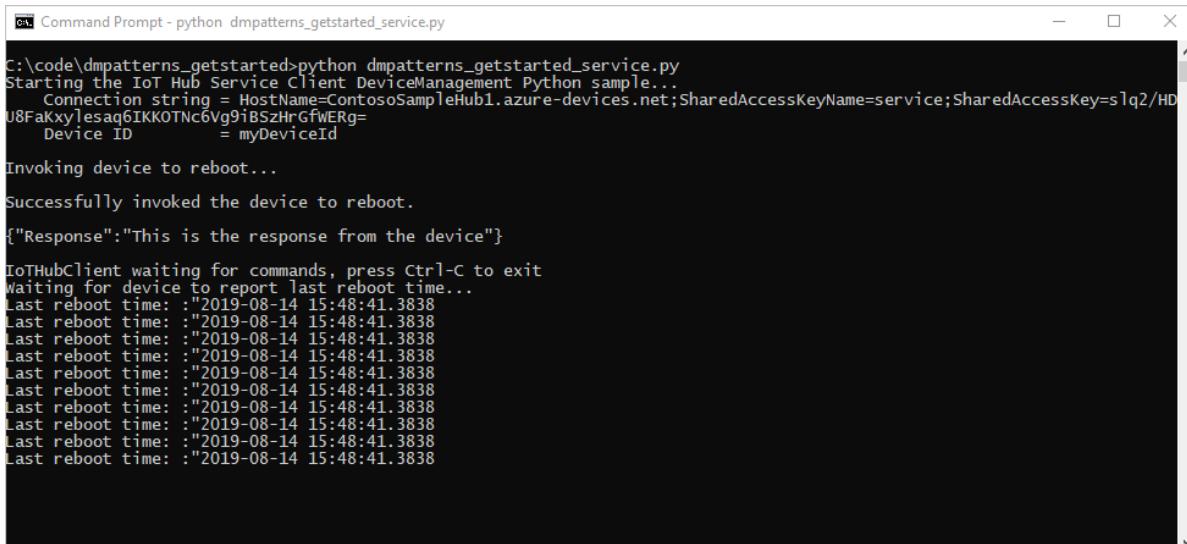
- You see the device response to the direct method in the console.

The following shows the device response to the reboot direct method:



```
Command Prompt - python dmpatterns_getstarted_device.py
Starting the IoT Hub Python sample...
IoTHubClient waiting for commands, press Ctrl-C to exit
Rebooting device
Device rebooted
Device twins updated with latest rebootTime
```

The following shows the service calling the reboot direct method and polling the device twin for status:



```
Command Prompt - python dmpatterns_getstarted_service.py
C:\code\dmpatterns_getstarted>python dmpatterns_getstarted_service.py
Starting the IoT Hub Service Client DeviceManagement Python sample...
Connection string = HostName=ContosoSampleHub1.azure-devices.net;SharedAccessKeyName=service;SharedAccessKey=s1q2/HD U8FaKxylesaq6IKKOTNc6Vg9iBSzHrGfwERg=
Device ID      = myDeviceId

Invoking device to reboot...

Successfully invoked the device to reboot.

{"Response": "This is the response from the device"}

IoTHubClient waiting for commands, press Ctrl-C to exit
Waiting for device to report last reboot time...
Last reboot time: :"2019-08-14 15:48:41.3838
```

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.js)

4/21/2020 • 10 minutes to read • [Edit Online](#)

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 available only in the standard tier of IoT Hub. For more information about the basic and standard/free IoT Hub tiers, see [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.

Prerequisites

- Node.js version 10.0.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.)
- Make sure that port 8883 is open in your firewall. The device sample in this article uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT](#)

Hub (MQTT).

Create an IoT hub

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

1. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.
 - **IoT Hub Name:** Enter a name for your hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

The screenshot shows the Azure portal's 'New' blade for creating an IoT hub. The 'Basics' tab is selected. The form includes fields for Subscription (Personal IoT items), Resource group (Create new), Region (East Asia), and IoT hub name (Once your hub is created, this name can't be changed). At the bottom, there are buttons for 'Review + create', '< Previous', 'Next: Size and scale >', and 'Automation options'.

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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
Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center On
Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.
If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select Next: Tags to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure](#)

resources.

The screenshot shows the 'Tags' tab of the IoT hub creation wizard. It displays a table of tags with one row added by the user:

| Name | Value | Resource |
|------------|------------|----------|
| department | accounting | IoT Hub |
| | | IoT Hub |

Below the table are navigation buttons: 'Review + create' (highlighted in blue), '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' page for the IoT hub. The 'Review + create' button is highlighted with a red box. The page displays the following configuration details:

| Category | Setting | Value |
|----------------|----------------------------|--------------------------------|
| Basics | Subscription | Personal testing |
| | Resource group | iot-hubs |
| | Region | West US 2 |
| | IoT hub name | you-hub-name |
| 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 |
| | Azure Security Center | 0.001 USD per device per month |
| | Tags | |
| Tags | department:accounting | |

At the bottom are buttons: 'Create' (highlighted with a red box), '< Previous: Tags', 'Next >', and 'Automation options'.

8. Select **Create** to create your new hub. Creating the hub takes a few minutes.

Register a new device in the IoT hub

In this section, you use the Azure CLI to create a device identity for this article. Device IDs are case sensitive.

1. Open [Azure Cloud Shell](#).
2. In Azure Cloud Shell, run the following command to install the Microsoft Azure IoT Extension for Azure CLI:

```
az extension add --name azure-iot
```

3. 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. Replace the `{yourDeviceConnectionString}` placeholder value with the device connection string you copied previously.

```
var connectionString = '{yourDeviceConnectionString}';
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!');
};


```

- 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);
    }
});

```

- 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](#).

Get the IoT hub connection string

In this article, you create a back-end service that schedules a job to invoke a direct method on a device, schedules a job to update the device twin, and monitors the progress of each job. To perform these operations, your service needs the **registry read** and **registry write** permissions. By default, every IoT hub is created with a shared access policy named **registryReadWrite** that grants these permissions.

To get the IoT Hub connection string for the **registryReadWrite** policy, follow these steps:

- In the [Azure portal](#), select **Resource groups**. Select the resource group where your hub is located, and then select your hub from the list of resources.
- On the left-side pane of your hub, select **Shared access policies**.
- From the list of policies, select the **registryReadWrite** policy.
- Under **Shared access keys**, select the copy icon for the **Connection string -- primary key** and save the value.

The screenshot shows the Azure IoT Hub Shared access policies page for the 'iot-hub-contoso-one' hub. On the left, a navigation menu includes 'Shared access policies' (which is selected and highlighted with a red box). The main area displays a table of policies:

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

To the right of the table, a modal window titled 'registryReadWrite' shows the policy configuration:

- Access policy name:** registryReadWrite
- Permissions:**
 - Registry read
 - Registry write **(selected)**
 - Service connect
 - Device connect
- Shared access keys:**
 - Primary key: [REDACTED]
 - Secondary key: [REDACTED]
 - Connection string—primary key: [REDACTED] **(highlighted with a red box)**
 - Connection string—secondary key: [REDACTED]

For more information about IoT Hub shared access policies and permissions, see [Access control and permissions](#).

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 **scheduleJobService.js** file:

```
'use strict';

var uuid = require('uuid');
var JobClient = require('azure-iothub').JobClient;
```

5. Add the following variable declarations. Replace the **{iothubconnectionstring}** placeholder value with the value you copied in [Get the IoT hub connection string](#). If you registered a device different than **myDeviceId**, be sure to change it in the query condition.

```
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 ===
                    'cancelled') {
                    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));
            }
        });
    }
});

```

8. 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));
      }
    });
  }
});

```

9. Save and close the `scheduleJobService.js` file.

Run the applications

You are now ready to run the applications.

- At the command prompt in the `simDevice` folder, run the following command to begin listening for the reboot direct method.

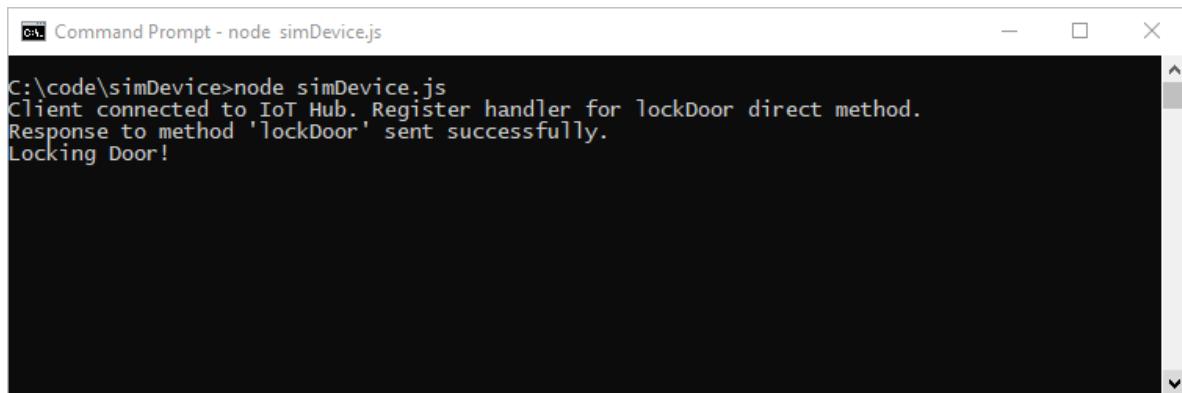
```
node simDevice.js
```

- 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
```

- You see the device response to the direct method and the job status in the console.

The following shows the device response to the direct method:



```
C:\code\simDevice>node simDevice.js
Client connected to IoT Hub. Register handler for lockDoor direct method.
Response to method 'lockDoor' sent successfully.
Locking Door!
```

The following shows the service scheduling jobs for the direct method and device twin update, and the jobs running to completion:



```
C:\code\scheduleJobService>node scheduleJobService.js
scheduling Device Method job with id: 13eab0cc-4847-4e6b-a866-007bbe36cbd6
scheduling Twin Update job with id: c1cd0b3a-ec82-4fc0-9bdf-c6066db815cc
Job: 13eab0cc-4847-4e6b-a866-007bbe36cbd6 - status: running
Job: c1cd0b3a-ec82-4fc0-9bdf-c6066db815cc - status: running
Job: 13eab0cc-4847-4e6b-a866-007bbe36cbd6 - status: completed
{
  "jobId": "13eab0cc-4847-4e6b-a866-007bbe36cbd6",
  "queryCondition": "deviceId IN ['myDeviceId']",
  "createdTime": "2019-08-02T17:01:56.3285777Z",
  "startTime": "2019-08-02T17:01:58.309Z",
  "endTime": "2019-08-02T17:02:01.7533915Z",
  "maxExecutionTimeInSeconds": 300,
  "type": "scheduleDeviceMethod",
  "cloudToDeviceMethod": {
    "methodName": "lockDoor",
    "payload": null,
    "responseTimeoutInSeconds": 15,
    "connectTimeoutInSeconds": 0
  },
  "status": "completed",
  "deviceJobStatistics": {
    "deviceCount": 1,
    "failedCount": 0,
    "succeededCount": 1,
    "runningCount": 0,
    "pendingCount": 0
  }
}
Job: c1cd0b3a-ec82-4fc0-9bdf-c6066db815cc - status: completed
{
  "jobId": "c1cd0b3a-ec82-4fc0-9bdf-c6066db815cc",
  "queryCondition": "deviceId IN ['myDeviceId']",
  "createdTime": "2019-08-02T17:01:56.393704Z",
  "startTime": "2019-08-02T17:01:58.309Z",
  "endTime": "2019-08-02T17:02:01.7046626Z",
  "maxExecutionTimeInSeconds": 300,
  "type": "scheduleUpdateTwin",
  "updateTwin": {
    "deviceId": null,
    "etag": "*",
    "properties": {
      "desired": {
        "building": "43",
        "floor": 3
      }
    }
  },
  "status": "completed",
  "deviceJobStatistics": {
    "deviceCount": 1,
    "failedCount": 0,
    "succeededCount": 1,
    "runningCount": 0,
    "pendingCount": 0
  }
}
C:\code\scheduleJobService>
```

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)

4/21/2020 • 10 minutes to read • [Edit Online](#)

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 available only in the standard tier of IoT Hub. For more information about the basic and standard/free IoT Hub tiers, see [Choose the right IoT Hub tier](#).

This tutorial shows you how to:

- Create a device app that implements a direct method called **LockDoor**, which 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**. This app connects to your IoT hub and implements the **LockDoor** direct method.
- **ScheduleJob**. This app uses jobs to call the **LockDoor** direct method and update the device twin desired properties on multiple devices.

Prerequisites

- Visual Studio.
- An active Azure account. If you don't have an account, you can create a [free account](#) in just a couple of minutes.
- Make sure that port 8883 is open in your firewall. The device sample in this article uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

Create an IoT hub

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

1. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.
 - **IoT Hub Name:** Enter a name for your hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

The screenshot shows the 'Basics' tab of the IoT hub creation wizard. A red box highlights the 'Project details' section, which includes fields for Subscription, Resource group, Region, and IoT hub name. Below this, a red box highlights the 'Next: Size and scale >' button. At the bottom, there are 'Review + create' and 'Automation options' buttons.

Home > New > IoT hub

IoT hub
Microsoft

Basics [Size and scale](#) [Tags](#) [Review + create](#)

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

Project details

Choose the subscription you'll use to manage deployments and costs. Use resource groups like folders to help you organize and manage resources.

Subscription * ⓘ

Resource group * ⓘ
[Create new](#)

Region * ⓘ

IoT hub name * ⓘ

[Review + create](#) [< Previous](#) [Next: Size and scale >](#) [Automation options](#)

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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
Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center On
Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.
If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure](#)

resources.

The screenshot shows the 'Tags' tab of the IoT hub configuration. It displays a table with one row of data:

| Name | Value | Resource |
|------------|------------|----------|
| department | accounting | IoT Hub |
| | | IoT Hub |

Below the table are navigation buttons: 'Review + create' (highlighted in blue), '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' step of the IoT hub creation process. It lists the following configuration details:

| Category | Setting | Value |
|----------------|----------------------------|--------------------------------|
| Basics | Subscription | Personal testing |
| | Resource group | iot-hubs |
| | Region | West US 2 |
| | IoT hub name | you-hub-name |
| 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 |
| | Azure Security Center | 0.001 USD per device per month |
| | Tags | |
| Tags | department:accounting | |

At the bottom are buttons: 'Create' (highlighted in red), '< Previous: Tags', 'Next >', and 'Automation options'.

8. Select **Create** to create your new hub. Creating the hub takes a few minutes.

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 a hub

unless it has an entry in the identity registry. For more information, see the [IoT Hub developer guide](#).

1. In your IoT hub navigation menu, open **IoT Devices**, then select **New** to add a device in your IoT hub.

The screenshot shows the Azure IoT Hub management interface for the 'iot-hub-contoso-one' hub. The top navigation bar includes 'Home', 'All resources', and the specific hub name. Below the navigation is a search bar and a toolbar with 'New' (highlighted), 'Refresh', and 'Delete' buttons. A message states: 'View, create, delete, and update devices in your IoT Hub.' To the right is a query editor with fields for 'Field' (set to 'select or enter a property name'), 'Operator' (set to '='), and 'Value' (set to 'specify constraint value'). Below the editor is a 'Query devices' button and a link to 'Switch to query editor'. The main table area has columns for 'DEVICE ID', 'STATUS', 'LAST ACTIVITY TIME (UTC)', 'LAST STATUS UPDATE (UTC)', 'AUTHENTICATION T...', and 'CLOUD ...'. A message 'No results' is displayed. The left sidebar contains sections for Overview, Activity log, Access control (IAM), Tags, Events, Settings (with Shared access policies, Pricing and scale, IP Filter, Certificates, Built-in endpoints, Manual failover (preview), Properties, Locks, Export template), Explorers (Query explorer, IoT devices - highlighted with a red box), and Automatic Device Management.

2. In **Create a device**, provide a name for your new device, such as **myDeviceId**, and select **Save**. This action creates a device identity for your IoT hub.

Home > All resources > iot-hub-contoso-one - IoT devices > Create a device

Create a device

Find Certified for Azure IoT devices in the Device Catalog

*** Device ID** myDeviceId

Authentication type: Symmetric key

*** Primary key**

*** Secondary key**

Auto-generate keys:

Connect this device to an IoT hub: Enable

Parent device: No 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 **Primary Connection String** to use later.

Home > iot-hub-contoso-one - IoT devices > myDeviceId

| Device ID | myDeviceId | | |
|--|---|-------------------------------------|--------------------------|
| Primary Key | H2Awv1PN3suNBkaiQU1UeEINB3j0= | | |
| Secondary Key | G7615rzcbyWFzcfIlgmad55lGVa4I= | | |
| Primary Connection String | HostName=iot-hub-contoso-one.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=QdSim6i7cptUCeMYGVSeIRKOV2ZGFSJpbmyklVYM9df= | | |
| Secondary Connection String | HostName=iot-hub-contoso-one.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=q32oiXuwifEXbbqKYkjv8sF82qZInqzGZspqkl2nqz= | | |
| Enable connection to IoT Hub | <input checked="" type="radio"/> Enable <input type="radio"/> Disable | | |
| Parent device | No parent device | | |
| Module Identities Configurations | | | |
| MODULE ID | CONNECTION STATE | CONNECTION STATE LAST UPDATED (UTC) | LAST ACTIVITY TIME (UTC) |
| There are no module identities for this device. | | | |

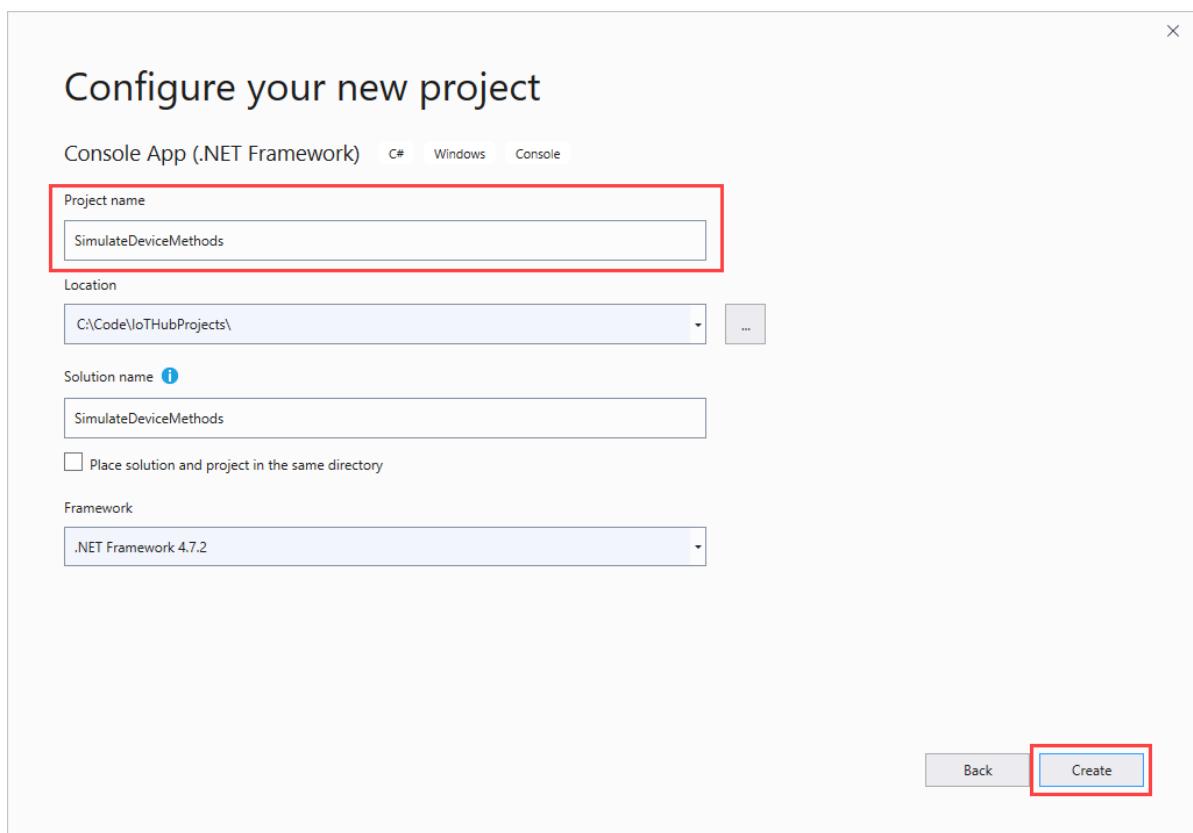
NOTE

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](#).

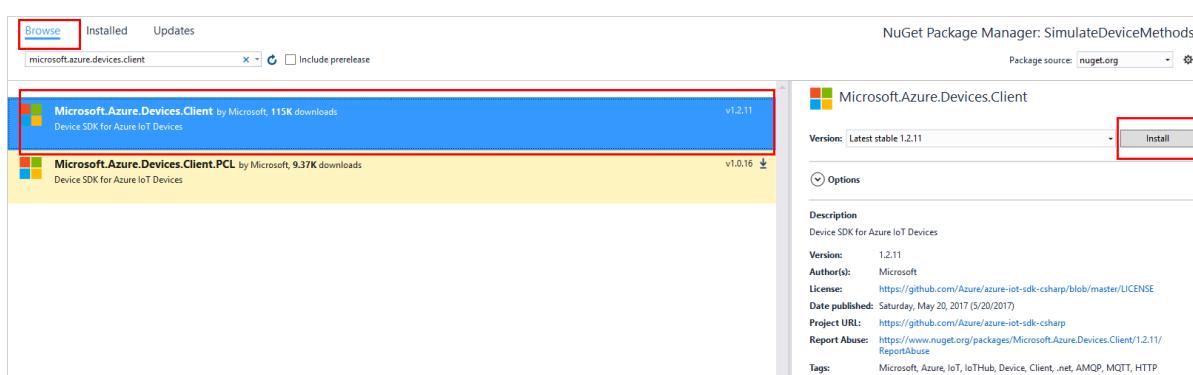
Create a simulated device app

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, select **Create a new project**, and then choose the **Console App (.NET Framework)** project template. Select **Next** to continue.
2. In **Configure your new project**, name the project *SimulateDeviceMethods*, and then select **Create**.



3. In Solution Explorer, right-click the **SimulateDeviceMethods** project, and then select **Manage NuGet Packages**.
4. In **NuGet Package Manager**, select **Browse** and search for and choose **Microsoft.Azure.Devices.Client**. Select **Install**.



This step downloads, installs, and adds a reference to the [Azure IoT device SDK](#) NuGet package and its dependencies.

5. 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;
```

6. 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;
```

7. Add the following code 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));
}
```

8. Add the following method 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));
}
```

9. 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);
}

```

10. Save your work and build your solution.

NOTE

To keep things simple, this tutorial does not implement any retry policies. In production code, you should implement retry policies (such as connection retry), as suggested in [Transient fault handling](#).

Get the IoT hub connection string

In this article, you create a back-end service that schedules a job to invoke a direct method on a device, schedules a job to update the device twin, and monitors the progress of each job. To perform these operations, your service needs the **registry read** and **registry write** permissions. By default, every IoT hub is created with a shared access policy named **registryReadWrite** that grants these permissions.

To get the IoT Hub connection string for the **registryReadWrite** policy, follow these steps:

1. In the [Azure portal](#), select **Resource groups**. Select the resource group where your hub is located, and then select your hub from the list of resources.
2. On the left-side pane of your hub, select **Shared access policies**.
3. From the list of policies, select the **registryReadWrite** policy.
4. Under **Shared access keys**, select the copy icon for the **Connection string -- primary key** and save the value.

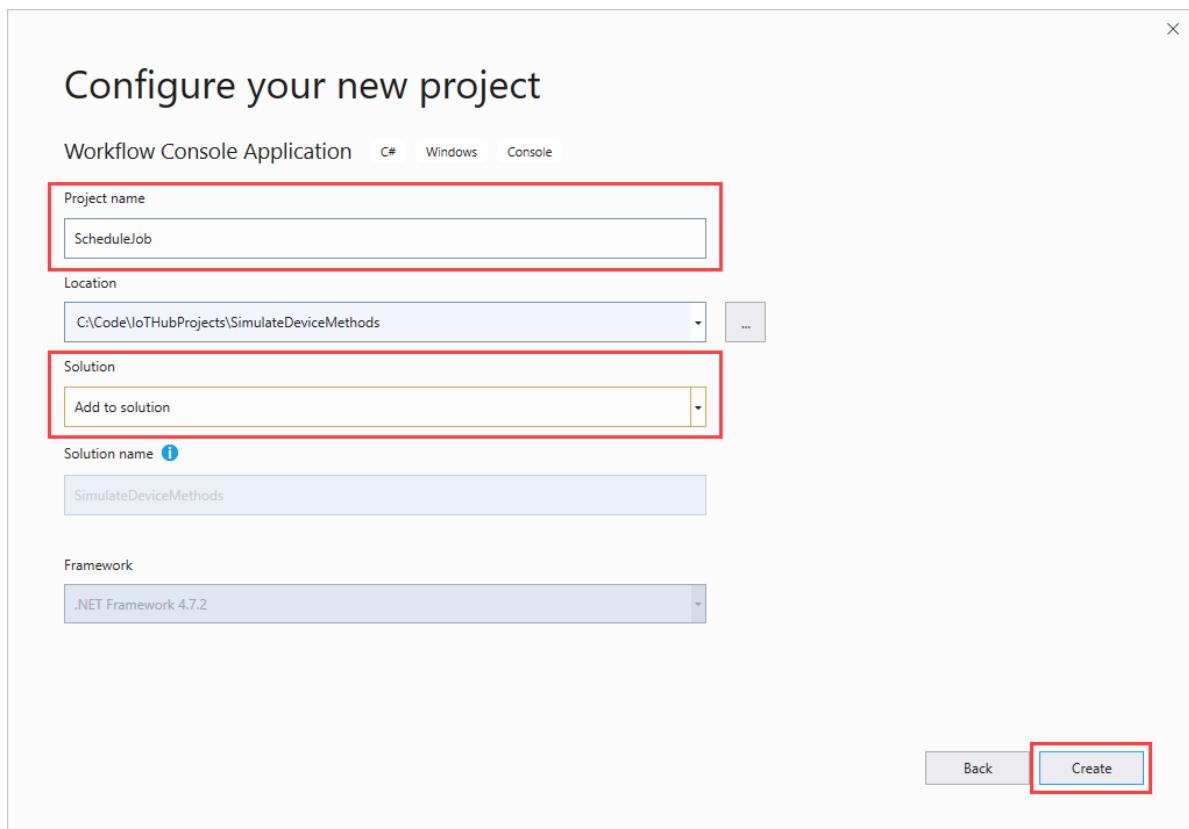
The screenshot shows the 'Shared access policies' section of the Azure IoT Hub. A new policy named 'registryReadWrite' is being created. Under the 'Permissions' section, 'Registry write' is checked. In the 'Shared access keys' section, both the 'Primary key' and 'Secondary key' fields are shown with their respective connection strings. The 'Connection string—primary key' field is highlighted with a red box.

For more information about IoT Hub shared access policies and permissions, see [Access control and permissions](#).

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, select **File > New > Project**. In **Create a new project**, choose **Console App (.NET Framework)**, and then select **Next**.
2. In **Configure your new project**, name the project *ScheduleJob*. For **Solution**, choose **Add to solution**, and then select **Create**.



3. In Solution Explorer, right-click the **ScheduleJob** project, and then select **Manage NuGet Packages**.
4. In the **NuGet Package Manager**, select **Browse**, search for and choose **Microsoft.Azure.Devices**, then select **Install**.

This step downloads, installs, and adds a reference to the [Azure IoT service SDK](#) NuGet package and its dependencies.

5. Add the following `using` statements at the top of the **Program.cs** file:

```
using Microsoft.Azure.Devices;
using Microsoft.Azure.Devices.Shared;
```

6. Add the following `using` statement if not already present in the default statements.

```
using System.Threading;
using System.Threading.Tasks;
```

7. Add the following fields to the **Program** class. Replace the placeholders with the IoT Hub connection string that you copied previously in [Get the IoT hub connection string](#) and the name of your device.

```
static JobClient jobClient;
static string connString = "<yourIoTHubConnectionString>";
static string deviceId = "<yourDeviceId>";
```

8. 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));
}
```

9. 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");
}
```

10. 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 select **Set StartUp Projects**.
- Select **Common Properties > Startup Project**, and then select **Multiple startup projects**.
- Make sure **SimulateDeviceMethods** is at the top of the list followed by **ScheduleJob**. Set both their actions to **Start** and select **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:\Code\IoTHubProjects\SimulateDeviceMethods\SimulateDeviceMethods\bin\Debug\SimulateDeviceMethods.exe', displays the following output:

```
Connecting to hub
Waiting for direct method call and device twin update
Press enter to exit.

Locking Door!

Returning response for method LockDoor
Desired property change:
{"LocationUpdate":"2019-08-08T17:41:22.0464263Z","$version":2}
```

The right window, titled 'C:\Code\IoTHubProjects\ScheduleJob\bin\Debug\ScheduleJob.exe', displays the following output:

```
Press ENTER to start running jobs.

Job Status : Unknown
Started Method Job
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 : 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)

7/29/2020 • 15 minutes to read • [Edit Online](#)

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 available only in the standard tier of IoT Hub. For more information about the basic and standard/free IoT Hub tiers, see [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

- [Java SE Development Kit 8](#). Make sure you select **Java 8** under **Long-term support** to get to downloads for JDK 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.)
- Make sure that port 8883 is open in your firewall. The device sample in this article uses MQTT protocol, which communicates over port 8883. This port may be blocked in some corporate and educational network environments. For more information and ways to work around this issue, see [Connecting to IoT Hub \(MQTT\)](#).

Create an IoT hub

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

1. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.
 - **IoT Hub Name:** Enter a name for your hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

Home > New > IoT hub

IoT hub

Microsoft

[X](#)

[Basics](#) [Size and scale](#) [Tags](#) [Review + create](#)

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

Project details

Choose the subscription you'll use to manage deployments and costs. Use resource groups like folders to help you organize and manage resources.

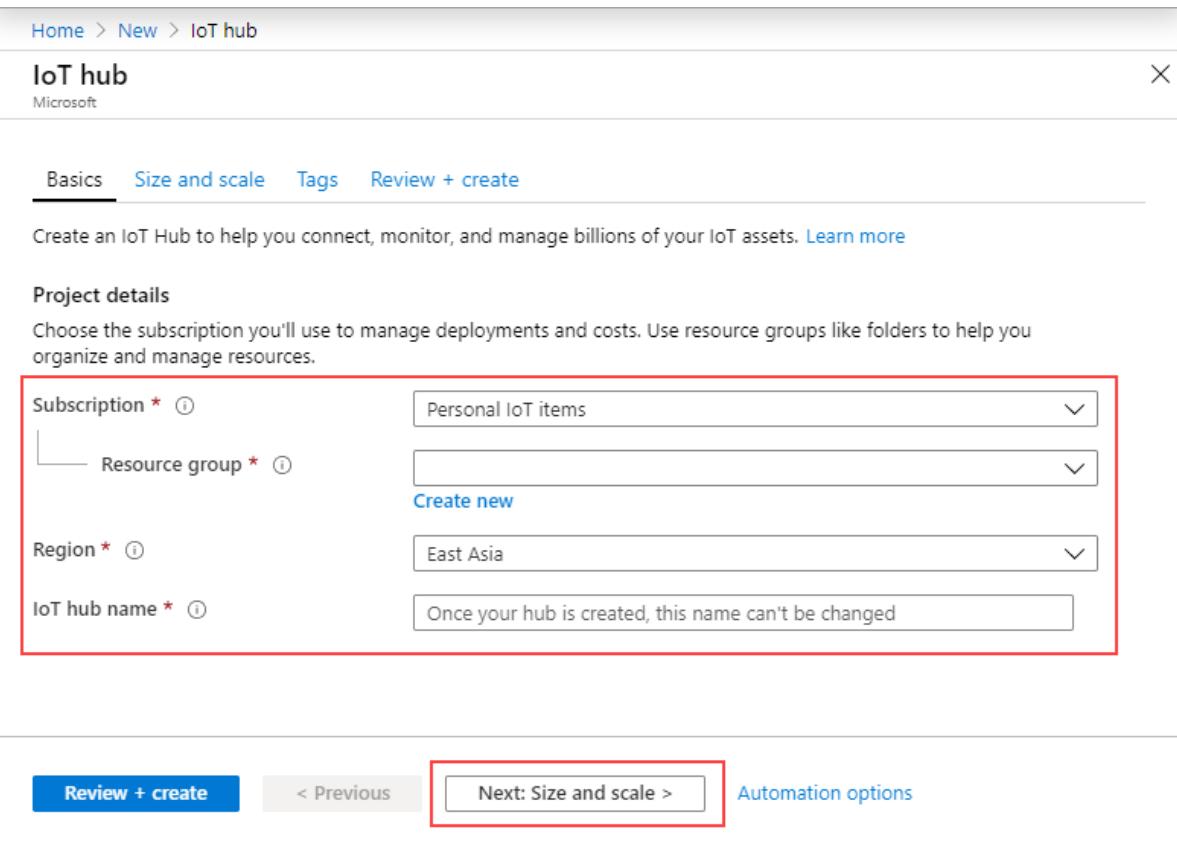
Subscription * ⓘ Personal IoT items

Resource group * ⓘ [Create new](#)

Region * ⓘ East Asia

IoT hub name * ⓘ Once your hub is created, this name can't be changed

[Review + create](#) [< Previous](#) [Next: Size and scale >](#) [Automation options](#)



5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics **Size and scale** Tags 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 [Determines how your IoT hub can scale. You can change this later if your needs increase.](#)

Azure Security Center [On](#) Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.
If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure](#)

resources.

The screenshot shows the 'Tags' tab of the IoT hub configuration page. It displays a table with one row of data:

| Name | Value | Resource |
|------------|------------|----------|
| department | accounting | IoT Hub |
| | | IoT Hub |

Below the table are navigation buttons: 'Review + create' (highlighted in blue), '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' page of the IoT hub creation wizard. It lists the following configuration details:

| Setting | Value |
|----------------------------|--------------------------------|
| Subscription | Personal testing |
| Resource group | iot-hubs |
| Region | West US 2 |
| IoT hub name | you-hub-name |
| 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 |
| Azure Security Center | 0.001 USD per device per month |
| Tags | |
| department | accounting |

At the bottom are buttons: 'Create' (highlighted in red), '< Previous: Tags', 'Next >', and 'Automation options'.

8. Select **Create** to create your new hub. Creating the hub takes a few minutes.

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 a hub

unless it has an entry in the identity registry. For more information, see the [IoT Hub developer guide](#).

1. In your IoT hub navigation menu, open **IoT Devices**, then select **New** to add a device in your IoT hub.

The screenshot shows the Azure IoT Hub management interface for the 'iot-hub-contoso-one' hub. The left sidebar contains a navigation menu with items such as Home, All resources, Overview, Activity log, Access control (IAM), Tags, Events, Settings (Shared access policies, Pricing and scale, IP Filter, Certificates, Built-in endpoints, Manual failover (preview), Properties, Locks, Export template), Explorers (Query explorer, IoT devices - highlighted with a red box), and Automatic Device Management. The main content area is titled 'iot-hub-contoso-one - IoT devices'. It features a search bar, a 'New' button (also highlighted with a red box), Refresh, and Delete buttons. A query editor allows for filtering devices based on properties, operators, and values. Below the editor is a table with columns: DEVICE ID, STATUS, LAST ACTIVITY TIME (UTC), LAST STATUS UPDATE (UTC), AUTHENTICATION T..., and CLOUD The table currently displays 'No results'.

2. In **Create a device**, provide a name for your new device, such as **myDeviceId**, and select **Save**. This action creates a device identity for your IoT hub.

Home > All resources > **iot-hub-contoso-one - IoT devices** > Create a device

Create a device

Find Certified for Azure IoT devices in the Device Catalog

*** Device ID** myDeviceId

Authentication type: Symmetric key

*** Primary key** Enter your primary key

*** Secondary key** Enter your secondary key

Auto-generate keys:

Connect this device to an IoT hub: Enable

Parent device: No 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 **Primary Connection String** to use later.

Home > **iot-hub-contoso-one - IoT devices** > myDeviceId

myDeviceId
iot-hub-contoso-one

Save Message to Device Direct Method Add Module Identity Device Twin Manage keys Refresh

| | | |
|------------------------------|---|--|
| Device ID | myDeviceId | |
| Primary Key | H2Awv1PN3suNBkaiQU1UeEINB3j0= | |
| Secondary Key | G7615rzcbqyWFzcfTlgmad55/GVa4I= | |
| Primary Connection String | HostName=iot-hub-contoso-one.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=QdSim8l7cptUCeMYGVSeiRKOV2ZGFSJpbmykIVYM9df= | |
| Secondary Connection String | HostName=iot-hub-contoso-one.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=q32joixuiffXbbqKYkjv8sF82qZnqzGZspqlk2nqz= | |
| Enable connection to IoT Hub | <input checked="" type="radio"/> Enable <input type="radio"/> Disable | |
| Parent device | No parent device | |

Module Identities Configurations

MODULE ID CONNECTION STATE CONNECTION STATE LAST UPDATED (UTC) LAST ACTIVITY TIME (UTC)

There are no module identities for this 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](#).

You can also use the [IoT extension for Azure CLI](#) tool to add a device to your IoT hub.

Get the IoT hub connection string

In this article, you create a back-end service that schedules a job to invoke a direct method on a device, schedules a job to update the device twin, and monitors the progress of each job. To perform these operations, your service needs the **registry read** and **registry write** permissions. By default, every IoT hub is created with a shared access policy named **registryReadWrite** that grants these permissions.

To get the IoT Hub connection string for the **registryReadWrite** policy, follow these steps:

1. In the [Azure portal](#), select **Resource groups**. Select the resource group where your hub is located, and then select your hub from the list of resources.
2. On the left-side pane of your hub, select **Shared access policies**.
3. From the list of policies, select the **registryReadWrite** policy.
4. Under **Shared access keys**, select the copy icon for the **Connection string -- primary key** and save the value.

The screenshot shows the Azure portal interface for managing an IoT hub. The left sidebar shows the hub's navigation menu with 'Shared access policies' highlighted. The main content area displays the 'Shared access policies' list, where the 'registryReadWrite' policy is selected. On the right, a detailed view of the 'registryReadWrite' policy shows its permissions (Registry read and Registry write checked) and its shared access keys. The 'Primary key' field is selected and has a red border, indicating it is the copied value.

For more information about IoT Hub shared access policies and permissions, see [Access control and permissions](#).

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>1.17.1</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 that you copied previously in [Get the IoT hub connection string](#):

```
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, replace the code in the **main** method with the following code:

```

// 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>1.17.5</version>
</dependency>

```

NOTE

You can check for the latest version of `iot-device-client` using [Maven search](#).

4. Add the following dependency to the **dependencies** node. This dependency configures a NOP for the Apache [SLF4J](#) logging facade, which is used by the device client SDK to implement logging. This configuration is optional, but if you omit it, you may see a warning in the console when you run the app. For more information about logging in the device client SDK, see [Logging](#) in the *Samples for the Azure IoT device SDK for Java* readme file.

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-nop</artifactId>
  <version>1.7.28</version>
</dependency>
```

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 **simulated-device\src\main\java\com\mycompany\app\App.java** file.
8. 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;
```

9. Add the following class-level variables to the **App** class. Replace `{yourdeviceconnectionstring}` with the device connection string you copied previously in the [Register a new device in the IoT hub](#) section:

```
private static String connString = "{yourdeviceconnectionstring}";
private static IoTHttpClientProtocol protocol = IoTHttpClientProtocol.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.

10. 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());
    }
}

```

11. 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());
    }
}

```

12. 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;
    }
}

```

13. Update the **main** method signature to include the following `throws` clause:

```
public static void main( String[] args ) throws IOException, URISyntaxException
```

14. Replace the code in the **main** method with the following code 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);
    }
};

```

15. 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...");
}

```

16. 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();

```

17. Save and close the **simulated-device\src\main\java\com\mycompany\app\App.java** file.

18. 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"
```

```
Command Prompt - mvn exec:java -Dexec.mainClass="com.mycompany.app.App"

:\code\iot-schedule-jobs\simulated-device>mvn exec:java -Dexec.mainClass="com.mycompany.app.App"
[INFO] Scanning for projects...
[INFO] -----< com.mycompany.app:simulated-device >-----
[INFO] Building simulated-device 1.0-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO] --- exec-maven-plugin:1.6.0:java (default-cli) @ simulated-device ---
Press any key to exit...
IoT Hub responded to device twin operation with status OK
IoT Hub responded to direct method operation with status OK_EMPTY
```

- 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"
```

```
Command Prompt
Status running for job DMCMDf50dad6d-8c02-457b-b35a-45ab617ce2dc
Status completed for job DMCMDf50dad6d-8c02-457b-b35a-45ab617ce2dc
Final status completed for job DMCMDf50dad6d-8c02-457b-b35a-45ab617ce2dc

Query device jobs since 2019-08-20T19:59:59.817Z
{
    "jobId": "DPCMD0c36532d-1b1e-45fd-b3ce-8b2d3849083a",
    "createdTime": "Aug 20, 2019 1:00:03 PM",
    "startTime": "Aug 20, 2019 1:00:00 PM",
    "lastUpdatedDateTime": "Aug 20, 2019 1:00:03 PM",
    "endTime": "Aug 20, 2019 1:00:30 PM",
    "jobType": "scheduleUpdateTwin",
    "jobStatus": "completed",
    "deviceId": "myDeviceId",
    "outcome": "[]"
}
{
    "jobId": "DMCMDf50dad6d-8c02-457b-b35a-45ab617ce2dc",
    "createdTime": "Aug 20, 2019 1:00:12 PM",
    "startTime": "Aug 20, 2019 1:00:10 PM",
    "lastUpdatedDateTime": "Aug 20, 2019 1:00:12 PM",
    "endTime": "Aug 20, 2019 1:00:40 PM",
    "jobType": "scheduleDeviceMethod",
    "jobStatus": "completed",
    "deviceId": "myDeviceId",
    "outcome": "[{"status":200,"payload":"Executed direct method lockDoor"}]",
    "outcomeResult": {
        "status": 200,
        "payload": "Executed direct method lockDoor"
    }
}
Shutting down schedule-jobs app
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 20.735 s
[INFO] Finished at: 2019-08-20T13:00:19-07:00
[INFO] -----
```

- The device app handles the desired property change and the direct method call:

```
Command Prompt - mvn exec:java -Dexec.mainClass="com.mycompany.app.App"

:\code\iot-schedule-jobs\simulated-device>mvn exec:java -Dexec.mainClass="com.mycompany.app.App"
[INFO] Scanning for projects...
[INFO] -----< com.mycompany.app:simulated-device >-----
[INFO] Building simulated-device 1.0-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO] --- exec-maven-plugin:1.6.0:java (default-cli) @ simulated-device ---
Press any key to exit...
IoT Hub responded to device twin operation with status OK
IoT Hub responded to direct method operation with status OK_EMPTY
Executing direct method: lockDoor
IoT Hub responded to direct method operation with status OK_EMPTY
```

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.

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)

7/29/2020 • 12 minutes to read • [Edit Online](#)

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 available only in the standard tier of IoT Hub. For more information about the basic and standard/free IoT Hub tiers, see [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.

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.

NOTE

IoT Hub has SDK support for many device platforms and languages (including C, Java, Javascript, and Python) through [Azure IoT device SDKs](#). For instructions on how to use Python to connect your device to this tutorial's code, and generally to Azure IoT Hub, see the [Azure IoT Python SDK](#).

Prerequisites

- An active Azure account. (If you don't have an account, you can create a [free account](#) in just a couple of minutes.)
- [Python version 3.7 or later](#) is recommended. 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. For other versions of Python supported, see [Azure IoT Device Features](#) in the SDK documentation. If you choose to use Python 2.7, you may need to [install or upgrade pip, the Python package management system](#).

Create an IoT hub

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

1. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.
 - **IoT Hub Name:** Enter a name for your hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

Home > New > IoT hub

IoT hub

Microsoft

X

Basics Size and scale Tags Review + create

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

Project details

Choose the subscription you'll use to manage deployments and costs. Use resource groups like folders to help you organize and manage resources.

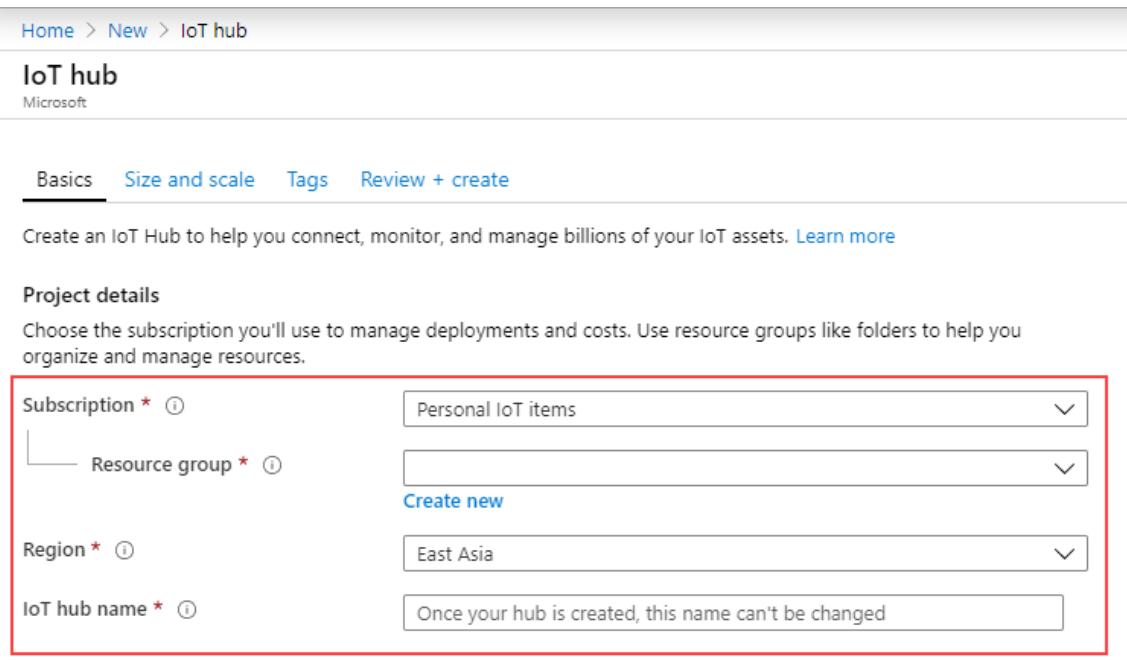
Subscription * ⓘ Personal IoT items

Resource group * ⓘ Create new

Region * ⓘ East Asia

IoT hub name * ⓘ Once your hub is created, this name can't be changed

Review + create < Previous **Next: Size and scale >** Automation options



5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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
Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center On
Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.
If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure](#)

resources.

The screenshot shows the 'Tags' section of the IoT hub creation wizard. It displays a table with one row of data:

| Name | : | Value | Resource | Actions |
|------------|---|------------|----------|-----------------------------|
| department | : | accounting | IoT Hub | ... Delete |
| | : | | IoT Hub | |

Below the table are navigation buttons: 'Review + create' (highlighted in blue), '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' page for the IoT hub. It displays the following configuration details:

| Section | Setting | Value |
|----------------------------|--------------------------------|------------------|
| Basics | Subscription | Personal testing |
| | Resource group | iot-hubs |
| | Region | West US 2 |
| | IoT hub name | you-hub-name |
| | 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 | |
| Azure Security Center | 0.001 USD per device per month | |
| Tags | | |
| department | accounting | |

At the bottom are buttons: 'Create' (highlighted in red), '< Previous: Tags', 'Next >', and 'Automation options'.

8. Select **Create** to create your new hub. Creating the hub takes a few minutes.

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 a hub

unless it has an entry in the identity registry. For more information, see the [IoT Hub developer guide](#).

1. In your IoT hub navigation menu, open **IoT Devices**, then select **New** to add a device in your IoT hub.

The screenshot shows the Azure IoT Hub management interface for the hub 'iot-hub-contoso-one'. The left sidebar contains a navigation menu with items such as Home, All resources, Overview, Activity log, Access control (IAM), Tags, Events, Settings (with Shared access policies, Pricing and scale, IP Filter, Certificates, Built-in endpoints, Manual failover (preview), Properties, Locks, and Export template), Explorers (Query explorer), and Automatic Device Management (which is also highlighted with a red box). The main content area is titled 'iot-hub-contoso-one - IoT devices'. It features a search bar at the top with a '+ New' button (highlighted with a red box) and a 'Refresh' and 'Delete' button. Below the search bar is a query editor with fields for 'Field' (select or enter a property name), 'Operator' (=), and 'Value' (specify constraint value). A link 'Switch to query editor' is also present. The main table below the query editor has columns: DEVICE ID, STATUS, LAST ACTIVITY TIME (UTC), LAST STATUS UPDATE (UTC), AUTHENTICATION T..., and CLOUD The table displays the message 'No results'.

2. In **Create a device**, provide a name for your new device, such as **myDeviceId**, and select **Save**. This action creates a device identity for your IoT hub.

Home > All resources > iot-hub-contoso-one - IoT devices > Create a device

Create a device

Find Certified for Azure IoT devices in the Device Catalog

*** Device ID** myDeviceId

Authentication type: Symmetric key

*** Primary key** Enter your primary key

*** Secondary key** Enter your secondary key

Auto-generate keys:

Connect this device to an IoT hub: Enable

Parent device: 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 **Primary Connection String** to use later.

Home > iot-hub-contoso-one - IoT devices > myDeviceId

myDeviceId
iot-hub-contoso-one

Save Message to Device Direct Method Add Module Identity Device Twin Manage keys Refresh

| | | |
|------------------------------|---|--|
| Device ID | myDeviceId | |
| Primary Key | H2Awv1PN3suIBkaiQU1UeEINB3j0= | |
| Secondary Key | G7615zcboqWFzcfTlgmad55/GVa4I= | |
| Primary Connection String | HostName=iot-hub-contoso-one.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=QdSim6i7cptUCeMYGVSeIRKOV2ZGFSJpbmykIVYM9df= | |
| Secondary Connection String | HostName=iot-hub-contoso-one.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=q32oiXuwffXbbqKYkjy8sF82qZInqzGZspqkl2nqz= | |
| Enable connection to IoT Hub | <input checked="" type="radio"/> Enable <input type="radio"/> Disable | |
| Parent device | No parent device | |

Module Identities Configurations

MODULE ID CONNECTION STATE CONNECTION STATE LAST UPDATED (UTC) LAST ACTIVITY TIME (UTC)

There are no module identities for this 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** package:

```
pip install azure-iot-device
```

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 threading
import time
from azure.iot.device import IoTHubDeviceClient, MethodResponse

CONNECTION_STRING = "{deviceConnectionString}"
```

4. Add the following function callback to handle the **lockDoor** method:

```
def lockdoor_listener(client):
    while True:
        # Receive the direct method request
        method_request = client.receive_method_request("lockDoor") # blocking call
        print( "Locking Door! " )

        resp_status = 200
        resp_payload = {"Response": "lockDoor called successfully"}
        method_response = MethodResponse(method_request.request_id, resp_status, resp_payload)
        client.send_method_response(method_response)
```

5. Add another function callback to handle device twins updates:

```
def twin_update_listener(client):
    while True:
        patch = client.receive_twin_desired_properties_patch() # blocking call
        print ("")
        print ("Twin desired properties patch received:")
        print (patch)
```

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 = IoTHubDeviceClient.create_from_connection_string(CONNECTION_STRING)

        print( "Beginning to listen for 'lockDoor' direct method invocations...")
        lockdoor_listener_thread = threading.Thread(target=lockdoor_listener, args=(client,))
        lockdoor_listener_thread.daemon = True
        lockdoor_listener_thread.start()

        # Begin listening for updates to the Twin desired properties
        print ( "Beginning to listen for updates to Twin desired properties...")
        twin_update_listener_thread = threading.Thread(target=twin_update_listener, args=(client,))
        twin_update_listener_thread.daemon = True
        twin_update_listener_thread.start()

        while True:
            time.sleep(1000)

    except KeyboardInterrupt:
        print ( "IoTHubDeviceClient sample stopped" )

    if __name__ == '__main__':
        print ( "Starting the IoT Hub Python jobs sample..." )
        print ( "IoTHubDeviceClient waiting for commands, press Ctrl-C to exit" )

    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](#).

Get the IoT hub connection string

In this article, you create a backend service that invokes a direct method on a device and updates the device twin. The service needs the **service connect** permission to call a direct method on a device. The service also needs the **registry read** and **registry write** permissions to read and write the identity registry. There is no default shared access policy that contains only these permissions, so you need to create one.

To create a shared access policy that grants **service connect**, **registry read**, and **registry write** permissions and to get a connection string for this policy, follow these steps:

1. Open your IoT hub in the [Azure portal](#). The easiest way to get to your IoT hub is to select **Resource groups**, select the resource group where your IoT hub is located, and then select your IoT hub from the list of resources.
2. On the left-side pane of your IoT hub, select **Shared access policies**.
3. From the top menu above the list of policies, select **Add**.
4. On the **Add a shared access policy** pane, enter a descriptive name for your policy; for example: `serviceAndRegistryReadWrite`. Under **Permissions**, select **Service connect** and **Registry write** (**Registry read** is automatically selected when you select **Registry write**). Then select **Create**.

The screenshot shows the 'Shared access policies' page for an IoT hub. On the left, there's a sidebar with various settings like Activity log, Access control (IAM), Tags, Events, and Shared access policies (which is selected and highlighted with a red box). In the main area, there's a table showing existing policies: iothubowner (registry write, service connect), service (service connect), device (device connect), registryRead (registry read), and registryReadWrite (registry write). A modal window titled 'Add a shared access policy' is open, showing the policy name 'serviceAndRegistryReadWrite' and a list of permissions: Registry read (checked), Registry write (checked), Service connect (checked), and Device connect (unchecked). The 'Create' button at the bottom of the modal is highlighted with a red box.

5. Back on the Shared access policies pane, select your new policy from the list of policies.
6. Under Shared access keys, select the copy icon for the Connection string -- primary key and save the value.

The screenshot shows the same 'Shared access policies' page as before, but now the 'serviceAndRegistryReadWrite' policy is selected and highlighted with a red box. In the 'Shared access keys' section, the 'Primary key' field is shown as a series of asterisks and includes a copy icon. Below it, the 'Secondary key' field is also shown with a copy icon. The 'Connection string—primary key' field is highlighted with a red box, showing its value and a copy icon. The 'Save' and 'Discard' buttons are visible at the top right of the modal.

For more information about IoT Hub shared access policies and permissions, see [Access control and permissions](#).

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 also updates the device twin's desired properties.

1. At your command prompt, run the following command to install the **azure-iot-hub** package:

```
pip install azure-iot-hub
```

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.

Replace the `{IoTHubConnectionString}` placeholder with the IoT hub connection string you copied previously in [Get the IoT hub connection string](#). Replace the `{deviceId}` placeholder with the device ID you registered in [Register a new device in the IoT hub](#):

```
import sys
import time
import threading
import uuid

from azure.iot.hub import IoTHubRegistryManager
from azure.iot.hub.models import Twin, TwinProperties, CloudToDeviceMethod, CloudToDeviceMethodResult,
QuerySpecification, QueryResult

CONNECTION_STRING = "{IoTHubConnectionString}"
DEVICE_ID = "{deviceId}"

METHOD_NAME = "lockDoor"
METHOD_PAYLOAD = "{\"lockTime\":\"10m\"}"
UPDATE_PATCH = {"building":43,"floor":3}
TIMEOUT = 60
WAIT_COUNT = 5
```

4. Add the following function that is used to query for devices:

```
def query_condition(iothub_registry_manager, device_id):

    query_spec = QuerySpecification(query="SELECT * FROM devices WHERE deviceId = "
'{}'.format(device_id))
    query_result = iothub_registry_manager.query_iot_hub(query_spec, None, 1)

    return len(query_result.items)
```

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)

    iothub_registry_manager = IoTHubRegistryManager(CONNECTION_STRING)

if query_condition(iothub_registry_manager, device_id):
    deviceMethod = CloudToDeviceMethod(method_name=METHOD_NAME, payload=METHOD_PAYLOAD)

    response = iothub_registry_manager.invoke_device_method(DEVICE_ID, deviceMethod)

    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)

    iothub_registry_manager = IoTHubRegistryManager(CONNECTION_STRING)

if query_condition(iothub_registry_manager, device_id):

    twin = iothub_registry_manager.get_twin(DEVICE_ID)
    twin_patch = Twin(properties= TwinProperties(desired=UPDATE_PATCH))
    twin = iothub_registry_manager.update_twin(DEVICE_ID, twin_patch, twin.etag)

    print ( "" )
    print ( "Device twin updated." )

```

- 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 Exception as ex:
        print ( "" )
        print ( "Unexpected error {0}" % ex )
        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
```

- 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
```

- 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...
IoTHubDeviceClient waiting for commands, press Ctrl-C to exit
Beginning to listen for 'lockDoor' direct method invocations...
Beginning to listen for updates to Twin desired properties...
Twin desired properties patch received:
{'building': 43, 'floor': 3, '$version': 5}
Locking Door!
IoTHubDeviceClient 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

7/29/2020 • 8 minutes to read • [Edit Online](#)

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. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.
 - **IoT Hub Name:** Enter a name for your hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

Home > New > IoT hub

IoT hub

Microsoft

Basics [Size and scale](#) [Tags](#) [Review + create](#)

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

Project details

Choose the subscription you'll use to manage deployments and costs. Use resource groups like folders to help you organize and manage resources.

Subscription * [Personal IoT items](#)

Resource group * [Create new](#)

Region * [East Asia](#)

IoT hub name * [Once your hub is created, this name can't be changed](#)

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

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

[Basics](#) **Size and scale** [Tags](#) [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](#)

Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center [On](#)

Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|---|--------------------------------|---|---------|
| Pricing and scale tier ① | S1 | Device-to-cloud-messages ① | Enabled |
| Messages per day ① | 400,000 | Message routing ① | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ① | Enabled |
| Azure Security Center ① | 0.001 USD per device per month | IoT Edge ① | Enabled |
| | | Device management ① | Enabled |

[Advanced settings](#)

[Review + create](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.

If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure resources](#).

| Name | Value | Resource | Actions |
|------------|------------|----------|---------|
| department | accounting | IoT Hub | ... |
| | | IoT Hub | |

Review + create < Previous: Size and scale Next: Review + create > Automation options

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags **Review + create**

Basics

| | |
|----------------|------------------|
| Subscription | Personal testing |
| Resource group | iot-hubs |
| Region | West US 2 |
| IoT hub name | you-hub-name |

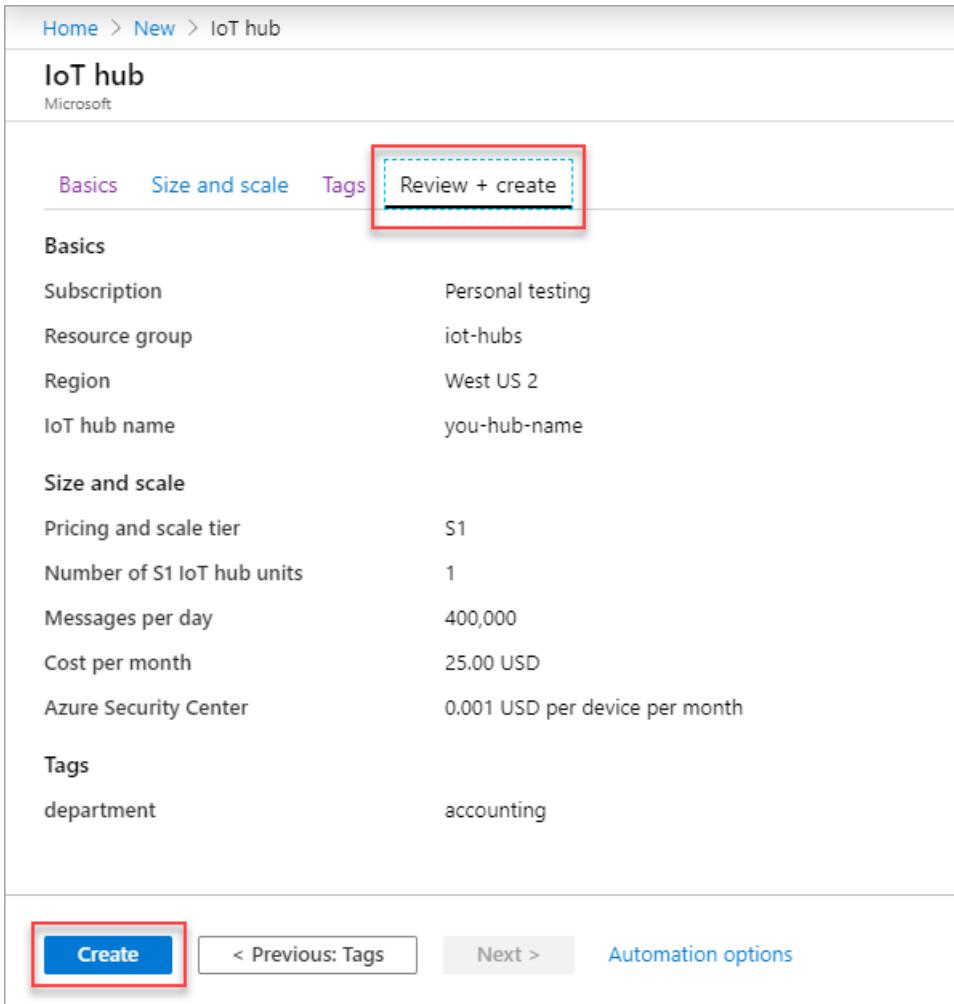
Size and scale

| | |
|----------------------------|--------------------------------|
| Pricing and scale tier | S1 |
| Number of S1 IoT hub units | 1 |
| Messages per day | 400,000 |
| Cost per month | 25.00 USD |
| Azure Security Center | 0.001 USD per device per month |

Tags

| | |
|------------|------------|
| department | accounting |
|------------|------------|

Create < Previous: Tags Next > Automation options



8. Select **Create** to create your new 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 Azure IoT Hub settings blade. On the left, there's a sidebar with sections for Settings (Shared access policies, 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), and Messaging (File upload, Message routing). The main pane displays basic properties of the IoT hub:

- Resource group**: contoso-hub-rgrp
- Status**: Active
- Location**: West US
- Hostname**: contoso-test-hub.azure-devices.net
- Pricing and scale tier**: S1 - Standard
- Number of IoT Hub units**: 1
- Subscription**: Microsoft Azure Internal Consumption
- Subscription ID**: <Your Subscription ID>

Below these details are two callout boxes:

- 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 screenshot shows the Azure portal interface for creating a shared access policy. On the left, there's a sidebar with a list of policies: iothubowner, service, device, registryRead, and registryReadWrite. The main area is titled 'Add a shared access policy' and contains fields for the access policy name ('testpolicy') and permissions. The 'Registry read' permission is selected. The 'Create' button at the bottom is highlighted with a red box.

- The **Registry read** and **Registry write** policies grant read and write access rights to the identity registry. These permissions are used by back-end cloud services to manage device identities. Choosing the write option automatically chooses the read option.
- The **Service connect** policy grants permission to access service endpoints. This permission is used by back-end cloud services to send and receive messages from devices as well as to update and read device twin and module twin data.
- The **Device connect** policy grants permissions for sending and receiving messages using the IoT Hub device-side endpoints. This permission is used by devices to send and receive messages from an IoT hub, update and read device twin and module twin data, and perform file uploads.

Click **Create** to add this newly created policy to the existing list.

For more detailed information about the access granted by specific permissions, see [IoT Hub permissions](#).

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 a hub unless it has an entry in the identity registry. For more information, see the [IoT Hub developer guide](#).

1. In your IoT hub navigation menu, open **IoT Devices**, then select **New** to add a device in your IoT hub.

The screenshot shows the 'iot-hub-contoso-one - IoT devices' blade in the Azure portal. On the left, a navigation menu includes 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Events', 'Settings' (with options like Shared access policies, Pricing and scale, IP Filter, Certificates, Built-in endpoints, Manual failover (preview), Properties, Locks, and Export template), 'Explorers' (with Query explorer selected), and 'Automatic Device Management'. The main area has a search bar, a '+ New' button (highlighted with a red box), a refresh button, and a delete button. A query editor interface is present with fields for Field, Operator, and Value, and a 'Query devices' button. A table header for DEVICE ID, STATUS, LAST ACTIVITY TIME (UTC), LAST STATUS UPDATE (UTC), AUTHENTICATION T..., and CLOUD... is shown, followed by a message 'No results'.

2. In **Create a device**, provide a name for your new device, such as **myDeviceId**, and select **Save**. This action creates a device identity for your IoT hub.

The screenshot shows the 'Create a device' blade. It includes a heading 'Create a device' with a help icon, a note about finding certified devices in the catalog, and a 'Device ID' input field containing 'myDeviceId' (highlighted with a red box). Below it are sections for 'Authentication type' (Symmetric key selected), 'Primary key' (input field placeholder 'Enter your primary key'), 'Secondary key' (input field placeholder 'Enter your secondary key'), 'Auto-generate keys' (checkbox checked), 'Connect this device to an IoT hub' (Enable button selected), and 'Parent device' (No parent device, Set a parent device link). At the bottom is a large blue 'Save' button (highlighted with a red box).

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 Primary Connection String to use later.

The screenshot shows the 'myDeviceId' device details page in the Azure IoT Hub. It displays the following information:

- Device ID:** myDeviceId
- Primary Key:** HZAw... (with a copy icon)
- Secondary Key:** G761... (with a copy icon)
- Primary Connection String:** HostName=iot-hub-contoso-one.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=QdSim... (with a copy icon)
- Secondary Connection String:** HostName=iot-hub-contoso-one.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=q32jo... (with a copy icon)
- Enable connection to IoT Hub:** Enabled (radio button selected)
- Parent device:** No 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](#).

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.

The screenshot shows the 'Routes' tab of the Message Routing pane. It includes the following sections:

- Send data from your devices to endpoints that you choose.**
- Routes** (selected) and **Custom endpoints** tabs.
- 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** button.
- Add**, **Test all routes**, and **Delete** buttons.
- Routes Table:**

| 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

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

1/12/2020 • 2 minutes to read • [Edit Online](#)

This article shows you how to use the [Azure IoT Tools for Visual Studio Code](#) to create an Azure IoT hub.

NOTE

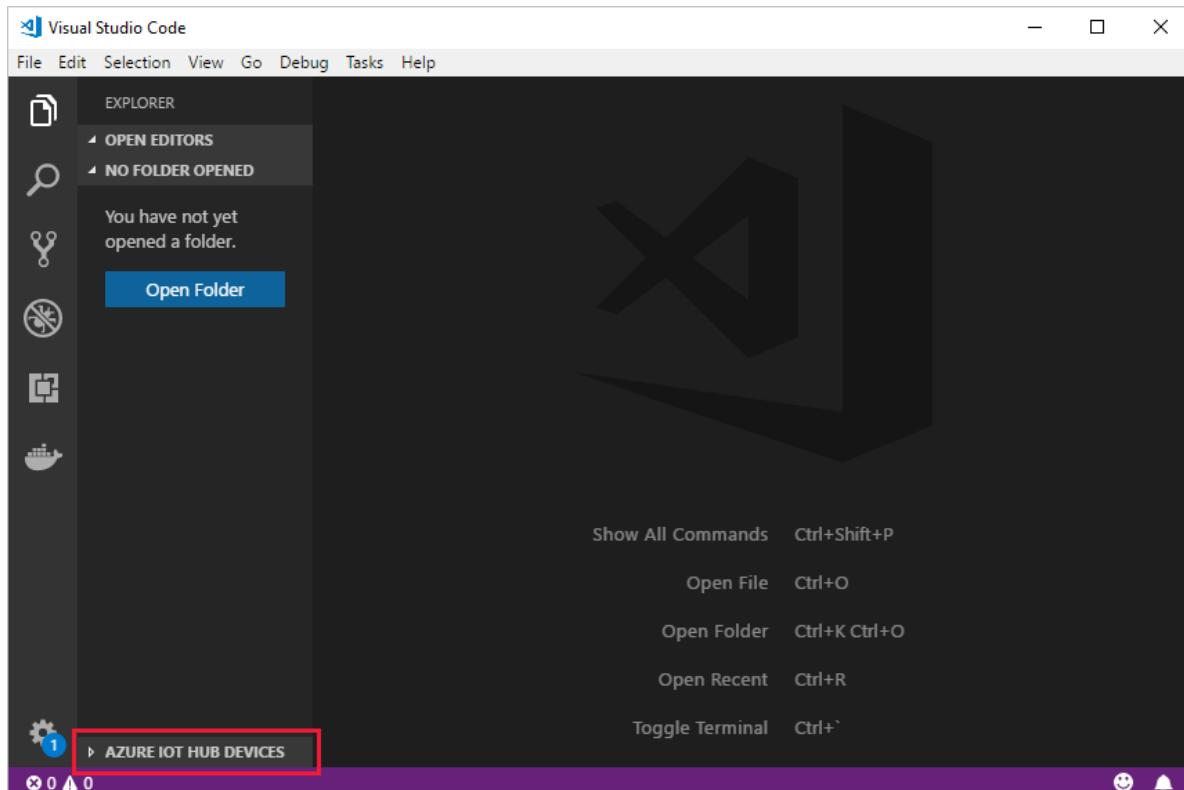
This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module 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 for VS Code wiki page.](#)

Create an IoT hub using the New-AzIoTHub cmdlet

11/8/2019 • 3 minutes to read • [Edit Online](#)

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. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module installation instructions, see [Install Azure PowerShell](#).

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|--|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal . |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

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

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name 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

11/8/2019 • 2 minutes to read • [Edit Online](#)

This article shows you how to create an IoT hub using Azure CLI.

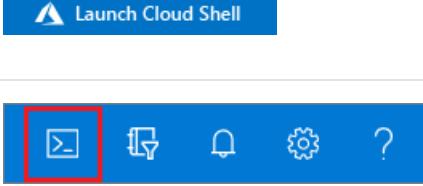
Prerequisites

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.

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|--|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal. | |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

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

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name 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)

12/23/2019 • 7 minutes to read • [Edit Online](#)

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

This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module installation instructions, see [Install Azure PowerShell](#).

To complete this tutorial, you need the following:

- Visual Studio.
- 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-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 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}"
```

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-AzADApplication -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-AzADServicePrincipal -ApplicationId {MyApplicationId}
```

7. Set up a role assignment using the following command, replacing **{MyApplicationId}** with your **ApplicationId**.

```
New-AzRoleAssignment -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

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name 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)

1/14/2020 • 2 minutes to read • [Edit Online](#)

Learn how to use an Azure Resource Manager template to create an IoT Hub and a consumer group. Resource Manager templates are JSON files that define the resources you need to deploy for your solution. For more information about developing Resource Manager templates, see [Azure Resource Manager documentation](#).

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

Create an IoT hub

The Resource Manager template used in this quickstart is from [Azure Quickstart templates](#). Here is a copy of the template:

```
{
  "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "iotHubName": {
      "type": "string",
      "minLength": 3,
      "metadata": {
        "description": "Specifies the name of the IoT Hub."
      }
    },
    "location": {
      "type": "string",
      "defaultValue": "[resourceGroup().location]",
      "metadata": {
        "description": "Location for all resources."
      }
    },
    "skuName": {
      "type": "string",
      "defaultValue": "F1",
      "metadata": {
        "description": "Specifies the IoTHub SKU."
      }
    },
    "capacityUnits": {
      "type": "int",
      "minValue": 1,
      "maxValue": 1,
      "defaultValue": 1,
      "metadata": {
        "description": "Specifies the number of provisioned IoT Hub units. Restricted to 1 unit for the F1 SKU. Can be set up to maximum number allowed for subscription."
      }
    }
  },
  "variables": {
    "consumerGroupName": "[concat(parameters('iotHubName'), '/events/cg1')]"
  },
  "resources": [
    {
      "type": "Microsoft.Devices/IotHubs",
      "apiVersion": "2018-04-01",
      "name": "[parameters('iotHubName')]"
    }
  ]
}
```

```

    "location": "[parameters('location')]",
    "properties": {
        "eventHubEndpoints": {
            "events": {
                "retentionTimeInDays": 1,
                "partitionCount": 2
            }
        },
        "cloudToDevice": {
            "defaultTtlAsIso8601": "PT1H",
            "maxDeliveryCount": 10,
            "feedback": {
                "ttlAsIso8601": "PT1H",
                "lockDurationAsIso8601": "PT60S",
                "maxDeliveryCount": 10
            }
        },
        "messagingEndpoints": {
            "fileNotifications": {
                "ttlAsIso8601": "PT1H",
                "lockDurationAsIso8601": "PT1M",
                "maxDeliveryCount": 10
            }
        }
    },
    "sku": {
        "name": "[parameters('skuName')]",
        "capacity": "[parameters('capacityUnits')]"
    }
},
{
    "type": "Microsoft.Devices/iotHubs/eventhubEndpoints/ConsumerGroups",
    "apiVersion": "2018-04-01",
    "name": "[variables('consumerGroupName')]",
    "dependsOn": [
        "[resourceId('Microsoft.Devices/IotHubs', parameters('iotHubName'))]"
    ]
}
]
}

```

The template creates an Azure IoT hub with three endpoints (eventhub, cloud-to-device, and messaging), and a consumer group. For more template samples, see [Azure Quickstart templates](#). The IoT Hub template schema can be found [here](#).

There are several methods for deploying a template. You use Azure PowerShell in this tutorial.

To run the PowerShell script, Select Try it to open the Azure Cloud shell. To paste the script, right-click the shell, and then select Paste:

```

$resourceGroupName = Read-Host -Prompt "Enter the Resource Group name"
.setLocation = Read-Host -Prompt "Enter the location (i.e. centralus)"
$iotHubName = Read-Host -Prompt "Enter the IoT Hub name"

New-AzResourceGroup -Name $resourceGroupName -Location "$location"
New-AzResourceGroupDeployment ` 
    -ResourceGroupName $resourceGroupName ` 
    -TemplateUri "https://raw.githubusercontent.com/Azure/azure-quickstart-templates/master/101-iothub-with-` 
    consumergroup-create/azuredeploy.json" ` 
    -iotHubName $iotHubName

```

As you can see from the PowerShell script, the template used is from Azure Quickstart templates. To use your own, you need to first upload the template file to the Cloud shell, and then use the `-TemplateFile` switch to specify the file name. For an example, see [Deploy the template](#).

Next steps

Now you have deployed an IoT hub by using an Azure Resource Manager template, 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)

1/14/2020 • 8 minutes to read • [Edit Online](#)

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. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module installation instructions, see [Install Azure PowerShell](#).

To complete this tutorial, you need the following:

- Visual Studio.
- 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-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 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}"
```

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-AzADApplication -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-AzADServicePrincipal -ApplicationId {MyApplicationId}
```

7. Set up a role assignment using the following command, replacing **{MyApplicationId}** with your **ApplicationId**.

```
New-AzRoleAssignment -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

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name 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 **template.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

4/4/2019 • 2 minutes to read • [Edit Online](#)

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 page for an IoT hub named 'getStartedWithAnIoTHub'. The left sidebar lists various configuration options under 'MESSAGING' (highlighted with a red box), including 'File upload' (which is selected and highlighted in blue), 'Endpoints', and 'Routes'. Other sections like 'EXPLORERS' (Device Explorer, Query Explorer) and 'MONITORING' (Metrics) are also visible. The main content area contains a summary message: 'Here you specify the storage container, file expiration, and retries for notifications when a file is uploaded.' Below this are settings for 'Storage container' (set to 'uploadcontainer'), 'Receive notifications for uploaded files' (set to 'On'), 'SAS TTL' (set to 1 hour), 'File notification settings' (Default TTL set to 1 hour), and 'Maximum delivery count' (set to 10).

- **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.

The screenshot shows two adjacent Azure blade windows. On the left is the 'Storage accounts' blade, which lists a single storage account named 'myiothubfileupload' of type 'Standard-LRS' under the resource group 'getstartedwithiothub_rg'. On the right is the 'Containers' blade, which lists several containers: 'another', 'fileupload', 'iothubimportexport', 'qtcontainer', and 'uploadcontainer'. Each container entry includes a 'LAST MODIFIED' timestamp.

- **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.

This screenshot shows the 'File upload' configuration page within the Azure IoT Hub blade. The left sidebar lists various configuration sections like Properties, Locks, Automation script, and others. The main area shows 'Storage container settings' with an 'Azure storage account' dropdown set to 'Uploadcontainer'. Below it, 'Azure storage container' is also set to 'Uploadcontainer'. A tooltip explains that enabling logging will log information about uploaded files to the IoT Hub endpoint. In the 'File notification settings' section, the 'On' toggle switch is highlighted with a red box. Below it, three sliders are shown: 'SAS TTL' (set to 1 hr), 'Default TTL' (set to 1 hr), and 'Maximum delivery count' (set to 10). The entire 'File notification settings' section is also highlighted with a large red box.

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

4/4/2019 • 3 minutes to read • [Edit Online](#)

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. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module 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 URLs 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 URLs 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

3/4/2020 • 3 minutes to read • [Edit Online](#)

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 to create one. For more information, see [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 URLs 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 URLs 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

7/29/2020 • 10 minutes to read • [Edit Online](#)

IoT Hub metrics give you information about the state of the Azure IoT resources in your Azure subscription. IoT Hub metrics help you assess the overall health of the IoT Hub service and the devices connected to it. These user-facing statistics help you see what is going on with your IoT hub and help perform root-cause analysis on issues without needing to contact Azure support.

Metrics are enabled by default. You can view IoT Hub metrics from the Azure portal.

NOTE

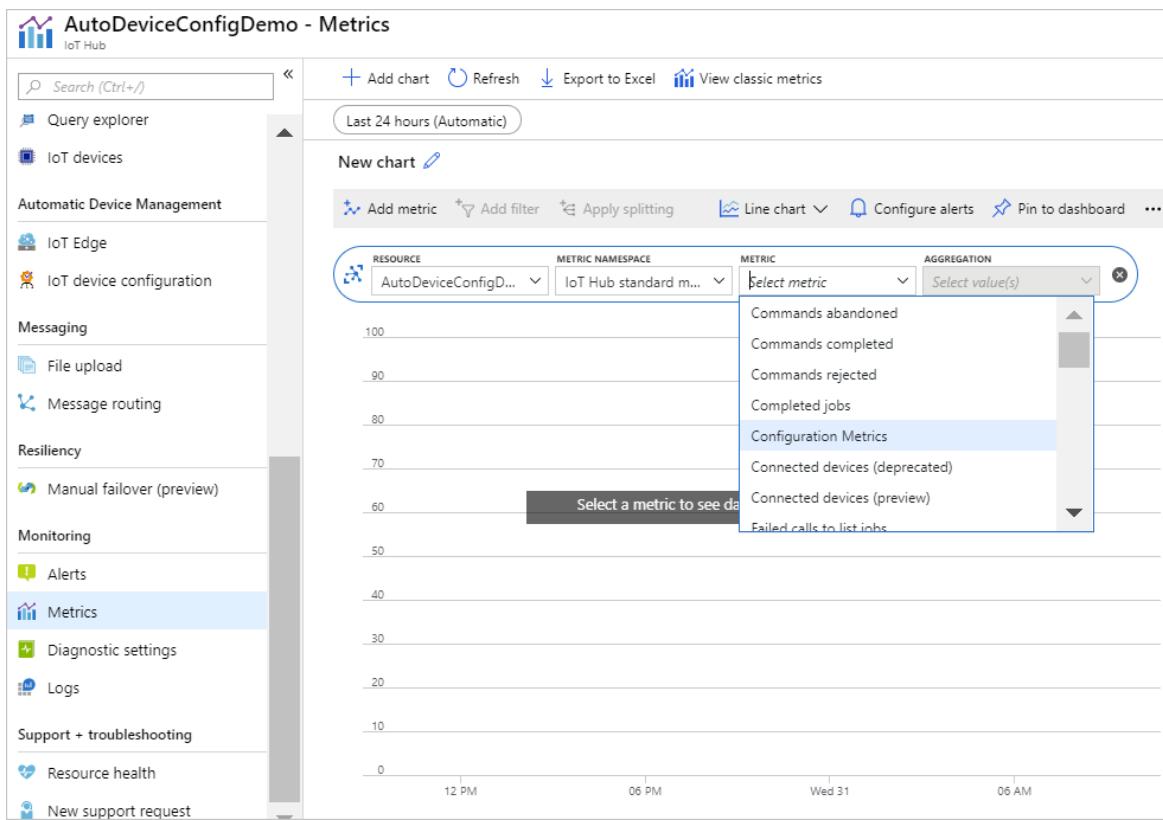
You can use IoT Hub metrics to view information about IoT Plug and Play devices connected to your IoT Hub. IoT Plug and Play devices are part of the [IoT Plug and Play public preview](#).

How to view IoT Hub metrics

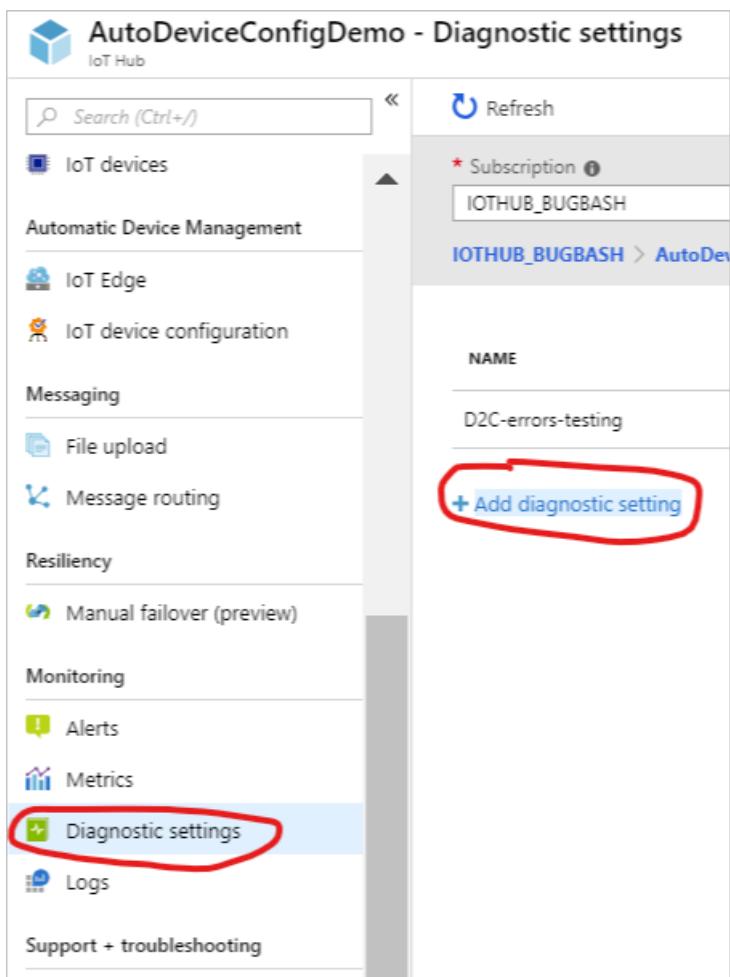
1. Create an IoT hub. You can find instructions on how to create an IoT hub in the [Send telemetry from a device to IoT Hub](#) guide.
2. Open the blade of your IoT hub. From there, click **Metrics**.

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), 'Support + troubleshooting' (Resource health, New support request), and 'Metrics' (highlighted with a red oval). The main pane displays the IoT hub's properties: Resource group (change) to '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) to '<subscription name>', Subscription ID '<subscription ID>', and Tags (change) with a link to 'Click here to add tags'. Below the properties, there are two promotional callouts: one for 'Need a way to provision millions of devices?' (IoT Hub Device Provisioning Service) and another for 'Want to learn 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 |
|-----------------------------------|---|--------------|------------------|--|---|
| RoutingDeliveries | Routing Delivery Attempts (preview) | Count | Total | This is the routing delivery metric. Use the dimensions to identify the delivery status for a specific endpoint or for a specific routing source. | ResourceID, Result, RoutingSource, EndpointType, FailureReasonCategory, EndpointName <i>More details on dimensions here.</i> |
| RoutingDeliveryLatency | Routing Latency (preview) | Milliseconds | Average | This is the routing delivery latency metric. Use the dimensions to identify the latency for a specific endpoint or for a specific routing source. | ResourceID, RoutingSource, EndpointType, EndpointName <i>More details on dimensions here.</i> |
| RoutingDataSizeInBytesDelivered | Routing Delivery Data Size In Bytes (preview) | Bytes | Total | The total number of bytes routed by IoT Hub to custom endpoint and built-in endpoint. Use the dimensions to identify data size routed to a specific endpoint or for a specific routing source. | ResourceID, RoutingSource, EndpointType, EndpointName <i>More details on dimensions here.</i> |
| 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 | None |
| d2c.telemetry.ingress.success | Telemetry messages sent | Count | Total | Number of device-to-cloud telemetry messages sent successfully to your IoT hub | None |

| METRIC | METRIC DISPLAY NAME | UNIT | AGGREGATION TYPE | DESCRIPTION | DIMENSIONS |
|--------------------------------------|---------------------------------------|-------|------------------|---|------------|
| c2d.commands.egress.complete.success | C2D message deliveries completed | Count | Total | Number of cloud-to-device message deliveries completed successfully by the device | None |
| c2d.commands.egress.abandon.success | C2D messages abandoned | Count | Total | Number of cloud-to-device messages abandoned by the device | None |
| c2d.commands.egress.reject.success | C2D messages rejected | Count | Total | Number of cloud-to-device messages rejected by the device | None |
| C2DMessagesExpired | C2D Messages Expired (preview) | Count | Total | Number of expired cloud-to-device messages | None |
| devices.totalDevices | Total devices (deprecated) | Count | Total | Number of devices registered to your IoT hub | None |
| devices.connectedDevices.allProtocol | Connected devices (deprecated) | Count | Total | Number of devices connected to your IoT hub | None |
| 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. | None |

| METRIC | METRIC DISPLAY NAME | UNIT | AGGREGATION TYPE | DESCRIPTION | 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. | None |
| 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 query, when fallback route is disabled. | None |
| 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. A message is incompatible with an endpoint when IoT Hub attempts to deliver the message to an endpoint and it fails with a non transient error. Invalid messages are not retried. This value does not include retries. | None |
| 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. | None |

| METRIC | METRIC DISPLAY NAME | UNIT | AGGREGATION TYPE | DESCRIPTION | DIMENSIONS |
|--|--|--------------|------------------|---|------------|
| d2c.endpoints.egress.eventHubs | Routing: messages delivered to Event Hub | Count | Total | The number of times IoT Hub routing successfully delivered messages to custom endpoints of type Event Hub. This does not include messages routes to built-in endpoint (events). | None |
| 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 custom endpoints of type Event Hub. This does not include messages routes to built-in endpoint (events). | None |
| 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. | None |
| d2c.endpoints.latency.serviceBusQueues | Routing: message latency for Service Bus Queue | Milliseconds | Average | The average latency (milliseconds) between message ingress to IoT Hub and message ingress into a Service Bus queue endpoint. | None |

| METRIC | METRIC DISPLAY NAME | UNIT | AGGREGATION TYPE | DESCRIPTION | 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. | None |
| d2c.endpoints.latency. serviceBusTopics | Routing: message latency for Service Bus Topic | Milliseconds | Average | The average latency (milliseconds) between message ingress to IoT Hub and message ingress into a Service Bus topic endpoint. | None |
| 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) and fallback route. | None |
| d2c.endpoints.latency. builtIn.events | Routing: message latency for messages/events | Milliseconds | Average | The average latency (milliseconds) between message ingress to IoT Hub and message ingress into the built-in endpoint (messages/events) and fallback route. | None |
| d2c.endpoints.egress. storage | Routing: messages delivered to storage | Count | Total | The number of times IoT Hub routing successfully delivered messages to storage endpoints. | None |

| METRIC | METRIC DISPLAY NAME | UNIT | AGGREGATION TYPE | DESCRIPTION | DIMENSIONS |
|------------------------------------|--------------------------------------|--------------|------------------|---|-------------------------------|
| d2c.endpoints.latency.storage | Routing: message latency for storage | Milliseconds | Average | The average latency (milliseconds) between message ingress to IoT Hub and message ingress into a storage endpoint. | None |
| d2c.endpoints.egress.storage.bytes | Routing: data delivered to storage | Bytes | Total | The amount of data (bytes) IoT Hub routing delivered to storage endpoints. | None |
| d2c.endpoints.egress.storage.blobs | Routing: blobs delivered to storage | Count | Total | The number of times IoT Hub routing delivered blobs to storage endpoints. | None |
| EventGridDeliveries | Event Grid deliveries(preview) | Count | Total | The number of IoT Hub events published to Event Grid. Use the Result dimension for the number of successful and failed requests. EventType dimension shows the type of event (https://aka.ms/ioteventgrid). | ResourceId, Result, EventType |
| EventGridLatency | Event Grid latency (preview) | Milliseconds | Average | The average latency (milliseconds) from when the IoT Hub event was generated to when the event was published to Event Grid. This number is an average between all event types. Use the EventType dimension to see latency of a specific type of event. | ResourceId, EventType |

| METRIC | METRIC DISPLAY NAME | UNIT | AGGREGATION TYPE | DESCRIPTION | DIMENSIONS |
|--------------------------|--|-------|------------------|---|------------|
| d2c.twin.read.success | Successful twin reads from devices | Count | Total | The count of all successful device-initiated twin reads. | None |
| d2c.twin.read.failure | Failed twin reads from devices | Count | Total | The count of all failed device-initiated twin reads. | None |
| d2c.twin.read.size | Response size of twin reads from devices | Bytes | Average | The number of all successful device-initiated twin reads. | None |
| d2c.twin.update.success | Successful twin updates from devices | Count | Total | The count of all successful device-initiated twin updates. | None |
| d2c.twin.update.failure | Failed twin updates from devices | Count | Total | The count of all failed device-initiated twin updates. | None |
| d2c.twin.update.size | Size of twin updates from devices | Bytes | Average | The total size of all successful device-initiated twin updates. | None |
| c2d.methods.success | Successful direct method invocations | Count | Total | The count of all successful direct method calls. | None |
| c2d.methods.failure | Failed direct method invocations | Count | Total | The count of all failed direct method calls. | None |
| c2d.methods.requestSize | Request size of direct method invocations | Bytes | Average | The count of all successful direct method requests. | None |
| c2d.methods.responseSize | Response size of direct method invocations | Bytes | Average | The count of all successful direct method responses. | None |
| c2d.twin.read.success | Successful twin reads from back end | Count | Total | The count of all successful back-end-initiated twin reads. | None |

| METRIC | METRIC DISPLAY NAME | UNIT | AGGREGATION TYPE | DESCRIPTION | DIMENSIONS |
|------------------------------------|--|-------|------------------|--|------------|
| c2d.twin.read.failure | Failed twin reads from back end | Count | Total | The count of all failed back-end-initiated twin reads. | None |
| c2d.twin.read.size | Response size of twin reads from back end | Bytes | Average | The count of all successful back-end-initiated twin reads. | None |
| c2d.twin.update.success | Successful twin updates from back end | Count | Total | The count of all successful back-end-initiated twin updates. | None |
| c2d.twin.update.failure | Failed twin updates from back end | Count | Total | The count of all failed back-end-initiated twin updates. | None |
| c2d.twin.update.size | Size of twin updates from back end | Bytes | Average | The total size of all successful back-end-initiated twin updates. | None |
| twinQueries.success | Successful twin queries | Count | Total | The count of all successful twin queries. | None |
| twinQueries.failure | Failed twin queries | Count | Total | The count of all failed twin queries. | None |
| twinQueries.resultSize | Twin queries result size | Bytes | Average | The total of the result size of all successful twin queries. | None |
| jobs.createTwinUpdateJob.success | Successful creations of twin update jobs | Count | Total | The count of all successful creation of twin update jobs. | None |
| jobs.createTwinUpdateJob.failure | Failed creations of twin update jobs | Count | Total | The count of all failed creation of twin update jobs. | None |
| jobs.createDirectMethodJob.success | Successful creations of method invocation jobs | Count | Total | The count of all successful creation of direct method invocation jobs. | None |

| METRIC | METRIC DISPLAY NAME | UNIT | AGGREGATION TYPE | DESCRIPTION | DIMENSIONS |
|------------------------------------|--|-------|------------------|---|------------|
| jobs.createDirectMethodJob.failure | Failed creations of method invocation jobs | Count | Total | The count of all failed creation of direct method invocation jobs. | None |
| jobs.listJobs.success | Successful calls to list jobs | Count | Total | The count of all successful calls to list jobs. | None |
| jobs.listJobs.failure | Failed calls to list jobs | Count | Total | The count of all failed calls to list jobs. | None |
| jobs.cancelJob.success | Successful job cancellations | Count | Total | The count of all successful calls to cancel a job. | None |
| jobs.cancelJob.failure | Failed job cancellations | Count | Total | The count of all failed calls to cancel a job. | None |
| jobs.queryJobs.success | Successful job queries | Count | Total | The count of all successful calls to query jobs. | None |
| jobs.queryJobs.failure | Failed job queries | Count | Total | The count of all failed calls to query jobs. | None |
| jobs.completed | Completed jobs | Count | Total | The count of all completed jobs. | None |
| jobs.failed | Failed jobs | Count | Total | The count of all failed jobs. | None |
| d2c.telemetry.ingress.sendThrottle | Number of throttling errors | Count | Total | Number of throttling errors due to device throughput throttles | None |
| dailyMessageQuotaUsed | 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. | None |
| deviceDataUsage | Total device data usage | Bytes | Total | Bytes transferred to and from any devices connected to IoT Hub | None |

| METRIC | METRIC DISPLAY NAME | UNIT | AGGREGATION TYPE | DESCRIPTION | DIMENSIONS |
|----------------------|-----------------------------------|-------|------------------|---|------------|
| deviceDataUsage V2 | Total device data usage (preview) | Bytes | Total | Bytes transferred to and from any devices connected to IoT Hub | None |
| totalDeviceCount | Total devices (preview) | Count | Average | Number of devices registered to your IoT hub | None |
| connectedDeviceCount | Connected devices (preview) | Count | Average | Number of devices connected to your IoT hub | None |
| configurations | Configuration Metrics | Count | Total | Number of total CRUD operations performed for device configuration and IoT Edge deployment, on a set of target devices. This also includes the number of operations that modify the device twin or module twin because of these configurations. | None |

Dimensions

Dimensions help identify more details about the metrics. Some of the routing metrics provide information per endpoint. Table below lists possible values for these dimensions.

| DIMENSION | VALUES |
|---------------|---|
| ResourceID | your IoT Hub resource |
| Result | success failure |
| RoutingSource | Device Messages Twin Change Events Device Lifecycle Events |
| EndpointType | eventHubs serviceBusQueues cosmosDB serviceBusTopics builtin blobStorage |

| DIMENSION | VALUES |
|-----------------------|--|
| FailureReasonCategory | invalid dropped orphaned null |
| EndpointName | endpoint name |

Next steps

Now that you've seen an overview of IoT Hub metrics, follow this link to learn more about managing Azure IoT Hub:

- [Set up diagnostic logs](#)
- [Learn how to troubleshoot message routing](#)

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

7/29/2020 • 12 minutes to read • [Edit Online](#)

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.

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

NOTE

This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module installation instructions, see [Install Azure PowerShell](#).

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 Azure portal interface for configuring IoT Hub diagnostic settings. At the top, there are filters for Subscription (Visual Studio Enterprise with MSDN), Resource group (IoT), and Resource type (IoT Hub). Below the filters, the path Visual Studio Enterprise with MSDN > IoT > DiagnosticsHub is displayed. A callout box highlights the 'Turn on diagnostics' section, which contains a list of monitoring operations: 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-AzAccount  
Select-AzSubscription -SubscriptionName <subscription that includes your IoT Hub>  
Set-AzDiagnosticSetting -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\\\",\\\"acceptingIpFilterRu  
le\\\"\\\":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\":<messageSize>,\"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": "2019-12-12T03:25:14Z",
      "resourceId": "/SUBSCRIPTIONS/91R34780-3DEC-123A-BE2A-213B5500DFF0/RESOURCEGROUPS/ANON-TEST/PROVIDERS/MICROSOFT.DEVICES/IOTHUBS/ANONHUB1",
      "operationName": "endpointUnhealthy",
      "category": "Routes",
      "level": "Error",
      "resultType": "403004",
      "resultDescription": "DeviceMaximumQueueDepthExceeded",
      "properties": "{\"deviceId\":null,\"endpointName\":\"anon-sb-1\", \"messageId\":null, \"details\":\"DeviceMaximumQueueDepthExceeded\", \"routeName\":null, \"statusCode\":\"403\"}",
      "location": "westus"
    }
  ]
}
```

Here are more details on routing diagnostic logs:

- [List of routing diagnostic log error codes](#)
- [List of routing diagnostic logs operationNames](#)

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\":\""
        \"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>\"}",
        "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\":\"<authenticationType>\",\"scope\":\"device\", \"type\":\"sas\", \"issuer\":\"iothub\", \"acceptingIpFilterRule\":null}",
        "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 |
|--------------------------|---------|--|
| <code>messageSize</code> | Integer | The size of device-to-cloud message in bytes |

| PROPERTY | TYPE | DESCRIPTION |
|---------------------------|---|--|
| deviceId | String of ASCII 7-bit alphanumeric characters | The identity of the device |
| 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 |

Configurations

IoT Hub configuration logs tracks events and error for the Automatic Device Management feature set.

```
{
  "records": [
    {
      "time": "2019-09-24T17:21:52Z",
      "resourceId": "Resource Id",
      "operationName": "ReadManyConfigurations",
      "category": "Configurations",
      "resultType": "",
      "resultDescription": "",
      "level": "Information",
      "durationMs": "17",
      "properties": "{\"configurationId\": \"\", \"sdkVersion\": \"2018-06-30\", \"messageSize\": \"0\", \"statusCode\": null}",
      "location": "southcentralus"
    }
  ]
}
```

Device Streams (Preview)

The device streams category tracks request-response interactions sent to individual devices.

```
{  
  "records":  
  [  
    {  
      "time": "2019-09-19T11:12:04Z",  
      "resourceId": "Resource Id",  
      "operationName": "invoke",  
      "category": "DeviceStreams",  
      "resultType": "",  
      "resultDescription": "",  
      "level": "Information",  
      "durationMs": "74",  
      "properties": "{\"deviceId\":\"myDevice\",\"moduleId\":\"myModule\",\"sdkVersion\":\"2019-05-01-  
preview\", \"requestSize\": \"3\", \"responseSize\": \"5\", \"statusCode\": null, \"requestName\": \"myRequest\", \"d  
irection\": \"c2d\"}”,  
      "location": "Central US"  
    }  
  ]  
}
```

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

7/29/2020 • 6 minutes to read • [Edit Online](#)

This tutorial shows the steps you need to secure your Azure IoT hub using the *X.509 Certificate Authentication*. For the purpose of illustration, we 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 2017](#) or [Visual Studio 2019](#) installed.

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 any of the following ways to get your certificates:

- Purchase X.509 certificates from a *root certificate authority (CA)*. This method is recommended for production environments.
- Create your own X.509 certificates using a third-party tool such as [OpenSSL](#). This technique is 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](#).
- Generate an [X.509 intermediate CA certificate](#) signed by an existing root CA certificate and upload it to the hub. Once the intermediate certificate is uploaded and verified, as instructed below, it can be used in the place of a root CA certificate mentioned below. Tools like OpenSSL ([openssl req](#) and [openssl ca](#)) can be used to generate and sign an intermediate CA certificate.

NOTE

Do not upload the 3rd party root if it is not unique to you because that would enable other customers of the 3rd party to connect their devices to your IoT Hub.

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 select **Settings > Certificates** for the hub.
2. Select **Add** to add a new certificate.
3. In **Certificate Name**, enter a friendly display name, and select the certificate file you created in the

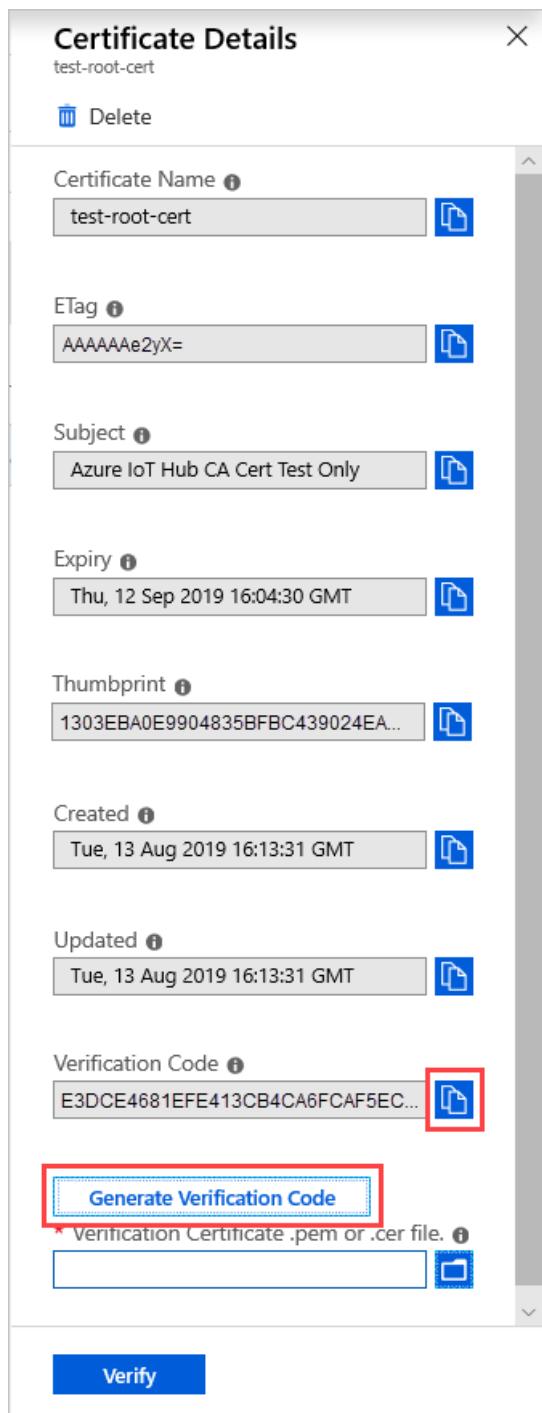
previous section from your computer.

- Once you get a notification that your certificate is successfully uploaded, select **Save**.

The screenshot shows the Microsoft Azure portal interface. On the left, there's a sidebar with various service icons and a 'Certificates' item under 'IoT Hub'. The main content area shows a 'Certificates' blade for the 'iot-hub-contoso-one' resource. At the top right of this blade is a 'Add Certificate' dialog. Inside the dialog, there are fields for 'Certificate Name' (containing 'test-root-cert') and 'Certificate .pem or .cer file' (containing 'azure-iot-test-only.root.ca.cert.pem'). A red box highlights the 'Save' button at the bottom right of the dialog. The status bar at the bottom of the portal window says 'No results'.

Your certificate appears in the certificates list with status of **Unverified**.

- Select the certificate that you just added to display **Certificate Details**, and then select **Generate Verification Code**.



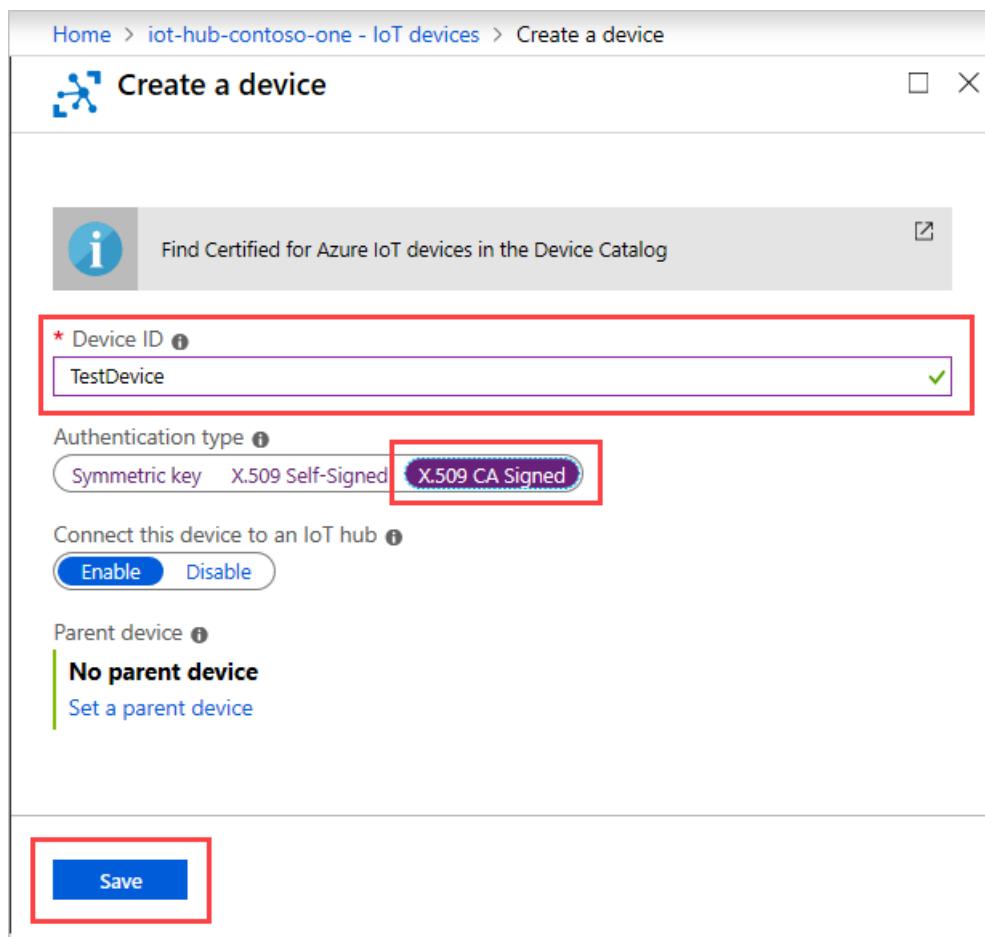
6. Copy the **Verification Code** to the clipboard. You use it to validate the certificate ownership.
7. Follow Step 3 in [Managing test CA certificates for samples and tutorials](#). This process signs your verification code with the private key associate with your X.509 CA certificate, which generates a signature. There are tools available to perform this signing process, for example, OpenSSL. This process is known as the [Proof of possession](#).
8. In **Certificate Details**, under **Verification Certificate .pem or .cer file**, find and open the signature file. Then select **Verify**.

The status of your certificate changes to **Verified**. Select **Refresh** if the certificate does not update automatically.

Create an X.509 device for your IoT hub

1. In the Azure portal, navigate to your IoT hub, and then select **Explorers > IoT devices**.
2. Select **New** to add a new device.

3. In Device ID, enter a friendly display name. For Authentication type, choose X.509 CA Signed, and then select Save.



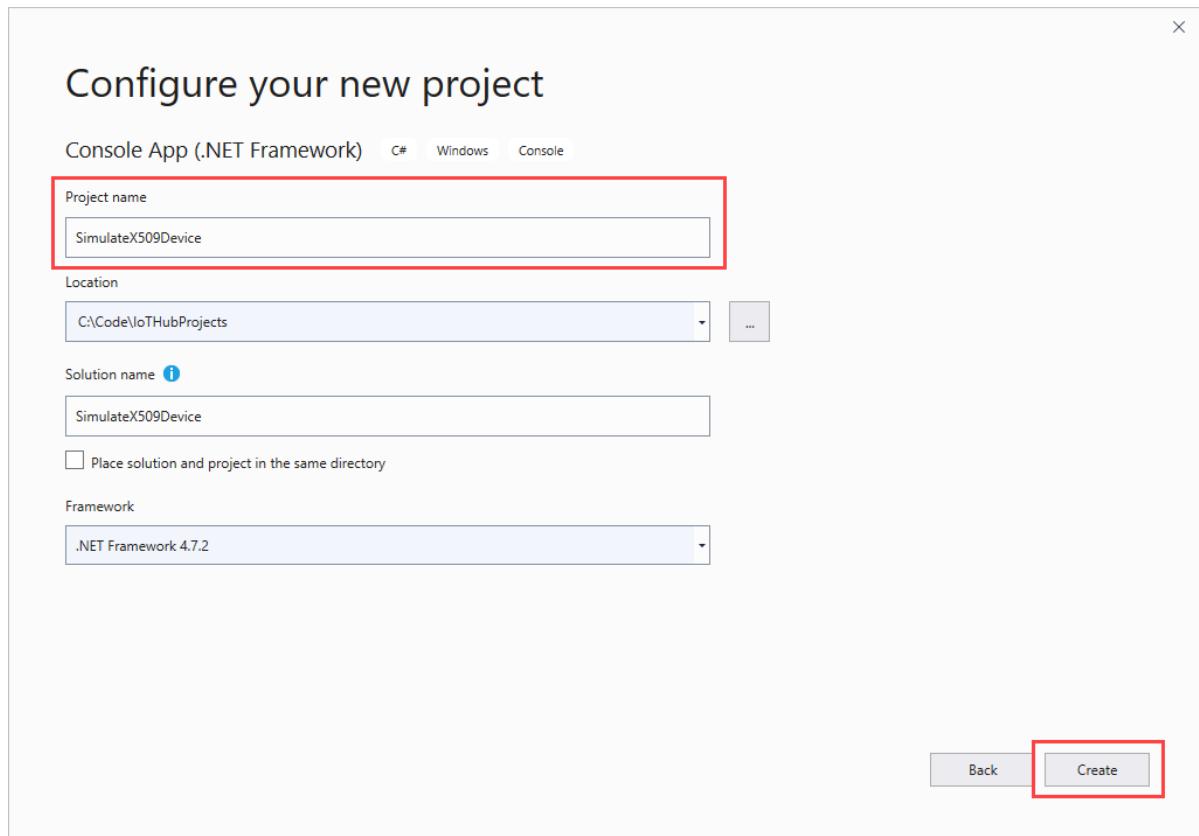
Authenticate your X.509 device with the X.509 certificates

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 result is a certificate chain from the CA certificate to the device's leaf certificate.

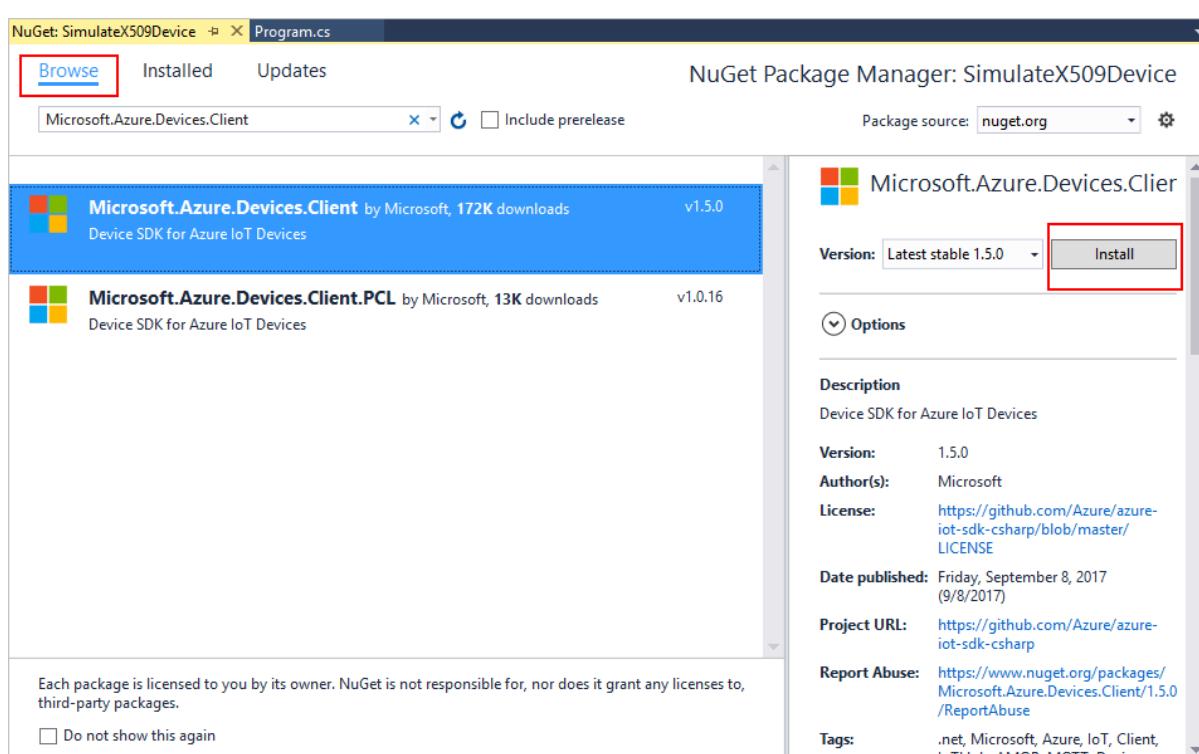
Step 4 in [Managing test CA certificates for samples and tutorials](#) generates a device certificate.

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. 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. Open Visual Studio, select **Create a new project**, and then choose the **Console App (.NET Framework)** project template. Select **Next**.
2. In **Configure your new project**, name the project *SimulateX509Device*, and then select **Create**.



3. In Solution Explorer, right-click the **SimulateX509Device** project, and then select **Manage NuGet Packages**.
4. In the **NuGet Package Manager**, select **Browse** and search for and choose **Microsoft.Azure.Devices.Client**. Select **Install**.



This step downloads, installs, and adds a reference to the Azure IoT device SDK NuGet package and its dependencies.

5. 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.Security.Cryptography.X509Certificates;
```

6. Add the following fields to 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>*.

7. 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);
    }
}
```

8. 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.

9. Run the app. Because this application accesses a *.pfx* file, you may need to run this app as an administrator.
 - a. Build the Visual Studio solution.
 - b. Open a new Command Prompt window by using **Run as administrator**.
 - c. Navigate to the folder that contains your solution, then navigate to the *bin/Debug* path within the solution folder.
 - d. Run the application **SimulateX509Device.exe** from the command prompt.

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...

```

Next steps

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

4/15/2019 • 2 minutes to read • [Edit Online](#)

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 [IoT Hub 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 contains the following navigation items:

- Overview
- Activity log
- Access control (IAM)
- Tags
- SETTINGS**
- Shared access policies
- Pricing and scale** (highlighted with a red box)
- Operations monitoring
- IP Filter
- Certificates
- Properties
- Locks
- 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 following settings:

- Overview
- Activity log
- Access control (IAM)
- Tags
- SETTINGS
- Shared access policies
- Pricing and scale** (highlighted with a blue background)
- Operations monitoring
- IP Filter
- Certificates

The main pane displays the following information:

- 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."
- A section titled "Pricing and scale tier" showing "S1 - Standard". This section is highlighted with a red box.
- A dropdown menu for "IoT Hub units" currently set to "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)

4/21/2020 • 11 minutes to read • [Edit Online](#)

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](#).

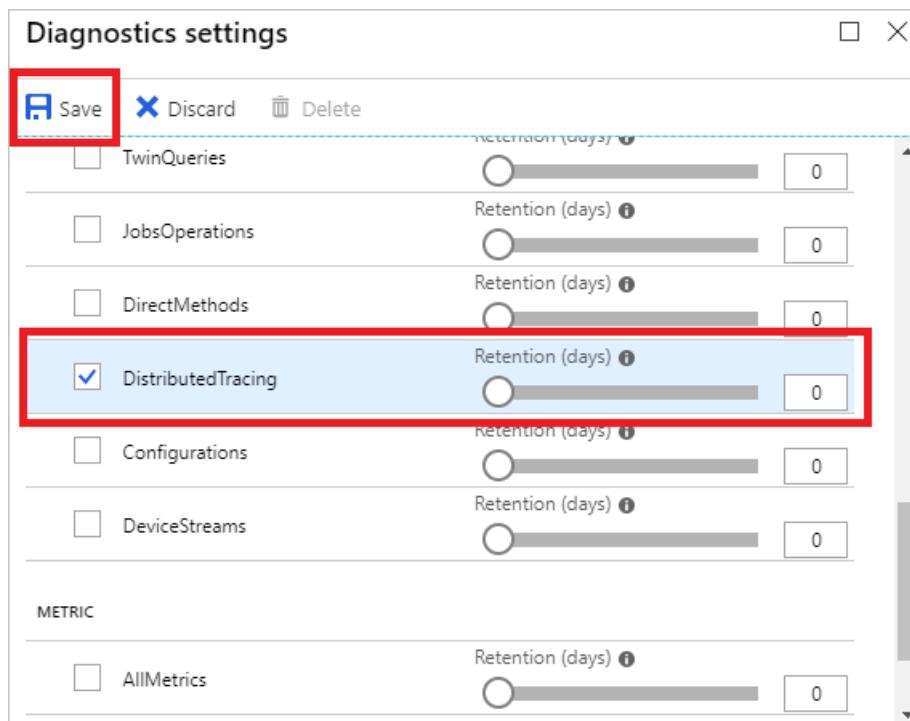
Configure IoT Hub

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 Visual Studio 2019. Visual Studio 2017 and 2015 are also supported.
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. Run the following commands to clone the latest release of the [Azure IoT C SDK GitHub repository](#):

```
git clone -b public-preview https://github.com/Azure/azure-iot-sdk-c.git
cd azure-iot-sdk-c
git submodule update --init
```

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. Run the following commands from the `azure-iot-sdk-c` directory:

```
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

GET THE SAMPLE ON
GITHUB

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.

5. 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

1. 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
```

2. Run the application. The device sends telemetry supporting distributed tracing.

```
Debug/iothub_ll_telemetry_sample.exe
```

3. Keep the app running. Optionally observe the message being sent to IoT Hub by looking at the console window.

Workaround for third-party clients

It's **not trivial** to preview the distributed tracing feature without using the C SDK. Thus, this approach is not recommended.

First, you must implement all the IoT Hub protocol primitives in your messages by following the dev guide [Create and read IoT Hub messages](#). Then, edit the protocol properties in the MQTT/AMQP messages to add `tracestate` as system property. Specifically,

- For MQTT, add `%24.tracestate=timestamp%3d1539243209` to the message topic, where `1539243209` should be replaced with the creation time of the message in the unix timestamp format. As an example, refer to the implementation [in the C SDK](#)
- For AMQP, add `key("tracestate")` and `value("timestamp=1539243209")` as message annotation. For a reference implementation, see [here](#).

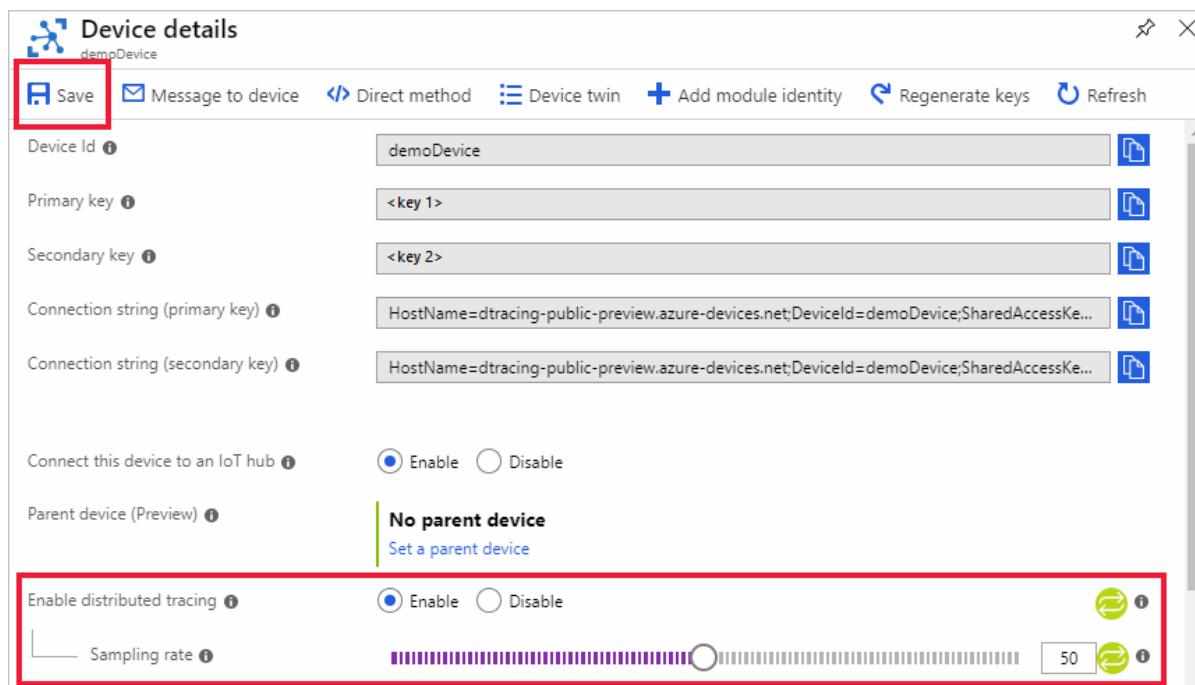
To control the percentage of messages containing this property, implement logic to listen to cloud-initiated events such as twin updates.

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**.



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.

```

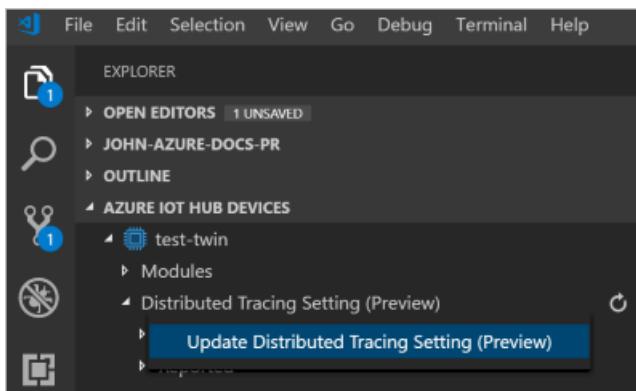
C:\Codes\e2e_diag\cmake\iothub_client\samples\iothub_ll_telemetry_sample\Release\iothub_ll_telemetry_sample.exe
-> 11:13:47 PUBLISH | IS_DUP: false | RETAIN: 0 | QOS: DELIVER_AT_LEAST_ONCE | TOPIC_NAME: devices/newDevice/messages/events/property_key=property_value | PACKET_ID: 24 | PAYLOAD_LEN: 12
<- 11:13:47 PUBACK | PACKET_ID: 23
confirmation callback received for message 16 with result IOTHUB_CLIENT_CONFIRMATION_OK
Sending message 18 to IoTHub
-> 11:13:48 PUBLISH | IS_DUP: false | RETAIN: 0 | QOS: DELIVER_AT_LEAST_ONCE | TOPIC_NAME: devices/newDevice/messages/events/property_key=property_value&%24.tracestate=timestamp%3d1547666028 | PACKET_ID: 25 | PAYLOAD_LEN: 12
<- 11:13:48 PUBACK | PACKET_ID: 24
confirmation callback received for message 17 with result IOTHUB_CLIENT_CONFIRMATION_OK
Sending message 19 to IoTHub
-> 11:13:49 PUBLISH | IS_DUP: false | RETAIN: 0 | QOS: DELIVER_AT_LEAST_ONCE | TOPIC_NAME: devices/newDevice/messages/events/property_key=property_value | PACKET_ID: 26 | PAYLOAD_LEN: 12
<- 11:13:49 PUBACK | PACKET_ID: 25
confirmation callback received for message 18 with result IOTHUB_CLIENT_CONFIRMATION_OK
Sending message 20 to IoTHub
-> 11:13:50 PUBLISH | IS_DUP: false | RETAIN: 0 | QOS: DELIVER_AT_LEAST_ONCE | TOPIC_NAME: devices/newDevice/messages/events/property_key=property_value&%24.tracestate=timestamp%3d1547666030 | PACKET_ID: 27 | PAYLOAD_LEN: 12
<- 11:13:50 PUBACK | PACKET_ID: 26
confirmation callback received for message 19 with result IOTHUB_CLIENT_CONFIRMATION_OK
Sending message 21 to IoTHub
-> 11:13:51 PUBLISH | IS_DUP: false | RETAIN: 0 | QOS: DELIVER_AT_LEAST_ONCE | TOPIC_NAME: devices/newDevice/messages/events/property_key=property_value | PACKET_ID: 28 | PAYLOAD_LEN: 12
<- 11:13:51 PUBACK | PACKET_ID: 27
confirmation callback received for message 20 with result IOTHUB_CLIENT_CONFIRMATION_OK
Sending message 22 to IoTHub
-> 11:13:52 PUBLISH | IS_DUP: false | RETAIN: 0 | QOS: DELIVER_AT_LEAST_ONCE | TOPIC_NAME: devices/newDevice/messages/events/property_key=property_value&%24.tracestate=timestamp%3d1547666032 | PACKET_ID: 29 | PAYLOAD_LEN: 12
<- 11:13:52 PUBACK | PACKET_ID: 28
confirmation callback received for message 21 with result IOTHUB_CLIENT_CONFIRMATION_OK

```

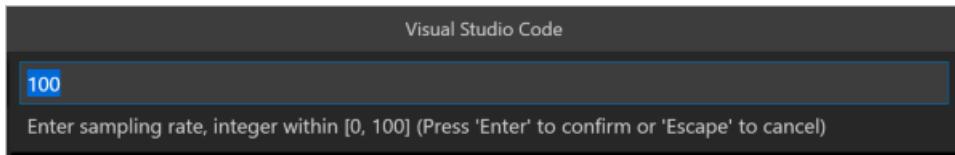
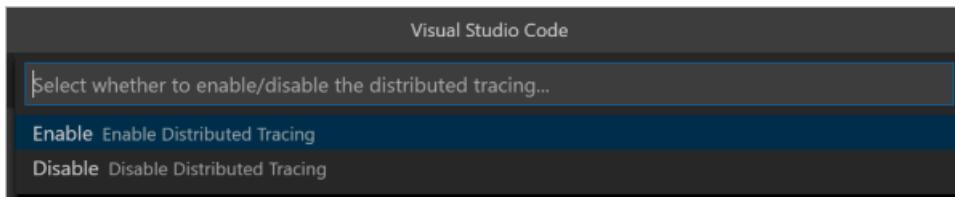
8. (Optional) Change the sampling rate to a different value, and observe the change in frequency that messages include `tracestate` in the application properties.

Update using Azure IoT Hub for VS Code

1. Install VS Code, then install the latest version of Azure IoT Hub 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 |
|---------------|----------|---------|---|
| sampling_mode | 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. |
| 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 | 00-8cd869a412459a25f5b4f31311223344-0144d2590aa cd909-01 | | {"deviceId":"AZ3166","messageSize":"96","callerLocalTimeUtc":"2018-02-22T03:27:28.633Z","calleeLocalTimeUtc":"2018-02-22T03:27:28.687Z"} |

| TIMEGENERATED | OPERATIONNAME | CATEGORY | LEVEL | CORRELATIONID | DURATIONMS | PROPERTIES |
|--------------------------|-------------------------|--------------------|---------------|---|------------|---|
| 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"} |

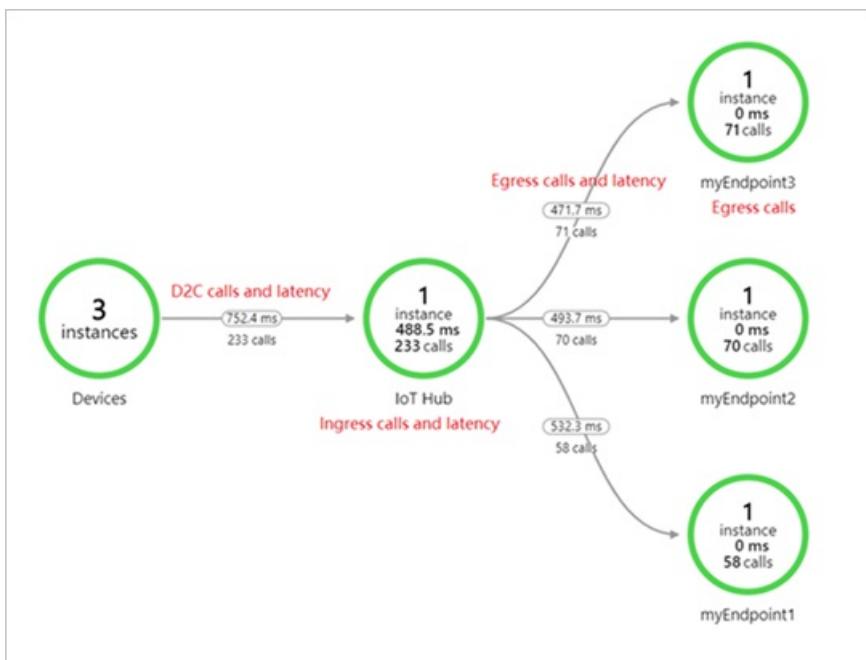
To understand the different types of logs, see [Azure IoT Hub diagnostic logs](#).

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 a globally unique `trace-id` for the message, a `span-id` for the "hop", and logs them to Azure Monitor diagnostic logs under the operation `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 operation `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

7/29/2020 • 5 minutes to read • [Edit Online](#)

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. However, clients reading directly from the [built-in Event Hub compatible endpoint](#) (not via the IoT Hub connection string) are not bound to the IP filter rules.

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. Rejecting IP addresses can prevent other Azure services such as Azure Stream Analytics, Azure Virtual Machines, or the Device Explorer in Azure portal from interacting with the IoT hub.

NOTE

If you must use Azure Stream Analytics (ASA) to read messages from an IoT hub with IP filter enabled, use the event hub-compatible name and endpoint of your IoT hub to manually add an [Event Hubs stream input](#) in the ASA.

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.

To get to the IP Filter settings page, select **Networking**, **Public access**, then choose **Selected IP Ranges**:

The screenshot shows the 'myIoTHub | Networking' blade in the Azure portal. The left sidebar has sections like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Events, Settings (Shared access policies, Pricing and scale, Networking), Certificates, Built-in endpoints, Failover, Properties, and Locks. The 'Networking' section is highlighted with a red box. The main area shows 'Public access' selected (also highlighted with a red box). It allows public network access to 'Selected IP ranges'. A button '+ Add IP Filter Rule' is visible.

Add or edit an IP filter rule

To add an IP filter rule, select **+ Add IP Filter Rule**.

This is a detailed view of the 'Add or edit an IP filter rule' dialog. It shows the 'Public access' tab selected. The 'Allow public network access to:' section has 'Selected IP ranges' selected. Below it, a note says 'IP filters block or allow specific IP address ranges. Your filters are applied in order. Drag and drop a filter to change its priority in the list.' A prominent blue button labeled '+ Add IP Filter Rule' is highlighted with a red box.

After selecting **Add IP Filter Rule**, fill in the fields.

Public access Private endpoint connections

 Save

 Revert

 Test

Allow public network access to:

- Disabled
- Selected IP ranges
- All networks

IP filters block or allow specific IP address ranges. Your filters are applied in order. Drag and drop a filter to change its priority in the list.

| NAME | ADDRESS RANGE | ACTION |
|--|---------------|---|
| MaliciousIP | 6.6.6.6/6 | <input checked="" type="checkbox"/> Block  |
|  Add IP Filter Rule | | |

- Provide a **name** for the IP Filter rule. This must be a unique, case-insensitive, alphanumeric string up to 128 characters long. Only the ASCII 7-bit alphanumeric characters plus `{'-', ':', '/', '\', '.', '+', '%', '_', '#', '*', '?', '!', '(', ')', ',', '=', '@', ';', '''}` are accepted.
- 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.
- Select **Allow** or **Block** as the **action** for the IP filter rule.

After filling in the fields, select **Save** to save the rule. You see an alert notifying you that the update is in progress.

Public access Private endpoint connections

 Save

 Revert

 Test

 Updating IoT Hub 

Allow public network access to:

- Disabled
- Selected IP ranges
- All networks

IP filters block or allow specific IP address ranges. Your filters are applied in order. Drag and drop a filter to change its priority in the list.

| NAME | ADDRESS RANGE | ACTION |
|--|---------------|---|
| MaliciousIP | 6.6.6.6/6 | <input checked="" type="checkbox"/> Block  |
|  Add IP Filter Rule | | |

The **Add** option is disabled when you reach the maximum of 10 IP filter rules.

To edit an existing rule, select the data you want to change, make the change, then select **Save** to save your edit.

Delete an IP filter rule

To delete an IP filter rule, select the trash can icon on that row and then select **Save**. The rule is removed and the change is saved.

The screenshot shows the 'IP filters' section of the Azure portal. At the top, there are two tabs: 'Public access' (selected) and 'Private endpoint connections'. Below the tabs are three buttons: 'Save' (highlighted with a red box), 'Revert', and 'Test'. The main area is titled 'Allow public network access to:' and contains three radio button options: 'Disabled', 'Selected IP ranges' (which is selected and highlighted with a blue circle), and 'All networks'. Below this, a note says: 'IP filters block or allow specific IP address ranges. Your filters are applied in order. Drag and drop a filter to change its priority in the list.' A table lists a single IP filter rule:

| NAME | ADDRESS RANGE | ACTION |
|--------------|---------------|---|
| MaliciousIP* | 6.6.6.6/6* | <input checked="" type="checkbox"/> Block |

At the bottom left is a button labeled '+ Add IP Filter Rule'.

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. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module 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**.

IP filters block or allow specific IP address ranges. Your filters are applied in order. Drag and drop a filter to change its priority in the list.

| NAME | ADDRESS RANGE | ACTION |
|------------------|------------------|--|
| AcceptMyRange | 192.168.100.0/22 | <input checked="" type="checkbox"/> Allow |
| MaliciousIP | 0.0.0.0/0 | <input type="checkbox"/> Block |
| | | |

Next steps

To further explore the capabilities of IoT Hub, see:

- [IoT Hub metrics](#)

Managing public network access for your IoT hub

7/29/2020 • 2 minutes to read • [Edit Online](#)

To restrict access to only [private endpoint for your IoT hub in your VNet](#), disable public network access. To do so, use Azure portal or the [publicNetworkAccess](#) API.

Turn off public network access using Azure portal

1. Visit [Azure portal](#)
2. Navigate to your IoT hub.
3. Select **Networking** from the left-side menu.
4. Under "Allow public network access to", select **Disabled**
5. Select **Save**.

The screenshot shows the Azure IoT Hub Networking settings page for 'MyIoTHub'. The left sidebar lists various settings like Overview, Activity log, and Networking (which is selected). The main area has tabs for 'Public access' and 'Private endpoint connections', with 'Public access' selected. Under 'Allow public network access to', the 'Disabled' option is selected. There are also options for 'Selected IP ranges' and 'All networks'. At the bottom right are 'Save', 'Revert', and 'Test' buttons.

To turn on public network access, select **Enabled**, then **Save**.

IP Filter

If public network access is disabled, all [IP Filter](#) rules are ignored. This is because all IPs from the public internet are blocked. To use IP Filter, use the [Selected IP ranges](#) option.

Bug fix with built-in Event Hub compatible endpoint

There is a bug with IoT Hub where the [built-in Event Hub compatible endpoint](#) continues to be accessible via public internet when public network access to the IoT Hub is disabled. To learn more and contact us about this bug, see [Disabling public network access for IoT Hub disables access to built-in Event Hub endpoint](#).

Automatic IoT device and module management using the Azure portal

7/29/2020 • 8 minutes to read • [Edit Online](#)

Automatic device management in Azure IoT Hub automates many of the repetitive and complex tasks of managing large device fleets. With automatic device management, you can target a set of devices based on their properties, define a desired configuration, and then let IoT Hub update the devices when they come into scope. This update is done using an *automatic device configuration* or *automatic module configuration*, which lets you summarize completion and compliance, handle merging and conflicts, and roll out configurations in a phased approach.

NOTE

The features described in this article are available only in the standard tier of IoT Hub. For more information about the basic and standard/free IoT Hub tiers, see [Choose the right IoT Hub tier](#).

Automatic device management works by updating a set of device twins or module twins with desired properties and reporting a summary that's based on twin reported properties. It introduces a new class and JSON document called a *Configuration* that has three parts:

- The **target condition** defines the scope of device twins or module twins to be updated. The target condition is specified as a query on twin tags and/or reported properties.
- The **target content** defines the desired properties to be added or updated in the targeted device twins or module 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 twin reported properties. System metrics are the default metrics that measure twin update status, such as the number of twins that are targeted and the number of twins that have been successfully updated.

Automatic configurations run for the first time shortly after the configuration is created and then at five minute intervals. Metrics queries run each time the automatic configuration runs.

Implement twins

Automatic device configurations require the use of device twins to synchronize state between the cloud and devices. For more information, see [Understand and use device twins in IoT Hub](#).

Automatic module configurations require the use of module twins to synchronize state between the cloud and modules. For more information, see [Understand and use module twins in IoT Hub](#).

Use tags to target twins

Before you create a configuration, you must specify which devices or modules you want to affect. Azure IoT Hub identifies devices and using tags in the device twin, and identifies modules using tags in the module twin. Each device or modules 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, 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 device configuration** or **Add module configuration**.

Home > MyIoTHub - IoT device configuration

MyIoTHub - IoT device configuration

IoT Hub

Search (Ctrl+)

+ Add Device Configuration + Add Module Configuration Refresh Delete

Properties Locks Export template

Explorers Query explorer IoT devices

Automatic Device Management

IoT Edge

IoT device configuration

Messaging

Device configurations provide the ability to perform IoT device configuration at scale. You can define configurations for modules and devices.

| Configuration Name | Configuration Type | Target Condition | Priority |
|--------------------|---------------------------|-------------------------|----------|
| mymoduletwinconfig | Module Twin Configuration | from devices.modules... | 100 |
| mydevicetwinconfig | Device Twin Configuration | tags.environment='test' | 100 |

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 defines the content to be set in targeted device or module twins. There are two inputs for each set of settings. The first is the 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, you could set the twin path to `properties.desired.chiller-water` and then provide the following JSON content:

```
{  
  "temperature": 66,  
  "pressure": 28  
}
```

Create Module Twin Configuration

MyIoTHub

- Name and Label
- Twin Settings**
- Metrics
- Target Modules
- Review + create

Module Twin Settings

Module twin paths define sections of desired properties to update. Entered values will subsequently be updated in targeted module twins.

Module Twin Property

Delete

Module Twin Property Content *

```

1  [
2    "temperature": 66,
3    "pressure": 28
4  ]

```

Add Desired Property

You can also set individual settings by specifying the entire twin path and providing the value with no brackets. For example, with the twin path `properties.desired.chiller-water.temperature`, set the content to `66`. Then create a new twin setting for the pressure property.

If two or more configurations target the same twin path, the content from the highest priority configuration will apply (priority is defined in step 4).

If you wish to remove an existing property, specify the property value to `null`.

You can add additional settings by selecting **Add Device Twin Setting** or **Add Module Twin Setting**.

Specify Metrics (optional)

Metrics provide summary counts of the various states that a device or module may report back after 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.

Each configuration can have up to five custom metrics.

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'
```

If you're building a metric to report on configured modules, select `moduleId` from `devices.modules`. For example:

```
SELECT deviceId, moduleId FROM devices.modules
WHERE properties.reported.lastDesiredStatus.code = 200
```

Target Devices

Use the tags property from your twins to target the specific devices or modules that should receive this configuration. You can also target twin reported properties.

Automatic device configurations can only target device twin tags, and automatic module configurations can only target module twin tags.

Since multiple configurations may target the same device or module, each configuration needs 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 or modules will be targeted with this configuration. The condition is based on twin tags or twin reported properties and should match the expression format.

For automatic device configuration, you can specify just the tag or reported property to target. For example, `tags.environment='test'` or `properties.reported.chillerProperties.model='4000x'`. You can specify `*` to target all devices.

For automatic module configuration, use a query to specify tags or reported properties from the modules registered to the IoT hub. For example, `from devices.modules where tags.environment='test'` or `from devices.modules where properties.reported.chillerProperties.model='4000x'`. The wildcard cannot be used to target all modules.

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 or modules.
 - **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 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 or modules that match the target condition.
 - **Metrics** - a list of system metrics and custom metrics. You can view a list of devices or modules that are counted for each metric by selecting the metric in the drop-down and then selecting **View Devices** or **View Modules**.
 - **Device Twin Settings** or **Module Twin Settings** - the 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 or modules.

If you update the target condition, the following updates occur:

- If a twin didn't meet the old target condition, but meets the new target condition and this configuration is the highest priority for that twin, then this configuration is applied.
- If a twin currently running this configuration no longer meets the target condition, the settings from the configuration will be removed and the twin will be modified by the next highest priority configuration.
- If a 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 to 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](#)

Automatic IoT device and module management using the Azure CLI

4/22/2020 • 9 minutes to read • [Edit Online](#)

Automatic device management in Azure IoT Hub automates many of the repetitive and complex tasks of managing large device fleets. With automatic device management, you can target a set of devices based on their properties, define a desired configuration, and then let IoT Hub update the devices when they come into scope. This update is done using an *automatic device configuration* or *automatic module configuration*, which lets you summarize completion and compliance, handle merging and conflicts, and roll out configurations in a phased approach.

NOTE

The features described in this article are available only in the standard tier of IoT Hub. For more information about the basic and standard/free IoT Hub tiers, see [Choose the right IoT Hub tier](#).

Automatic device management works by updating a set of device twins or module twins with desired properties and reporting a summary that's based on twin reported properties. It introduces a new class and JSON document called a *Configuration* that has three parts:

- The **target condition** defines the scope of device twins or module 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 or module 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 twin reported properties. System metrics are the default metrics that measure twin update status, such as the number of twins that are targeted and the number of twins that have been successfully updated.

Automatic configurations run for the first time shortly after the configuration is created and then at five minute intervals. Metrics queries run each time the automatic configuration runs.

CLI prerequisites

- An [IoT hub](#) in your Azure subscription.
- [Azure CLI](#) in your environment. At a minimum, your Azure CLI version must be 2.0.70 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](#).

NOTE

This article uses the newest version of the Azure IoT extension, called `azure-iot`. The legacy version is called `azure-cli-iot-ext`. You should only have one version installed at a time. You can use the command `az extension list` to validate the currently installed extensions.

Use `az extension remove --name azure-cli-iot-ext` to remove the legacy version of the extension.

Use `az extension add --name azure-iot` to add the new version of the extension.

To see what extensions you have installed, use `az extension list`.

Implement twins

Automatic device configurations require the use of device twins to synchronize state between the cloud and devices. For more information, see [Understand and use device twins in IoT Hub](#).

Automatic module configurations require the use of module twins to synchronize state between the cloud and modules. For more information, see [Understand and use module twins in IoT Hub](#).

Use tags to target twins

Before you create a configuration, you must specify which devices or modules you want to affect. Azure IoT Hub identifies devices and using tags in the device twin, and identifies modules using tags in the module twin. Each device or modules 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, 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 or module twin desired properties to set and reported properties to measure. To create an automatic configuration using Azure CLI, save the target content and metrics locally as .txt files. You use the file paths in a later section when you run the command to apply the configuration to your device.

Here's a basic target content sample for an automatic device configuration:

```
{  
  "content": {  
    "deviceContent": {  
      "properties.desired.chillerWaterSettings": {  
        "temperature": 38,  
        "pressure": 78  
      }  
    }  
  }  
}
```

Automatic module configurations behave very similarly, but you target `moduleContent` instead of `deviceContent`.

```
{
  "content": {
    "moduleContent": {
      "properties.desired.chillerWaterSettings": {
        "temperature": 38,
        "pressure": 78
      }
    }
  }
}
```

Here are examples of metric queries:

```
{
  "queries": {
    "Compliant": "select deviceId from devices where configurations.[[chillerdevicesettingswashington]].status = 'Applied' AND properties.reported.chillerWaterSettings.status='current'",
    "Error": "select deviceId from devices where configurations.[[chillerdevicesettingswashington]].status = 'Applied' AND properties.reported.chillerWaterSettings.status='error'",
    "Pending": "select deviceId from devices where configurations.[[chillerdevicesettingswashington]].status = 'Applied' AND properties.reported.chillerWaterSettings.status='pending'"
  }
}
```

Metric queries for modules are also similar to queries for devices, but you select for `moduleId` from `devices.modules`. For example:

```
{
  "queries": {
    "Compliant": "select deviceId, moduleId from devices.module where configurations.[[chillermodulesettingswashington]].status = 'Applied' AND properties.reported.chillerWaterSettings.status='current'"
  }
}
```

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 or modules will be targeted with this configuration. For automatic device 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.devicemodel='4000x'`. For automatic module configuration, the condition is based on module twin tags or module twin desired properties.. For example,


```
from devices.modules where tags.environment='test' or
from devices.modules where properties.reported.chillerProperties.model='4000x'.
```
- **--priority** - A positive integer. In the event that two or more configurations are targeted at the same device or module, 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 or module may report back after 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 or module twins in IoT Hub that match the targeting condition.
- **appliedCount** - A system metric specifies the number of devices or modules that have had the target content applied.
- **Your custom metric** - Any metrics you've defined are user metrics.

You can show a list of device IDs, module 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 or module 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 twin didn't meet the old target condition, but meets the new target condition and this configuration is the highest priority for that twin, then this configuration is applied.
- If a twin currently running this configuration no longer meets the target condition, the settings from the configuration will be removed and the twin will be modified by the next highest priority configuration.
- If a 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 or module twins take on their next highest priority configuration. If 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 to 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](#)

Import and export IoT Hub device identities in bulk

7/29/2020 • 13 minutes to read • [Edit Online](#)

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. To see a working sample in C# and learn how you can use this capability when cloning a hub to a different region, see [How to Clone an IoT Hub](#).

NOTE

IoT Hub has recently added virtual network support in a limited number of regions. This feature secures import and export operations and eliminates the need to pass keys for authentication. Initially, virtual network support is available only in these regions: *WestUS2*, *EastUS*, and *SouthCentralUS*. To learn more about virtual network support and the API calls to implement it, see [IoT Hub Support for virtual networks](#).

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));  
}
```

NOTE

If your storage account has firewall configurations that restrict IoT Hub's connectivity, consider using [Microsoft trusted first party exception](#) (available in select regions for IoT hubs with managed service identity).

Device import/export job limits

Only 1 active device import or export job is allowed at a time for all IoT Hub tiers. IoT Hub also has limits for rate of jobs operations. To learn more, see [Reference - IoT Hub quotas and throttling](#).

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 (SAS). For more information about shared access signatures, see [Grant limited access to Azure Storage resources using shared access signatures \(SAS\)](#).

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:

| IMPORT MODE | DESCRIPTION |
|--|--|
| <code>createOrUpdate</code> | <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 <code>ETag</code> 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> |
| <code>create</code> | <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> |
| <code>update</code> | <p>If a device already exists with the specified ID, existing information is overwritten with the provided input data without regard to the <code>ETag</code> value.</p> <p>If the device does not exist, an error is written to the log file.</p> |
| <code>updateIfMatchETag</code> | <p>If a device already exists with the specified ID, existing information is overwritten with the provided input data only if there is an <code>ETag</code> match.</p> <p>If the device does not exist, an error is written to the log file.</p> <p>If there is an <code>ETag</code> mismatch, an error is written to the log file.</p> |
| <code>createOrUpdateIfMatchETag</code> | <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 <code>ETag</code> match.</p> <p>If there is an <code>ETag</code> 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> |

| IMPORTMODE | DESCRIPTION |
|--------------------------|---|
| delete | If a device already exists with the specified ID, it is deleted without regard to the ETag value. If the device does not exist, an error is written to the log file. |
| 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. Many of these operations, including how to move devices from one hub to another, are used in the [Managing devices registered to the IoT hub section of How to Clone an IoT Hub](#).

The cloning article has a working sample associated with it, which is located in the IoT C# samples on this page: [Azure IoT Samples for C#](#), with the project being ImportExportDevicesSample. You can download the sample and try it out; there are instructions in the [How to Clone an IoT Hub](#) article.

To learn more about managing Azure IoT Hub, check out the following articles:

- [IoT Hub metrics](#)
- [IoT Hub logs](#)

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)

7/29/2020 • 6 minutes to read • [Edit Online](#)

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](#).



START RASPBERRY PI SIMULATOR

If you have physical devices, visit [Connect Raspberry Pi to Azure IoT Hub](#) to get started.

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.

START RASPBERRY PI
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

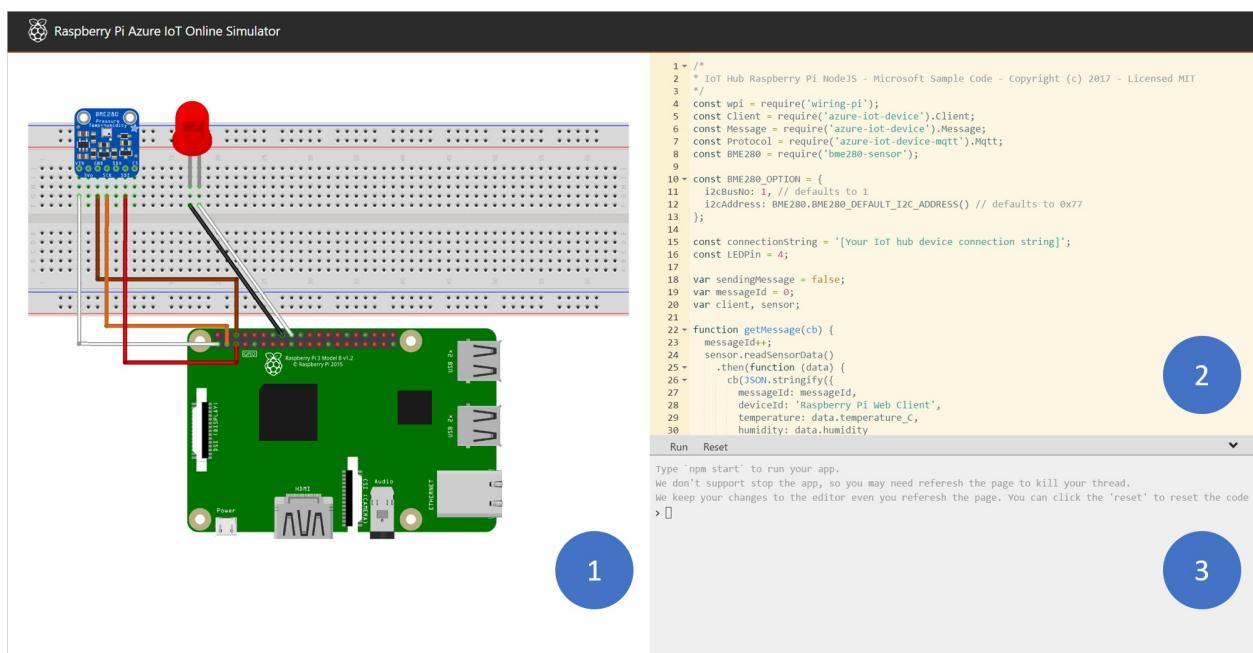
helps to collect sensor data from BME280 sensor and sends to your Azure IoT Hub. The application is fully compatible with real Pi devices.

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. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter **IoT Hub** in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.

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

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

The screenshot shows the 'Basics' tab of the 'Create IoT hub' wizard. A red box highlights the project details section, which includes fields for Subscription, Resource group, Region, and IoT hub name. Another red box highlights the 'Next: Size and scale >' button at the bottom.

Home > New > IoT hub

IoT hub
Microsoft

Basics [Size and scale](#) [Tags](#) [Review + create](#)

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

Project details

Choose the subscription you'll use to manage deployments and costs. Use resource groups like folders to help you organize and manage resources.

Subscription * ⓘ

Resource group * ⓘ [Create new](#)

Region * ⓘ

IoT hub name * ⓘ

[Review + create](#) [< Previous](#) **Next: Size and scale >** [Automation options](#)

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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](#)

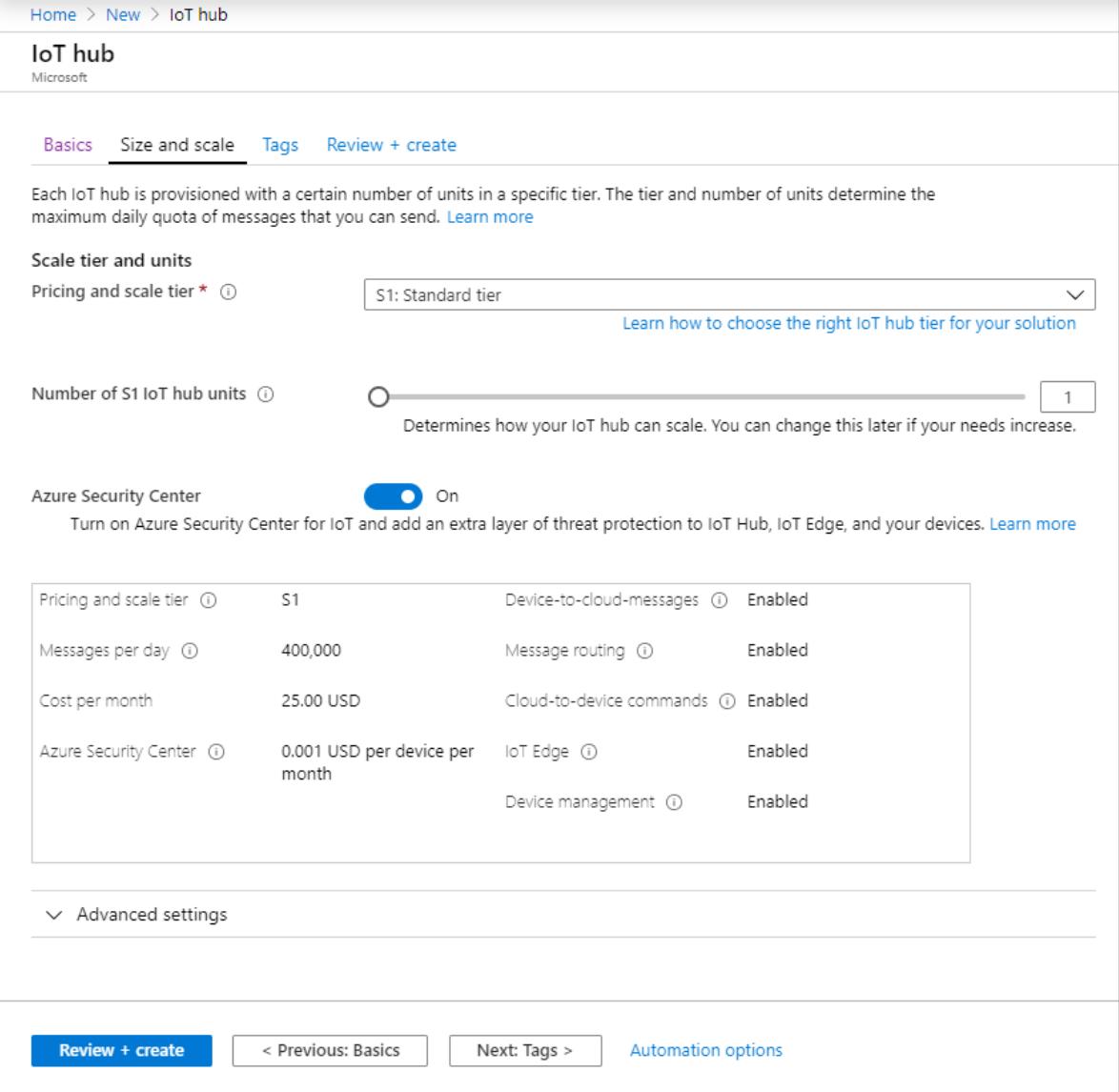
Number of S1 IoT hub units ⓘ 1
Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center On
Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)



You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.
If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure](#)

resources.

The screenshot shows the 'Tags' section of the IoT hub creation wizard. It displays a table with two rows of tag entries. The first row has 'Name' as 'department' and 'Value' as 'accounting'. The second row is empty. Below the table are navigation buttons: 'Review + create' (highlighted in blue), '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

| Name | Value | Resource |
|------------|------------|----------|
| department | accounting | IoT Hub |
| | | IoT Hub |

Review + create < Previous: Size and scale Next: Review + create > Automation options

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' page for the IoT hub. It lists the configuration details under three sections: Basics, Size and scale, and Tags. The 'Review + create' button is highlighted with a red box. At the bottom, the 'Create' button is also highlighted with a red box.

Basics

- Subscription: Personal testing
- Resource group: iot-hubs
- Region: West US 2
- IoT hub name: you-hub-name

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
- Azure Security Center: 0.001 USD per device per month

Tags

- department: accounting

Create < Previous: Tags Next > Automation options

8. Select **Create** to create your new hub. Creating the hub takes a few minutes.

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 a

hub unless it has an entry in the identity registry. For more information, see the [IoT Hub developer guide](#).

1. In your IoT hub navigation menu, open **IoT Devices**, then select **New** to add a device in your IoT hub.

The screenshot shows the Azure IoT Hub management interface for the 'iot-hub-contoso-one' hub. The top navigation bar includes 'Home > All resources > iot-hub-contoso-one - IoT devices'. The main area is titled 'iot-hub-contoso-one - IoT devices' with a sub-section 'IoT Hub'. A search bar and a red box highlight the '+ New' button. Below the search bar is a note: 'View, create, delete, and update devices in your IoT Hub.' There's a query editor with fields for 'Field' (set to 'select or enter a property name'), 'Operator' (set to '='), and 'Value' (set to 'specify constraint value'). A link 'Switch to query editor' is available. The table below lists columns: DEVICE ID, STATUS, LAST ACTIVITY TIME (UTC), LAST STATUS UPDATE (UTC), AUTHENTICATION T..., and CLOUD ... with a single row 'No results'. The left sidebar contains several sections: Overview, Activity log, Access control (IAM), Tags, Events, Settings (with Shared access policies, Pricing and scale, IP Filter, Certificates, Built-in endpoints, Manual failover (preview), Properties, Locks, Export template), Explorers (Query explorer, IoT devices, highlighted with a red box), and Automatic Device Management.

2. In **Create a device**, provide a name for your new device, such as **myDeviceId**, and select **Save**. This action creates a device identity for your IoT hub.

Home > All resources > iot-hub-contoso-one - IoT devices > Create a device

Create a device

Find Certified for Azure IoT devices in the Device Catalog

* 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 ⓘ
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 **Primary Connection String** to use later.

Home > iot-hub-contoso-one - IoT devices > myDeviceId

myDeviceId
iot-hub-contoso-one

Save Message to Device Direct Method + Add Module Identity Device Twin Manage keys Refresh

| | | |
|--------------------------------|---|--|
| Device ID ⓘ | myDeviceId | |
| Primary Key ⓘ | HZAwv1PN3suNBkaiQU1UeElNB3j0= | |
| Secondary Key ⓘ | G7615rzcbyWVfcflgmad55IGVa4l= | |
| Primary Connection String ⓘ | HostName=iot-hub-contoso-one.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=QdSim6i7cptUCeMYGVSeiRKOV22GFSJpbmyklVYM9df= | |
| Secondary Connection String ⓘ | HostName=iot-hub-contoso-one.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=q32joXuwffExbbqKYkjy8sF82qZlnqzGZspqkl2nqz= | |
| Enable connection to IoT Hub ⚡ | <input checked="" type="radio"/> Enable <input type="radio"/> Disable | |
| Parent device ⓘ | No parent device | |

Module Identities Configurations

MODULE ID CONNECTION STATE CONNECTION STATE LAST UPDATED (UTC) LAST ACTIVITY TIME (UTC)

There are no module identities for this 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](#).

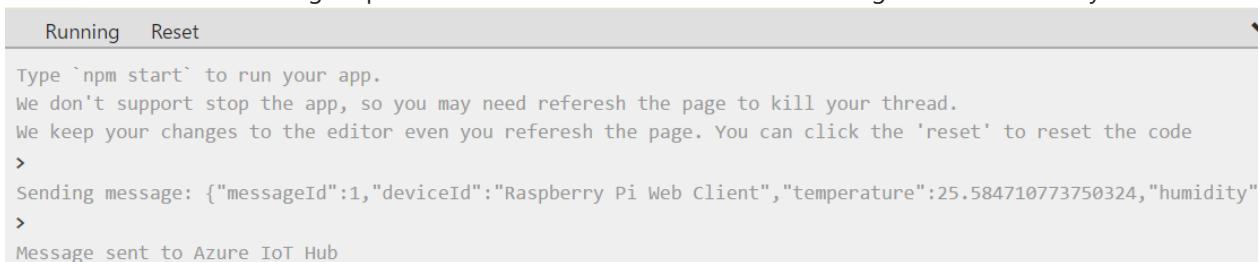
Run a sample application on Pi web simulator

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;
2. 17
```

3. Select **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



The screenshot shows a web-based IoT development environment. At the top, there are two buttons: "Running" and "Reset". Below them is a text area containing the following message:

```
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
```

Read the messages received by your hub

One way to monitor messages received by your IoT hub from the simulated device is to use the Azure IoT Tools for Visual Studio Code. To learn more, see [Use Azure IoT Tools for Visual Studio Code to send and receive messages between your device and IoT Hub](#).

For more ways to process data sent by your device, continue on to the next section.

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 extension for Visual Studio Code](#)
- [Manage devices with Azure IoT Hub extension for Visual Studio Code](#)
- [Set up message routing](#)
- [Use Power BI to visualize real-time sensor data from your IoT hub](#)
- [Use a web app 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

7/29/2020 • 2 minutes to read • [Edit Online](#)

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 |

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 extension |
| Manage your IoT device | Azure CLI and the IoT extension |
| Manage your IoT device | VS Code Azure IoT Hub 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)

7/29/2020 • 10 minutes to read • [Edit Online](#)

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. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.
 - **IoT Hub Name:** Enter a name for your hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

Home > New > IoT hub

IoT hub

Microsoft

[X](#)

[Basics](#) [Size and scale](#) [Tags](#) [Review + create](#)

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

Project details

Choose the subscription you'll use to manage deployments and costs. Use resource groups like folders to help you organize and manage resources.

Subscription * [Personal IoT items](#)

Resource group * [Create new](#)

Region * [East Asia](#)

IoT hub name * [Once your hub is created, this name can't be changed](#)

[Review + create](#) [< Previous](#) [Next: Size and scale >](#) [Automation options](#)

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

[Basics](#) [Size and scale](#) [Tags](#) [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](#)

Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center [On](#)

Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--|--------------------------------|--|---------|
| Pricing and scale tier ① | S1 | Device-to-cloud-messages ① | Enabled |
| Messages per day ① | 400,000 | Message routing ① | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ① | Enabled |
| Azure Security Center ① | 0.001 USD per device per month | IoT Edge ① | Enabled |
| | | Device management ① | Enabled |

[Advanced settings](#)

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.
If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure resources](#).

The screenshot shows the 'Tags' configuration page for an IoT hub. At the top, there are tabs for 'Basics', 'Size and scale', 'Tags' (which is underlined, indicating it is the active tab), and 'Review + create'. Below the tabs, a note states: 'Tags are name/value pairs. To categorize resources and consolidate billing, apply the same tag to multiple resources and resource groups. Your tags will update automatically if you change your resources.' A 'Learn more' link is provided. A table lists the current tags: one tag named 'department' with value 'accounting' is associated with an 'IoT Hub' resource, indicated by a blue icon. Another row shows a blank 'Name' field and a blank 'Value' field, also associated with an 'IoT Hub' resource. At the bottom, there are buttons for 'Review + create', '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' step of the IoT hub creation wizard. It displays the following configuration details:

| Setting | Value |
|----------------------------|--------------------------------|
| Subscription | Personal testing |
| Resource group | iot-hubs |
| Region | West US 2 |
| IoT hub name | you-hub-name |
| Pricing and scale tier | S1 |
| Number of S1 IoT hub units | 1 |
| Messages per day | 400,000 |
| Cost per month | 25.00 USD |
| Azure Security Center | 0.001 USD per device per month |
| Tags | department:accounting |

At the bottom, there are navigation buttons: 'Create' (highlighted), '< Previous: Tags', 'Next >', and 'Automation options'.

8. Select **Create** to create your new hub. Creating the hub takes a few minutes.

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 a hub unless it has an entry in the identity registry. For more information, see the [IoT Hub developer guide](#).

1. In your IoT hub navigation menu, open **IoT Devices**, then select **New** to add a device in your IoT hub.

The screenshot shows the 'IoT devices' blade in the Azure IoT hub. The left sidebar contains the following navigation items:

- Overview
- Activity log
- Access control (IAM)
- Tags
- Events
- Settings
 - Shared access policies
 - Pricing and scale (highlighted)
 - IP Filter
 - Certificates
 - Built-in endpoints
 - Manual failover (preview)
 - Properties
 - Locks
 - Export template
- Explorers
 - Query explorer
 - IoT devices (highlighted)
- Automatic Device Management

The main area shows a table with columns: DEVICE ID, STATUS, LAST ACTIVITY TIME (UTC), LAST STATUS UPDATE (UTC), AUTHENTICATION T..., and CLOUD A search bar and filter controls are also present.

2. In **Create a device**, provide a name for your new device, such as `myDeviceId`, and select **Save**. This action creates a device identity for your IoT hub.

The screenshot shows the 'Create a device' dialog in the Azure portal. The 'Device ID' field is highlighted with a red border and contains the value 'myDeviceId'. The 'Primary key' and 'Secondary key' fields are also visible. A 'Save' button at the bottom is also highlighted with a red border.

Home > All resources > iot-hub-contoso-one - IoT devices > Create a device

Create a device

Find Certified for Azure IoT devices in the Device Catalog

* 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 ⓘ
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 **Primary Connection String** to use later.

Device ID: myDeviceId

Primary Key: H2AwW1PN3uhBkaiQU1UeEINB3j0=

Secondary Key: G7615rzcbqyWFzcfIgma55lGVa4l=

Primary Connection String: HostName=iot-hub-contoso-one.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=QdSim6i7cptUCeMYGVSeRKOV2ZGFSJpbmykIVYM9df=

Secondary Connection String: HostName=iot-hub-contoso-one.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=q32joXuwlFExbqkYkj8sF82qZnqzGZspqkI2nqz=

Enable connection to IoT Hub: Enable Disable

Parent device: No parent device

Module Identities **Configurations**

| MODULE ID | CONNECTION STATE | CONNECTION STATE LAST UPDATED (U...) | LAST ACTIVITY TIME (UTC) |
|---|------------------|--------------------------------------|--------------------------|
| There are no module identities for this 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](#).

Set up Raspberry Pi

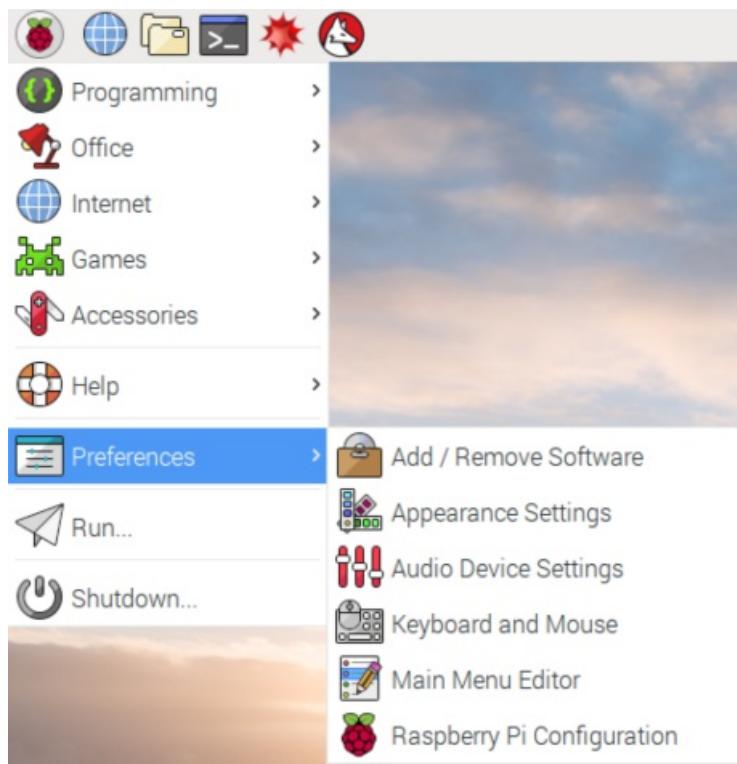
Install the Raspbian operating system for Pi

Prepare the microSD card for installation of the Raspbian image.

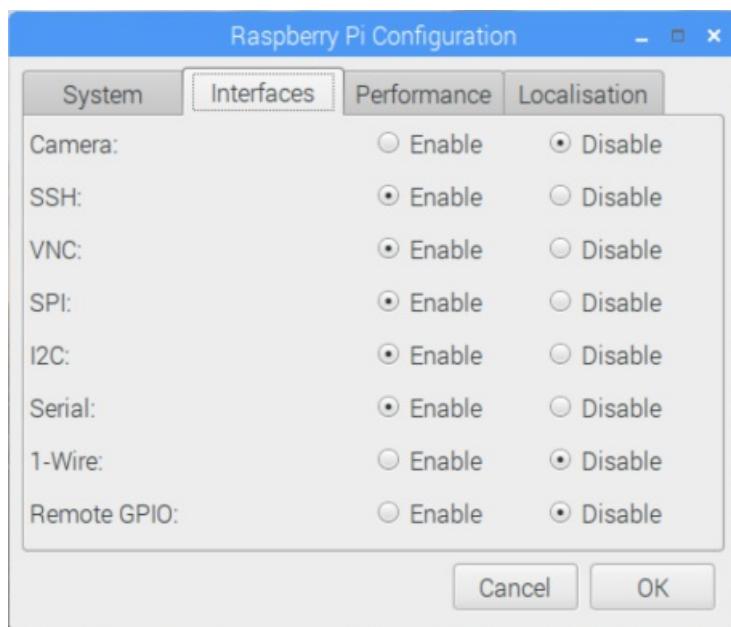
1. Download Raspbian.
 - a. [Raspbian Buster with desktop](#) (the .zip file).
 - b. Extract the Raspbian image to a folder on your computer.
2. Install Raspbian to the microSD card.
 - a. [Download and install the Etcher SD card burner utility](#).
 - b. Run Etcher and select the Raspbian image that you extracted in step 1.
 - c. Select the microSD card drive. 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 sign into 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.

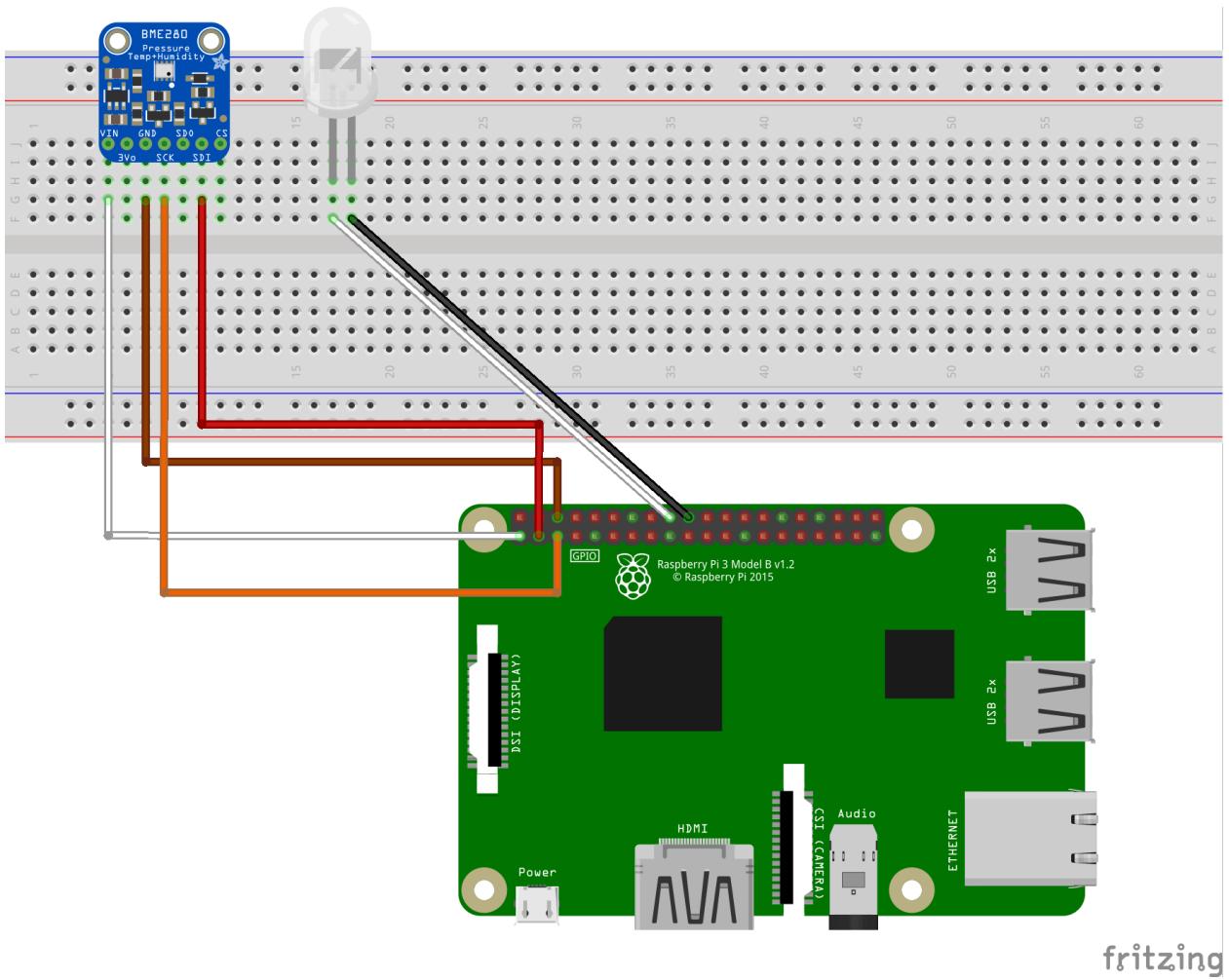


NOTE

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](#).



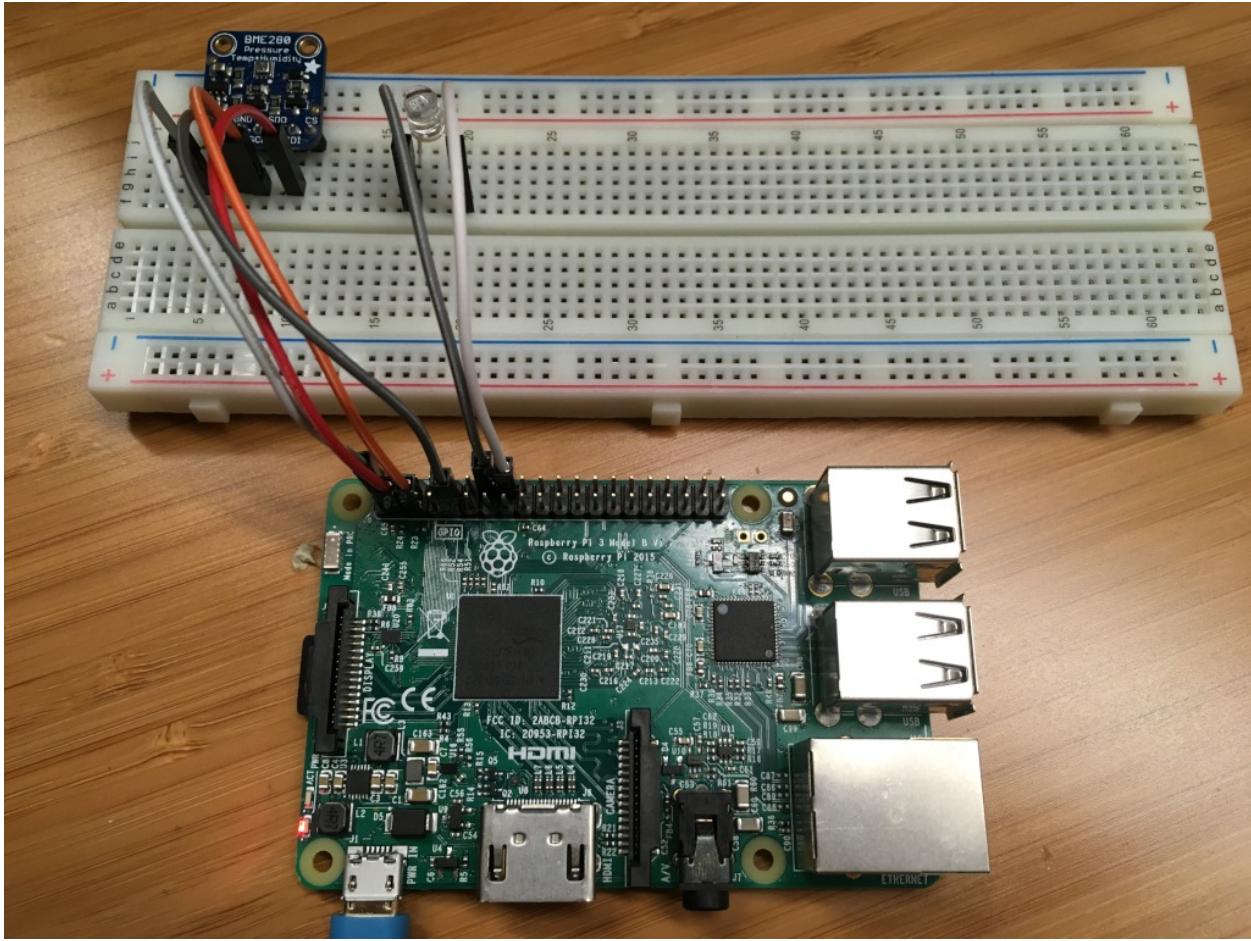
The BME280 sensor can collect temperature and humidity data. The LED blinks when the device sends a message to the cloud.

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.

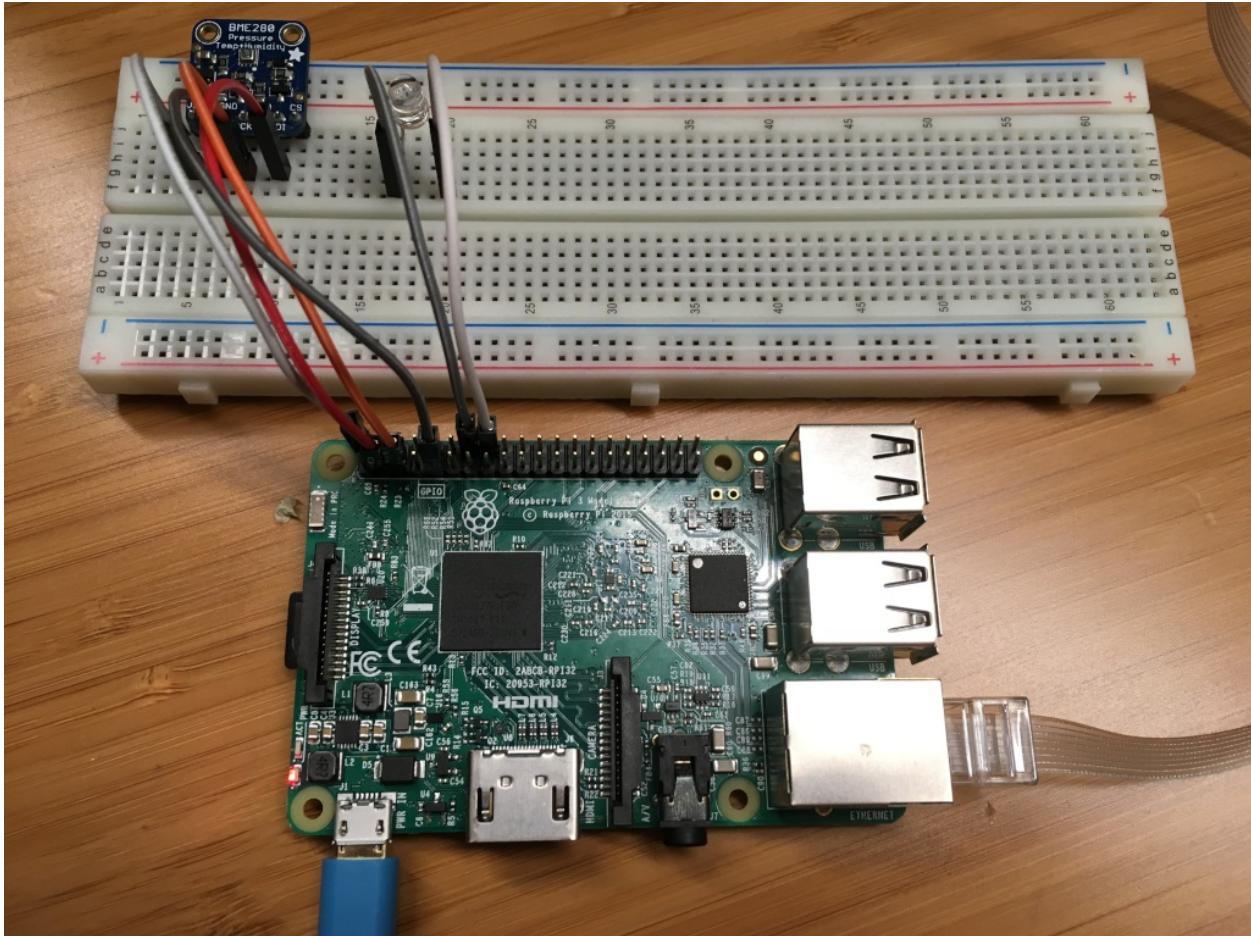
After you've successfully connected BME280 to your Raspberry Pi, it should be like below image.



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.

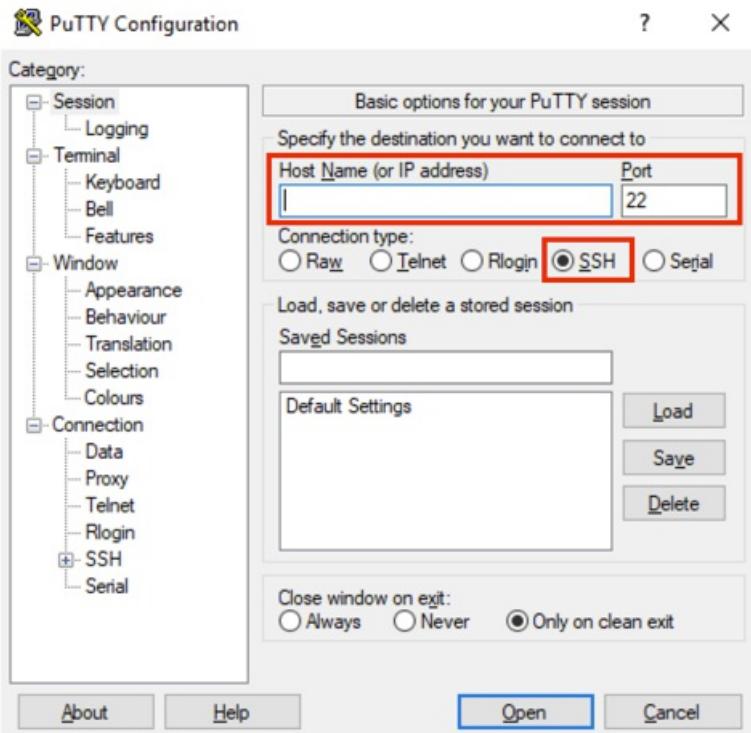
Run a sample application on Pi

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 Nodejs version.

```
node -v
```

If the version is lower than 10.x, or if there is no Node.js on your Pi, install the latest version.

```
curl -sL https://deb.nodesource.com/setup_10.x | sudo -E bash  
sudo apt-get -y install nodejs
```

3. Clone the sample application.

```
git clone https://github.com/Azure-Samples/azure-iot-samples-node.git
```

4. Install all packages for the sample. The installation includes Azure IoT device SDK, BME280 Sensor library, and Wiring Pi library.

```
cd azure-iot-samples-node/iot-hub/Tutorials/RaspberryPiApp  
npm install
```

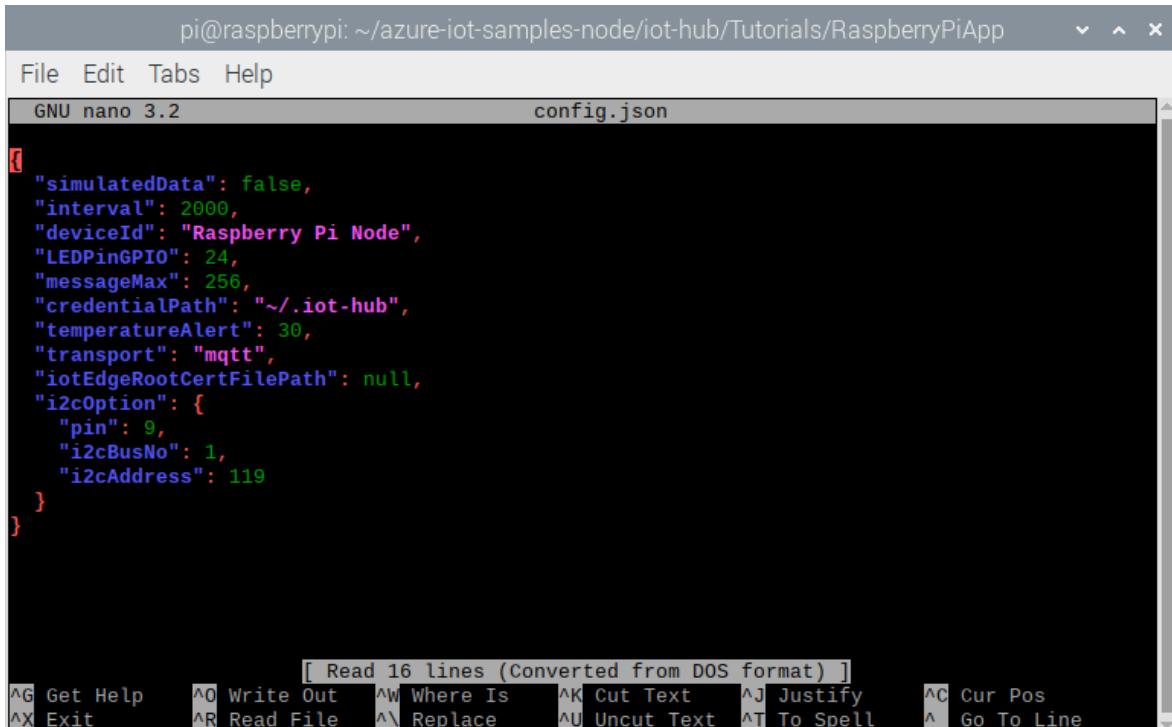
NOTE

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: ~/azure-iot-samples-node/iot-hub/Tutorials/RaspberryPiApp
```

```
File Edit Tabs Help
```

```
GNU nano 3.2 config.json
```

```
{ "simulatedData": false, "interval": 2000, "deviceId": "Raspberry Pi Node", "LEDPinGPIO": 24, "messageMax": 256, "credentialPath": "~/.iot-hub", "temperatureAlert": 30, "transport": "mqtt", "iotEdgeRootCertFilePath": null, "i2cOption": { "pin": 9, "i2cBusNo": 1, "i2cAddress": 119 } }
```

```
[ Read 16 lines (Converted from DOS format) ]
```

```
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos  
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line
```

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.

Note: The i2c address used in this tutorial is 0x77 by default. Depending on your configuration it might also be 0x76: if you encounter an i2c error, try to change the value to 118 and see if that works better. To see what address is used by your sensor, run `sudo i2cdetect -y 1` in a shell on the raspberry pi

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.

```
1. 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=d0q1tgHj6U8Wb+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.
```

Read the messages received by your hub

One way to monitor messages received by your IoT hub from your device is to use the Azure IoT Tools for Visual Studio Code. To learn more, see [Use Azure IoT Tools for Visual Studio Code to send and receive messages between your device and IoT Hub](#).

For more ways to process data sent by your device, continue on to the next section.

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 extension for Visual Studio Code](#)
- [Manage devices with Azure IoT Hub extension for Visual Studio Code](#)
- [Set up message routing](#)
- [Use Power BI to visualize real-time sensor data from your IoT hub](#)
- [Use a web app 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)

7/29/2020 • 9 minutes to read • [Edit Online](#)

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 supports simulated sensor data.

Create an IoT hub

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

1. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these limited features, you must select one of the supported regions.
 - **IoT Hub Name:** Enter a name for your hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

Home > New > IoT hub

IoT hub

Microsoft

X

Basics Size and scale Tags Review + create

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

Project details

Choose the subscription you'll use to manage deployments and costs. Use resource groups like folders to help you organize and manage resources.

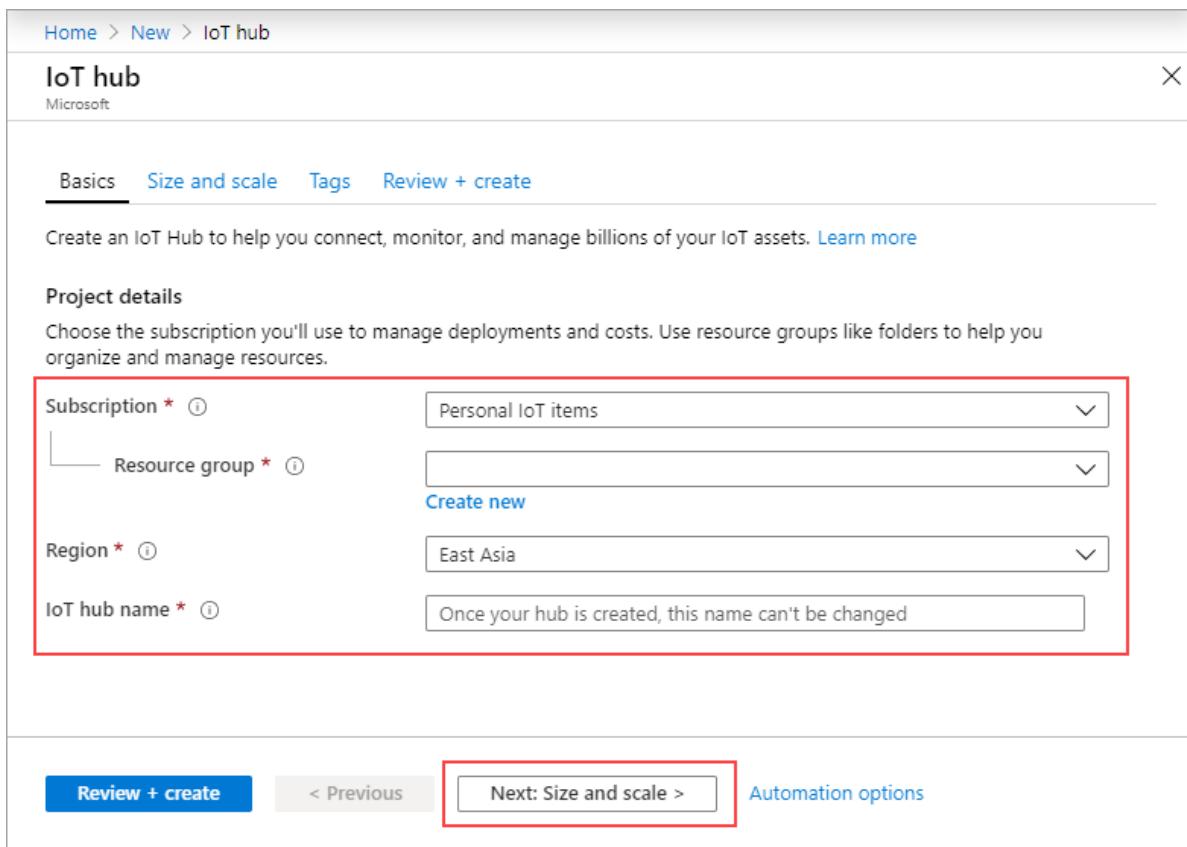
Subscription * ⓘ Personal IoT items

Resource group * ⓘ Create new

Region * ⓘ East Asia

IoT hub name * ⓘ Once your hub is created, this name can't be changed

Review + create < Previous **Next: Size and scale >** Automation options



5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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
 Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center [On](#) Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.

If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure](#)

resources.

The screenshot shows the 'Tags' step in the IoT hub creation wizard. At the top, there are tabs for 'Basics', 'Size and scale', 'Tags' (which is underlined in blue), and 'Review + create'. Below the tabs, a note says: 'Tags are name/value pairs. To categorize resources and consolidate billing, apply the same tag to multiple resources and resource groups. Your tags will update automatically if you change your resources.' A 'Learn more' link is provided. A table lists two tags: 'department' with value 'accounting' and another tag row with empty fields. At the bottom are buttons for 'Review + create' (highlighted in blue), '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

The screenshot shows the 'Review + create' step in the IoT hub creation wizard. At the top, there are tabs for 'Basics', 'Size and scale', 'Tags' (underlined in blue), and 'Review + create' (highlighted with a red dashed box). Below the tabs, sections show configuration details:

- Basics:** Subscription (Personal testing), Resource group (iot-hubs), Region (West US 2), IoT hub name (you-hub-name).
- 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), Azure Security Center (0.001 USD per device per month).
- Tags:** department (accounting).

At the bottom are buttons for 'Create' (highlighted with a red box), '< Previous: Tags', 'Next >', and 'Automation options'.

8. Select **Create** to create your new hub. Creating the hub takes a few minutes.

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 a

hub unless it has an entry in the identity registry. For more information, see the [IoT Hub developer guide](#).

1. In your IoT hub navigation menu, open **IoT Devices**, then select **New** to add a device in your IoT hub.

The screenshot shows the Azure IoT Hub management interface for the 'iot-hub-contoso-one' hub. The left sidebar contains a navigation menu with various options like Overview, Activity log, Access control (IAM), Tags, Events, Settings (Shared access policies, Pricing and scale, IP Filter, Certificates, Built-in endpoints, Manual failover (preview), Properties, Locks, Export template), Explorers (Query explorer), and Automatic Device Management (with 'IoT devices' highlighted). The main content area is titled 'iot-hub-contoso-one - IoT devices'. It features a search bar, a 'New' button (which is highlighted with a red box), and a 'Refresh' and 'Delete' button. Below these are sections for 'View, create, delete, and update devices in your IoT Hub.' and a 'Query devices' section with a query editor. The table below shows no results.

| DEVICE ID | STATUS | LAST ACTIVITY TIME (UTC) | LAST STATUS UPDATE (UTC) | AUTHENTICATION T... | CLOUD ... |
|------------|--------|--------------------------|--------------------------|---------------------|-----------|
| No results | | | | | |

2. In **Create a device**, provide a name for your new device, such as **myDeviceId**, and select **Save**. This action creates a device identity for your IoT hub.

Home > All resources > iot-hub-contoso-one - IoT devices > Create a device

Create a device

Find Certified for Azure IoT devices in the Device Catalog

* Device ID ✓

Authentication type Symmetric key X.509 Self-Signed X.509 CA Signed

* Primary key

* Secondary key

Auto-generate keys

Connect this device to an IoT hub Enable Disable

Parent device **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 **Primary Connection String** to use later.

Home > iot-hub-contoso-one - IoT devices > myDeviceId

myDeviceId
iot-hub-contoso-one

Save Message to Device Direct Method Add Module Identity Device Twin Manage keys Refresh

| | | |
|------------------------------|---|--|
| Device ID | myDeviceId | |
| Primary Key | HZAwv1PN3suNBkaiQU1UeEiNB3j0= | |
| Secondary Key | G7615rzcbyWVzcfTlgmad55lGVa4l= | |
| Primary Connection String | HostName=iot-hub-contoso-one.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=QdSim6i7cptUcemyGVSeiRKOV2ZGFSJpbmykIVYM9df= | |
| Secondary Connection String | HostName=iot-hub-contoso-one.azure-devices.net;DeviceId=myDeviceId;SharedAccessKey=q32joXuwIEbbqKYkj0sF82q2lnqzGZspqkl2nqz= | |
| Enable connection to IoT Hub | <input checked="" type="radio"/> Enable <input type="radio"/> Disable | |
| Parent device | No parent device | |

Module Identities Configurations

MODULE ID CONNECTION STATE CONNECTION STATE LAST UPDATED (UTC) LAST ACTIVITY TIME (UTC)

There are no module identities for this 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](#).

Set up Raspberry Pi

Now set up the Raspberry Pi.

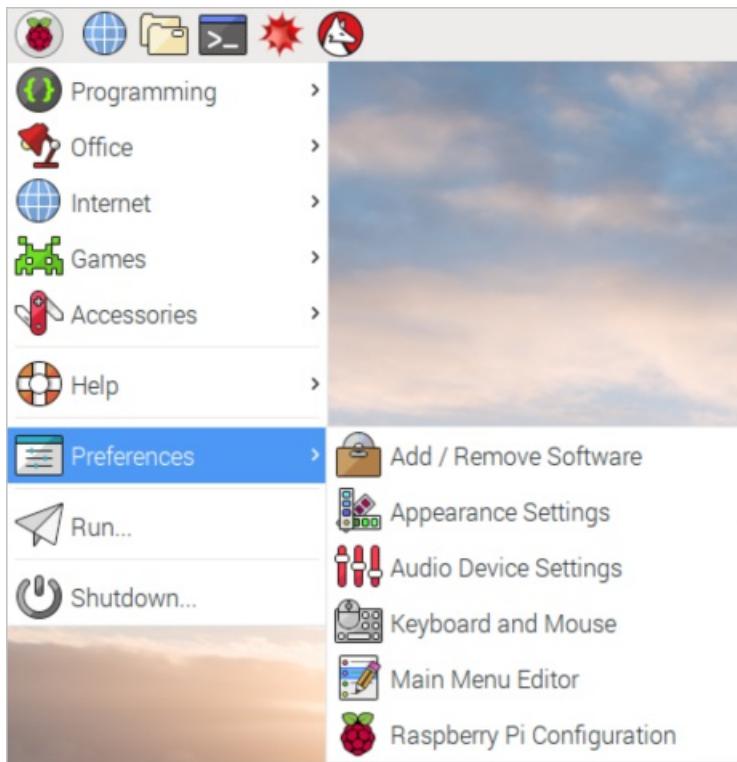
Install the Raspbian operating system for Pi

Prepare the microSD card for installation of the Raspbian image.

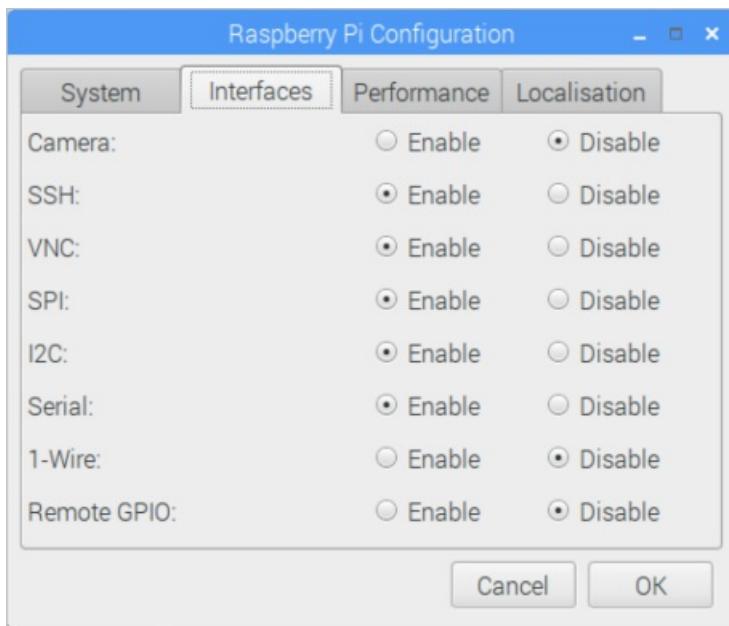
1. Download Raspbian.
 - a. [Download Raspbian Stretch with Desktop](#) (the .zip file).
 - b. Extract the Raspbian image to a folder on your computer.
2. Install Raspbian to the microSD card.
 - a. [Download and install the Etcher SD card burner utility](#).
 - b. Run Etcher and select the Raspbian 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 sign in to 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.

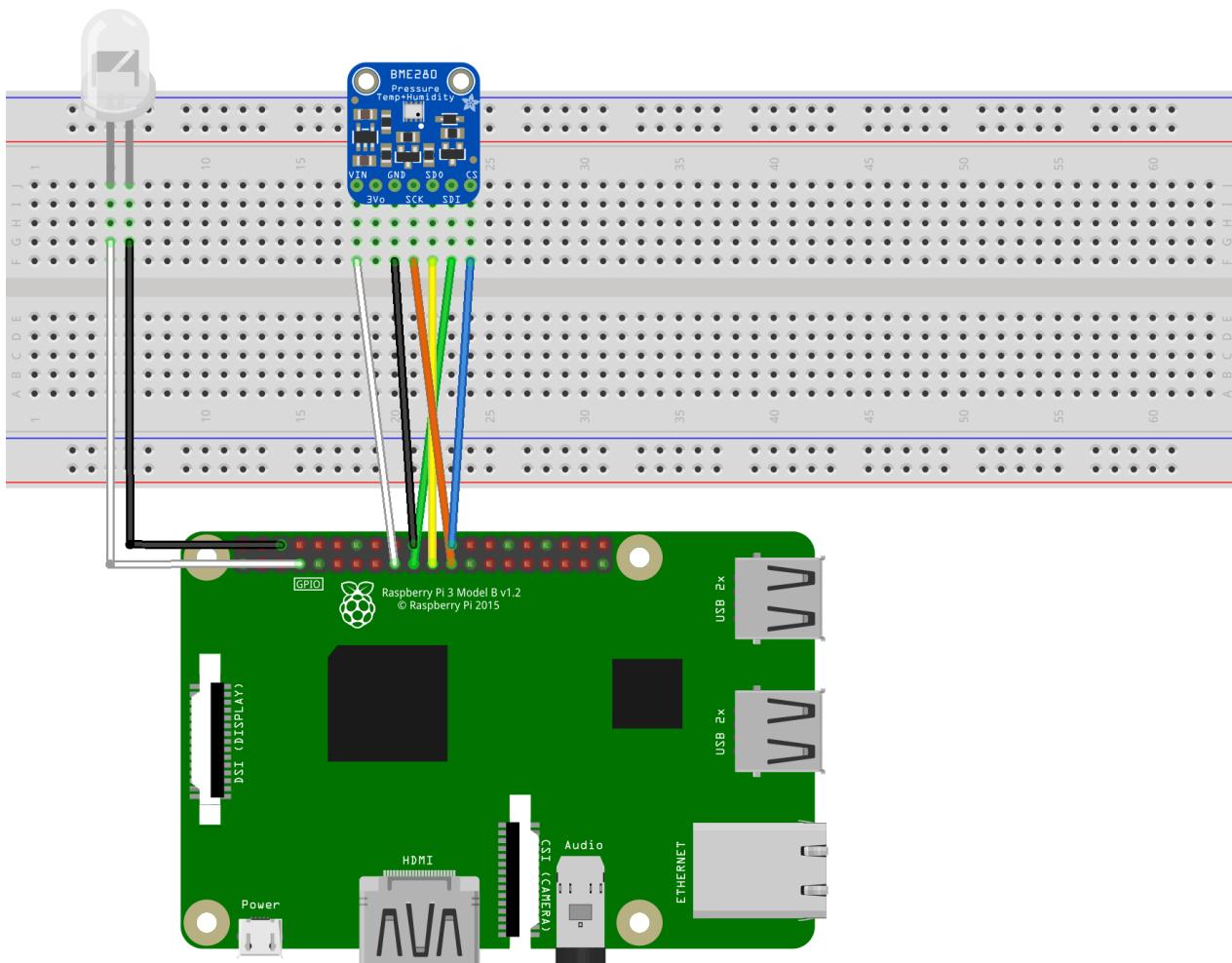


NOTE

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

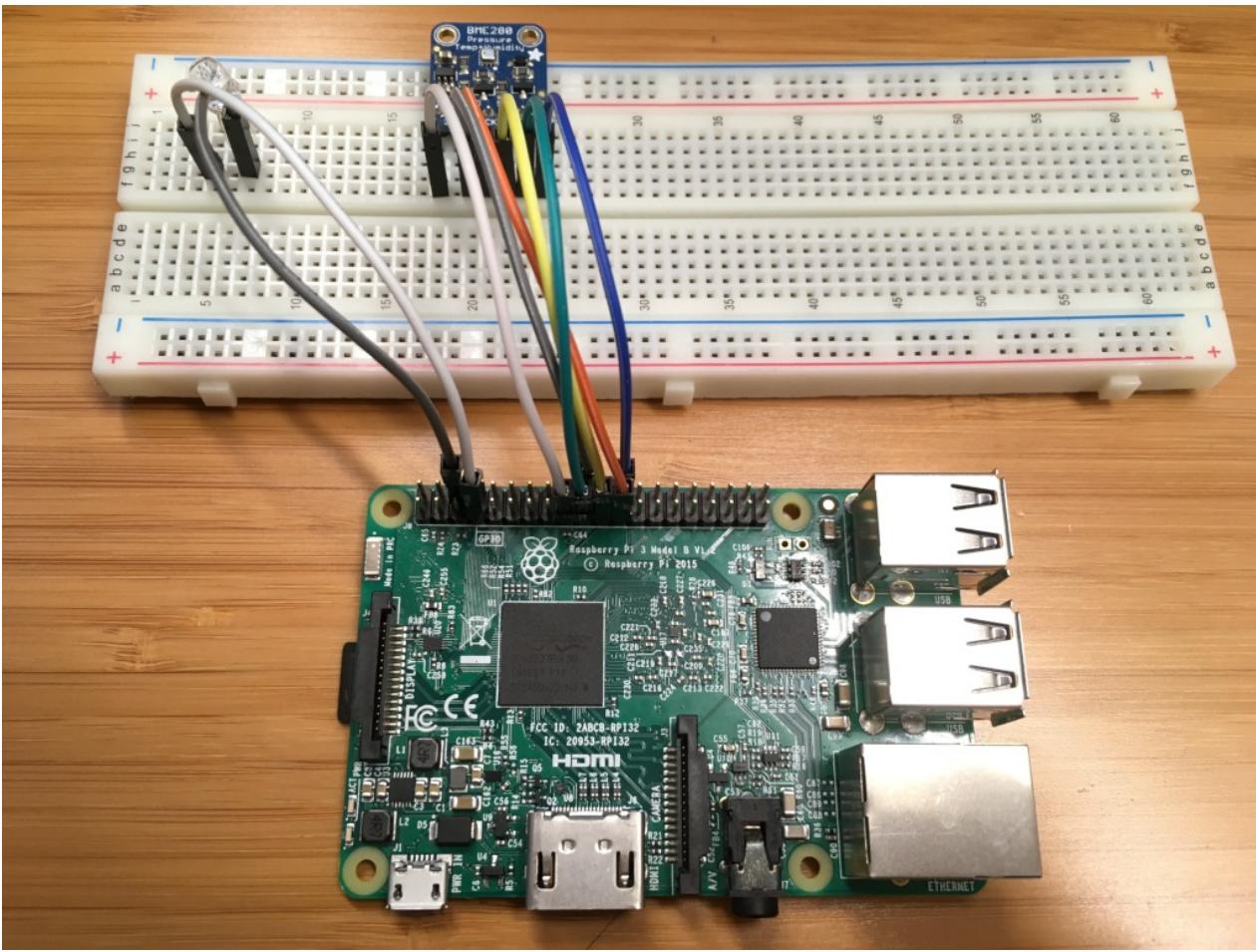
The BME280 sensor can collect temperature and humidity data. And the LED will blink if there is a communication between device and the cloud.

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 |

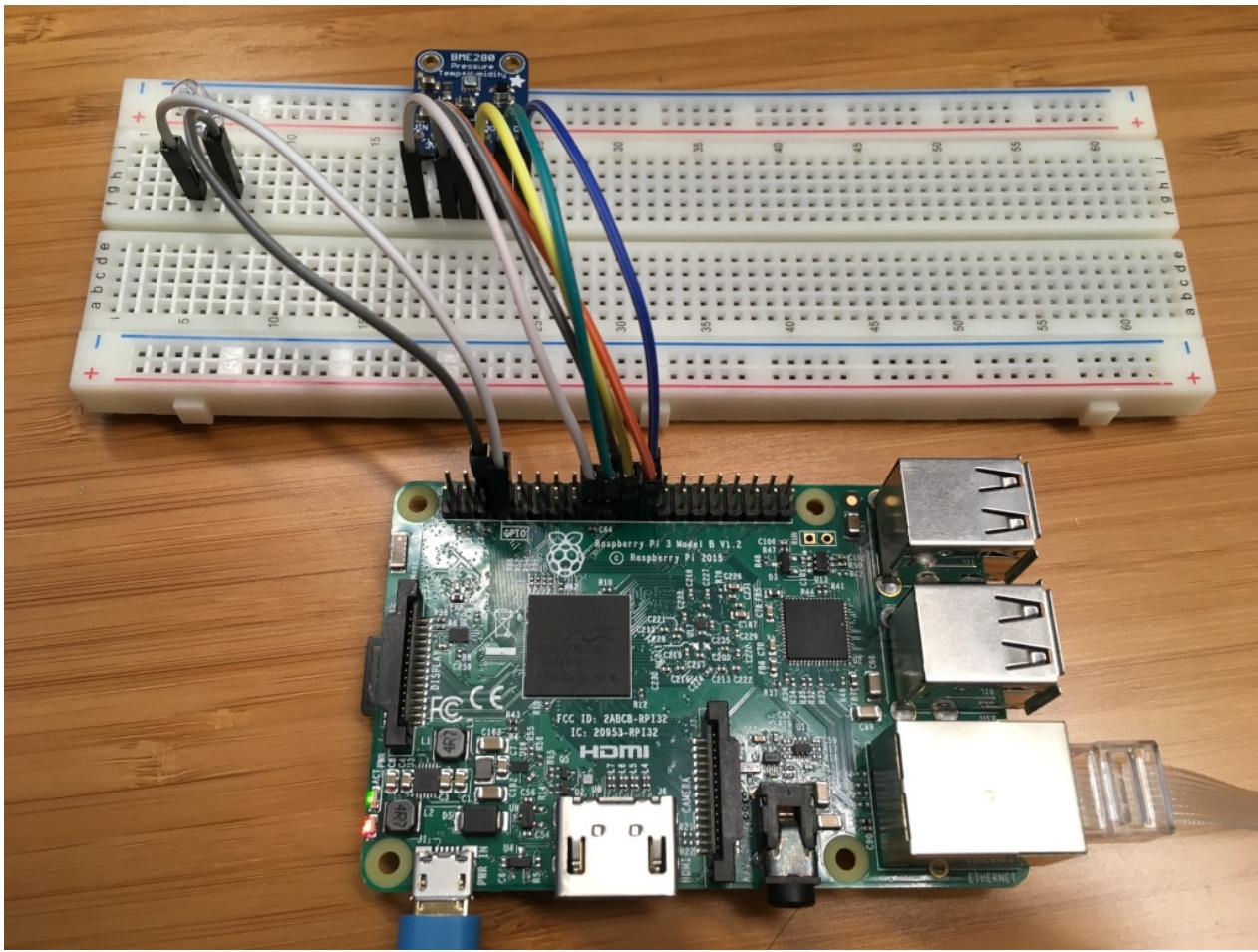
Click to view [Raspberry Pi 2 & 3 Pin mappings](#) for your reference.

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](#).



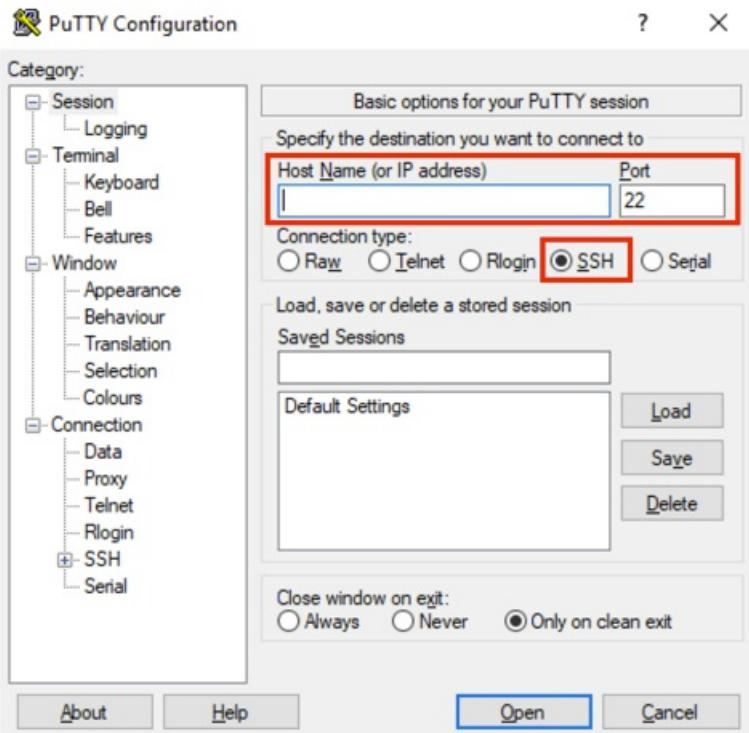
Run a sample application on Pi

Sign into 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-app

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 $
```

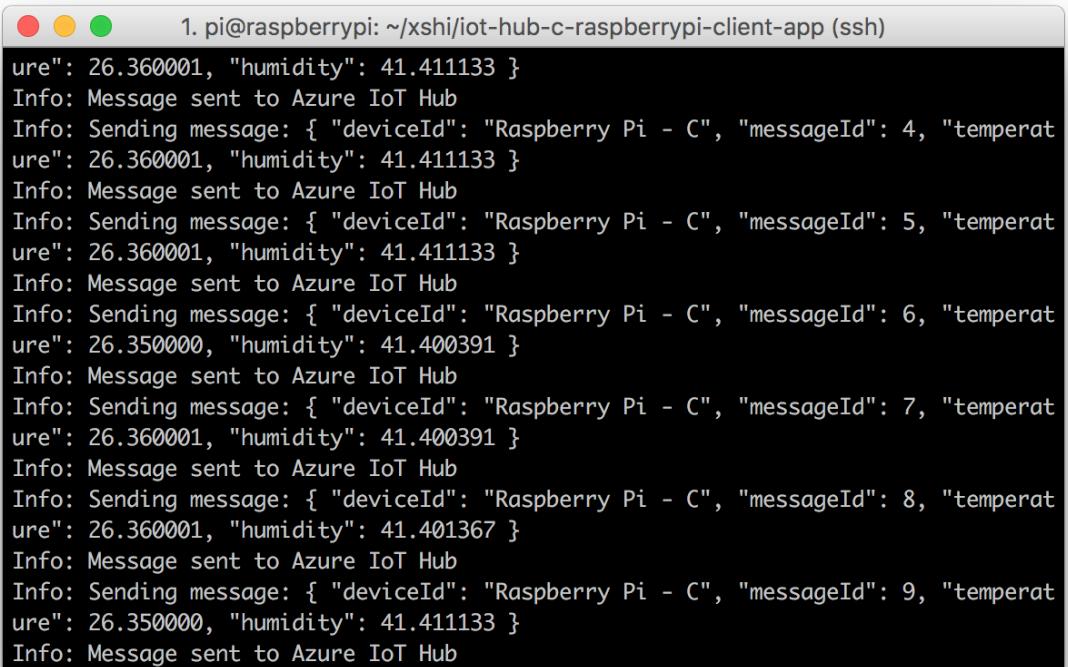
- Run the sample application by running the following command:

```
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.



A terminal window titled "1. pi@raspberrypi: ~xshi/iot-hub-c-raspberrypi-client-app (ssh)" displays a series of log messages. The messages show the device sending messages to the Azure IoT Hub at regular intervals. Each message contains a device ID ("Raspberry Pi - C"), a message ID (ranging from 4 to 9), a temperature reading (26.360001 or 26.350000), and a humidity reading (41.411133 or 41.400391). The log entries are as follows:

```
ure": 26.360001, "humidity": 41.411133 }
Info: Message sent to Azure IoT Hub
Info: Sending message: { "deviceId": "Raspberry Pi - C", "messageId": 4, "temperature": 26.360001, "humidity": 41.411133 }
Info: Message sent to Azure IoT Hub
Info: Sending message: { "deviceId": "Raspberry Pi - C", "messageId": 5, "temperature": 26.360001, "humidity": 41.411133 }
Info: Message sent to Azure IoT Hub
Info: Sending message: { "deviceId": "Raspberry Pi - C", "messageId": 6, "temperature": 26.350000, "humidity": 41.400391 }
Info: Message sent to Azure IoT Hub
Info: Sending message: { "deviceId": "Raspberry Pi - C", "messageId": 7, "temperature": 26.360001, "humidity": 41.400391 }
Info: Message sent to Azure IoT Hub
Info: Sending message: { "deviceId": "Raspberry Pi - C", "messageId": 8, "temperature": 26.360001, "humidity": 41.401367 }
Info: Message sent to Azure IoT Hub
Info: Sending message: { "deviceId": "Raspberry Pi - C", "messageId": 9, "temperature": 26.350000, "humidity": 41.411133 }
Info: Message sent to Azure IoT Hub
```

Read the messages received by your hub

One way to monitor messages received by your IoT hub from your device is to use the Azure IoT Tools for Visual Studio Code. To learn more, see [Use Azure IoT Tools for Visual Studio Code to send and receive messages between your device and IoT Hub](#).

For more ways to process data sent by your device, continue on to the next section.

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 extension for Visual Studio Code](#)
- [Manage devices with Azure IoT Hub extension for Visual Studio Code](#)
- [Set up message routing](#)
- [Use Power BI to visualize real-time sensor data from your IoT hub](#)
- [Use a web app 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

7/29/2020 • 14 minutes to read • [Edit Online](#)

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 rich [sample gallery](#).

What you learn

- How to create an IoT hub and register a device for the MXChip IoT DevKit.
- How to connect the IoT DevKit to Wi-Fi and configure the IoT Hub connection string.
- How to send the DevKit sensor telemetry data to your IoT hub.
- How to prepare the development environment and develop application for the IoT DevKit.

Don't have a DevKit yet? Try the [DevKit simulator](#) or [purchase a DevKit](#).

You can find the source code for all DevKit tutorials from [code samples gallery](#).

What you need

- A MXChip IoT DevKit board with a Micro-USB cable. [Get it now](#).
- A computer running Windows 10, macOS 10.10+ or Ubuntu 18.04+.
- An active Azure subscription. [Activate a free 30-day trial Microsoft Azure account](#).

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|--|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal . |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.

3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.

4. Select **Enter** to run the code.

Prepare your hardware

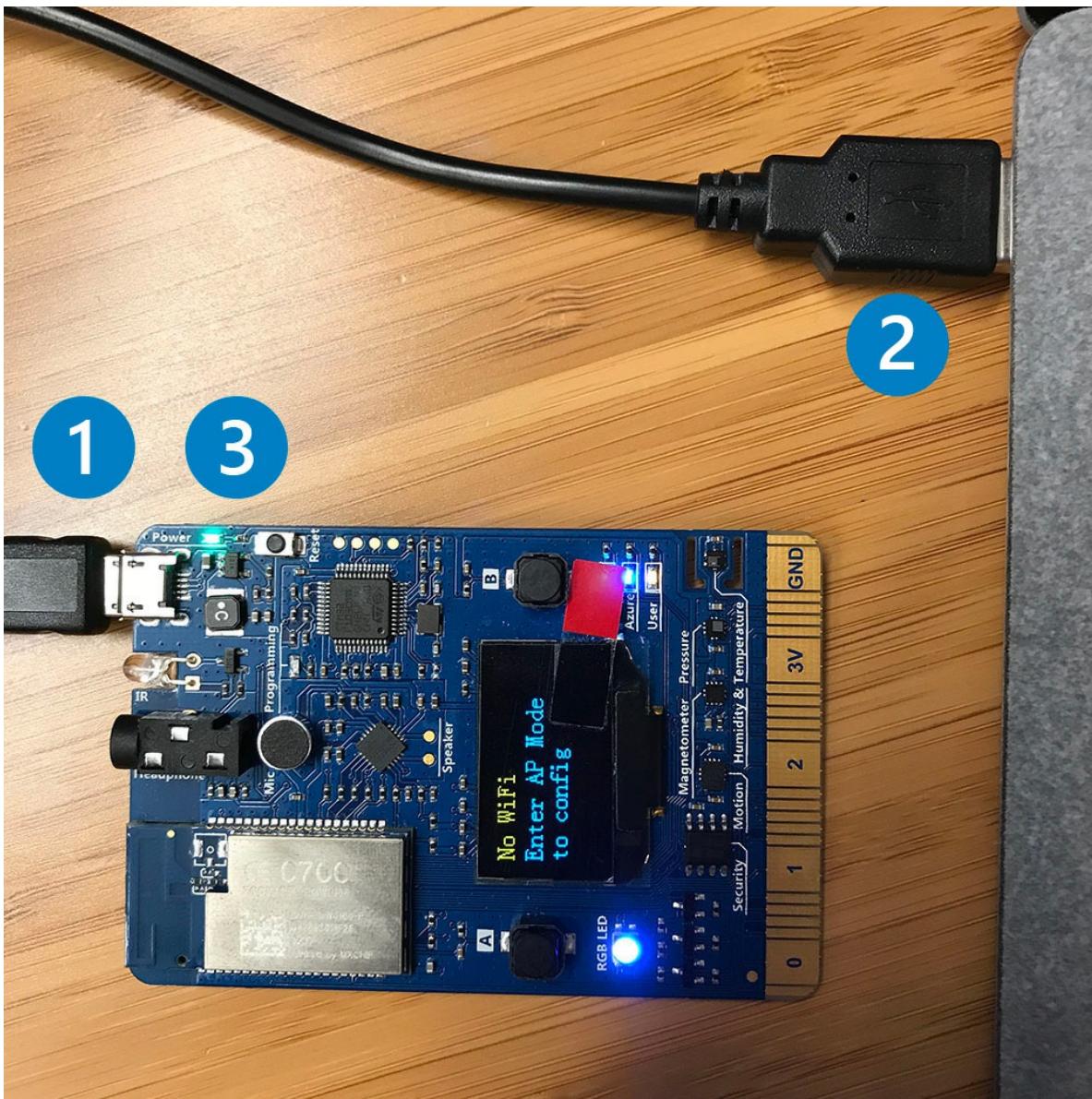
Hook up the following hardware to your computer:

- 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.



Quickstart: Send telemetry from DevKit to an IoT Hub

The quickstart uses pre-compiled DevKit firmware to send the telemetry to the IoT Hub. Before you run it, you create an IoT hub and register a device with the hub.

Create an IoT hub

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

1. Sign in to the [Azure portal](#).
2. From the Azure homepage, select the **+ Create a resource** button, and then enter *IoT Hub* in the **Search the Marketplace** field.
3. Select **IoT Hub** from the search results, and then select **Create**.
4. On the **Basics** tab, complete the fields as follows:
 - **Subscription:** Select the subscription to use for your hub.
 - **Resource Group:** Select a resource group or create a new one. To create a new one, select **Create new** and fill in the name you want to use. To use an existing resource group, select that resource group. For more information, see [Manage Azure Resource Manager resource groups](#).
 - **Region:** Select the region in which you want your hub to be located. Select the location closest to you. Some features, such as [IoT Hub device streams](#), are only available in specific regions. For these

limited features, you must select one of the supported regions.

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

IMPORTANT

Because the IoT hub will be publicly discoverable as a DNS endpoint, be sure to avoid entering any sensitive or personally identifiable information when you name it.

The screenshot shows the 'Basics' step of the IoT hub creation wizard. It includes fields for Subscription, Resource group, Region, and IoT hub name, all highlighted with a red box. Below the form are navigation buttons: 'Review + create' (blue), '< Previous' (disabled), 'Next: Size and scale >' (highlighted with a red box), and 'Automation options'.

Home > New > IoT hub

IoT hub
Microsoft

Basics [Size and scale](#) [Tags](#) [Review + create](#)

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

Project details

Choose the subscription you'll use to manage deployments and costs. Use resource groups like folders to help you organize and manage resources.

Subscription * ⓘ

Resource group * ⓘ [Create new](#)

Region * ⓘ

IoT hub name * ⓘ

[Review + create](#) [< Previous](#) [Next: Size and scale >](#) [Automation options](#)

5. Select **Next: Size and scale** to continue creating your hub.

Home > New > IoT hub

IoT hub

Microsoft

Basics Size and scale Tags 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 [Determines how your IoT hub can scale. You can change this later if your needs increase.](#)

Azure Security Center [On](#) Turn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

| | | | |
|--------------------------|--------------------------------|----------------------------|---------|
| Pricing and scale tier ⓘ | S1 | Device-to-cloud-messages ⓘ | Enabled |
| Messages per day ⓘ | 400,000 | Message routing ⓘ | Enabled |
| Cost per month | 25.00 USD | Cloud-to-device commands ⓘ | Enabled |
| Azure Security Center ⓘ | 0.001 USD per device per month | IoT Edge ⓘ | Enabled |
| | | Device management ⓘ | Enabled |

Advanced settings

[Review + create](#) [< Previous: Basics](#) [Next: Tags >](#) [Automation options](#)

You can accept the default settings here. If desired, you can modify any of the following fields:

- **Pricing and scale tier:** Your selected 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 hub and up to 8,000 messages per day. Each Azure subscription can create one IoT hub in the free tier.
- If you are working through a Quickstart for IoT Hub device streams, select 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 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](#).
- **Azure Security Center:** Turn this on to add an extra layer of threat protection to IoT and your devices. This option is not available for hubs in the free tier. For more information about this feature, see [Azure Security Center for IoT](#).
- **Advanced Settings > Device-to-cloud partitions:** This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most hubs need only four partitions.

6. Select **Next: Tags** to continue to the next screen.

Tags are name/value pairs. You can assign the same tag to multiple resources and resource groups to categorize resources and consolidate billing. For more information, see [Use tags to organize your Azure](#)

resources.

The screenshot shows the 'Tags' section of the IoT hub creation wizard. It displays two tag entries:

| Name | Value | Resource |
|------------|------------|----------|
| department | accounting | IoT Hub |
| | | IoT Hub |

Below the table are navigation buttons: 'Review + create' (highlighted in blue), '< Previous: Size and scale', 'Next: Review + create >', and 'Automation options'.

7. Select **Next: Review + create** to review your choices. You see something similar to this screen, but with the values you selected when creating the hub.

This screenshot shows the final review step before creating the IoT hub. The 'Review + create' tab is highlighted with a red box. The page displays the following information:

| Basics | Size and scale | Tags |
|---|----------------|------|
| Subscription : Personal testing | | |
| Resource group : iot-hubs | | |
| Region : West US 2 | | |
| IoT hub name : you-hub-name | | |
| 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 | | |
| Azure Security Center : 0.001 USD per device per month | | |
| Tags | | |
| department : accounting | | |

At the bottom, the 'Create' button is highlighted with a red box, along with other buttons: '< Previous: Tags', 'Next >', and 'Automation options'.

8. Select **Create** to create your new 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 command in Azure Cloud Shell 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 iot hub device-identity create --hub-name YourIoTHubName --device-id MyNodeDevice
```

NOTE

If you get an error running `device-identity`, install the [Azure IoT Extension for Azure CLI](#). Run the following command to add the Microsoft Azure IoT Extension for Azure CLI to your Cloud Shell instance. The IoT Extension adds commands that are specific to IoT Hub, IoT Edge, and IoT Device Provisioning Service (DPS) to Azure CLI.

```
az extension add --name azure-iot
```

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:

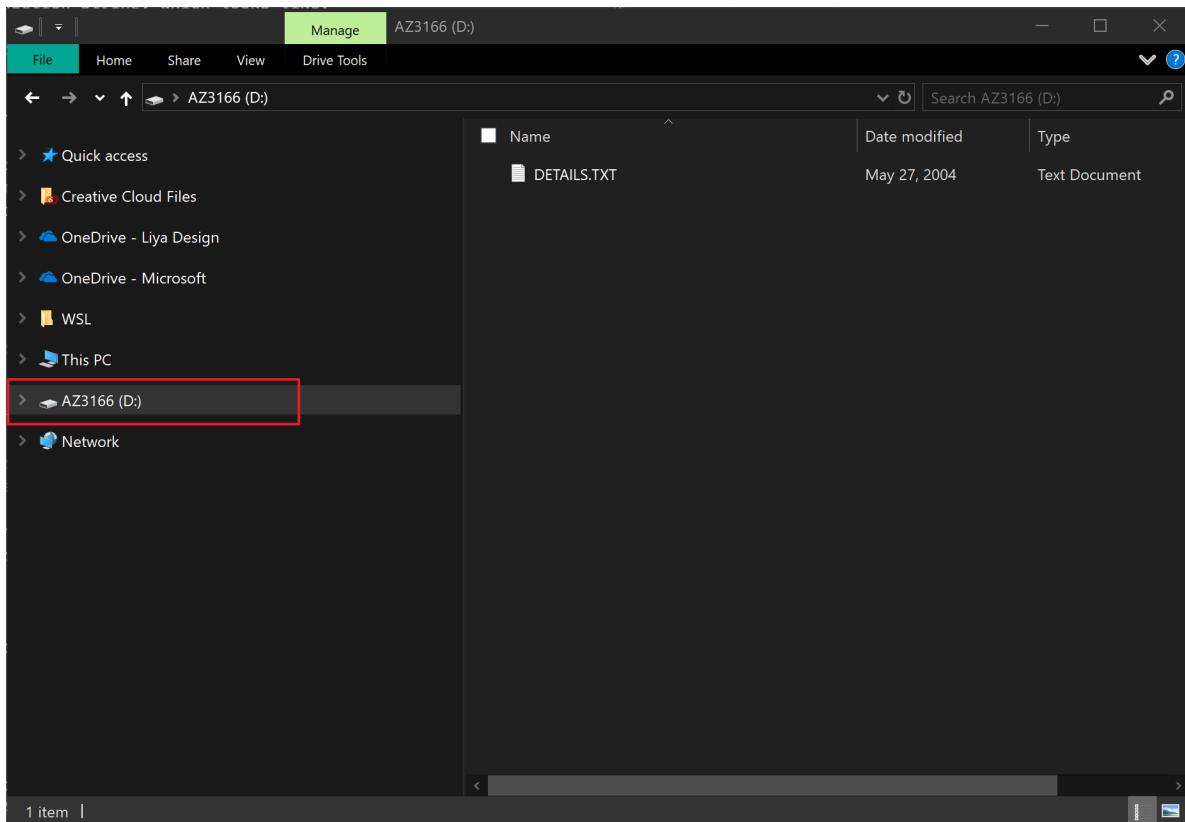
```
HostName={YourIoTHubName}.azure-devices.net;DeviceId=MyNodeDevice;SharedAccessKey={YourSharedAccessKey}
```

You use this value later in the quickstart.

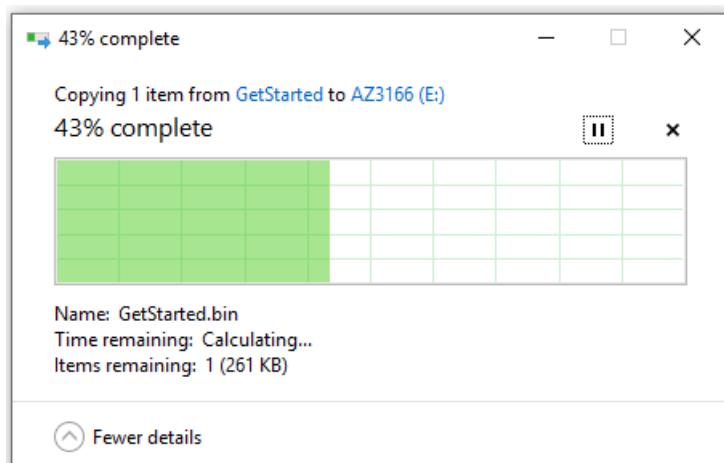
Send DevKit telemetry

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

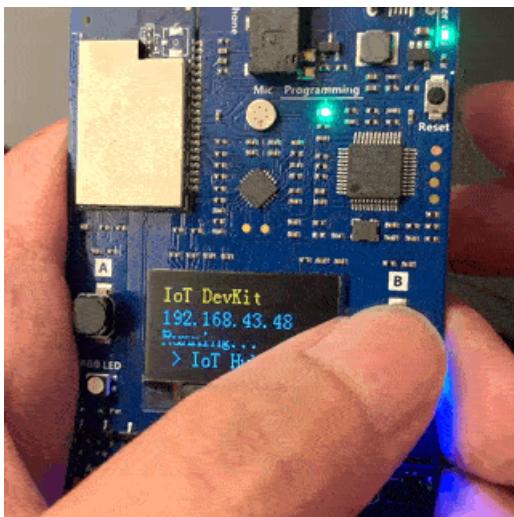
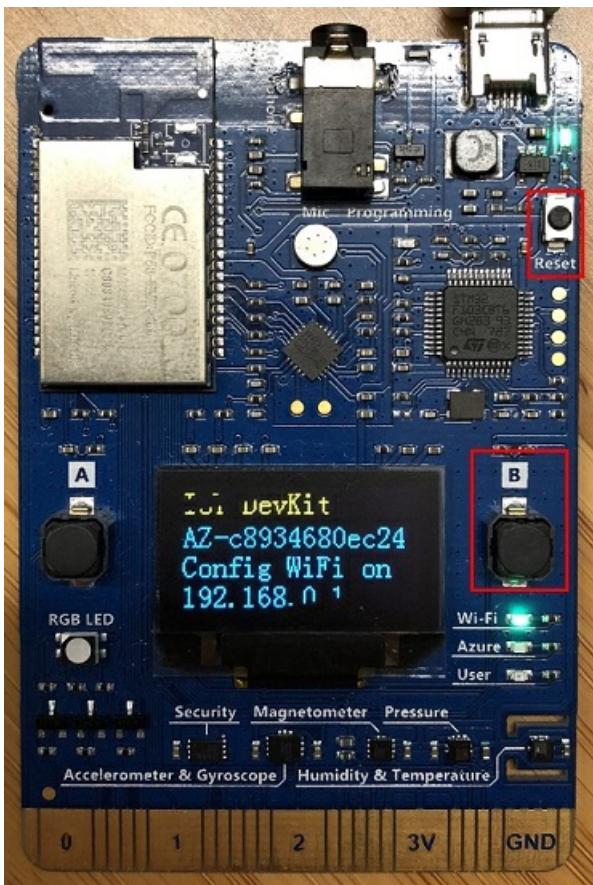
1. Download the latest version of [GetStarted firmware](#) for IoT DevKit.
2. Make sure IoT DevKit connect to your computer via USB. Open File Explorer there is a USB mass storage device called **AZ3166**.



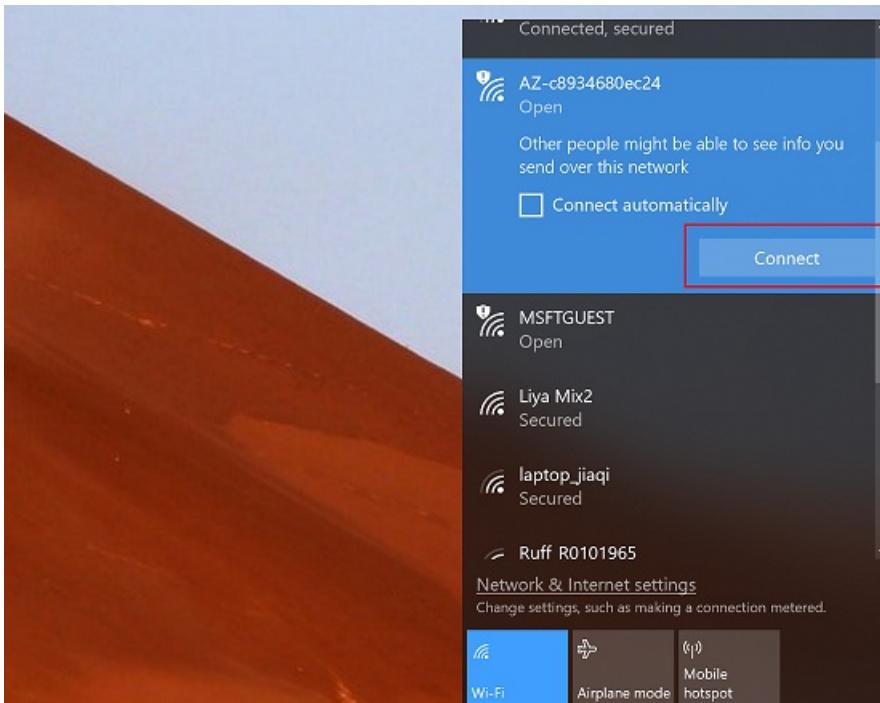
3. Drag and drop the firmware just downloaded into the mass storage device and it will flash automatically.



4. On the DevKit, Hold down button **B**, push and release the **Reset** button, and then release button **B**. Your DevKit enters AP mode. To confirm, the screen displays the service set identifier (SSID) of the DevKit and the configuration portal IP address.



5. Use a Web browser on a different Wi-Fi enabled device (computer or mobile phone) to connect to the IoT DevKit SSID displayed in the previous step. If it asks for a password, leave it empty.



6. Open **192.168.0.1** in the browser. Select the Wi-Fi that you want the IoT DevKit connect to, type the Wi-Fi password, then paste the device connection string you made note of previously. Then click Save.

The screenshot shows a web-based configuration interface for the IoT DevKit. The title bar says "IoT DevKit Settings". Below it is a form with fields for SSID (radio buttons for "List" and "SSID"), Password, and IoT Device Connection String. A large blue "Save" button is at the bottom. A note at the bottom of the page says: "Please refresh this page to update SSID if you cannot find it from the list".

NOTE

The IoT DevKit only supports 2.4GHz network. Check [FAQ](#) for more details.

7. The WiFi information and device connection string will be stored into the IoT DevKit when you see the result page.

← → ⌂ ⌂ 192.168.0.1/result ⌂ ⌂ ⌂

IoT DevKit Settings

| Settings |
|--------------------------------------|
| Wi-Fi SSID and Password - saved |
| IoT Device Connection String - saved |

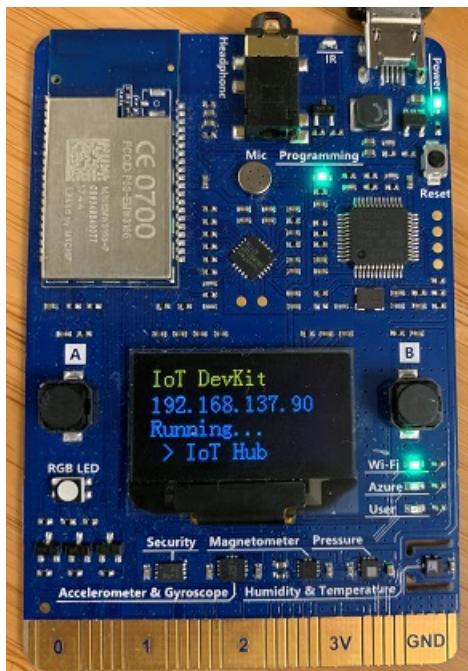
Wi-Fi is connected:

Wait a few seconds for the IoT DevKit to reboot...

NOTE

After Wi-Fi is configured, your credentials will persist on the device for that connection, even if the device is unplugged.

8. The IoT DevKit reboots in a few seconds. On the DevKit screen, you see the IP address for the DevKit follows by the telemetry data including temperature and humidity value with message count send to Azure IoT Hub.





9. To verify the telemetry data sent to Azure, run the following command in Azure Cloud Shell:

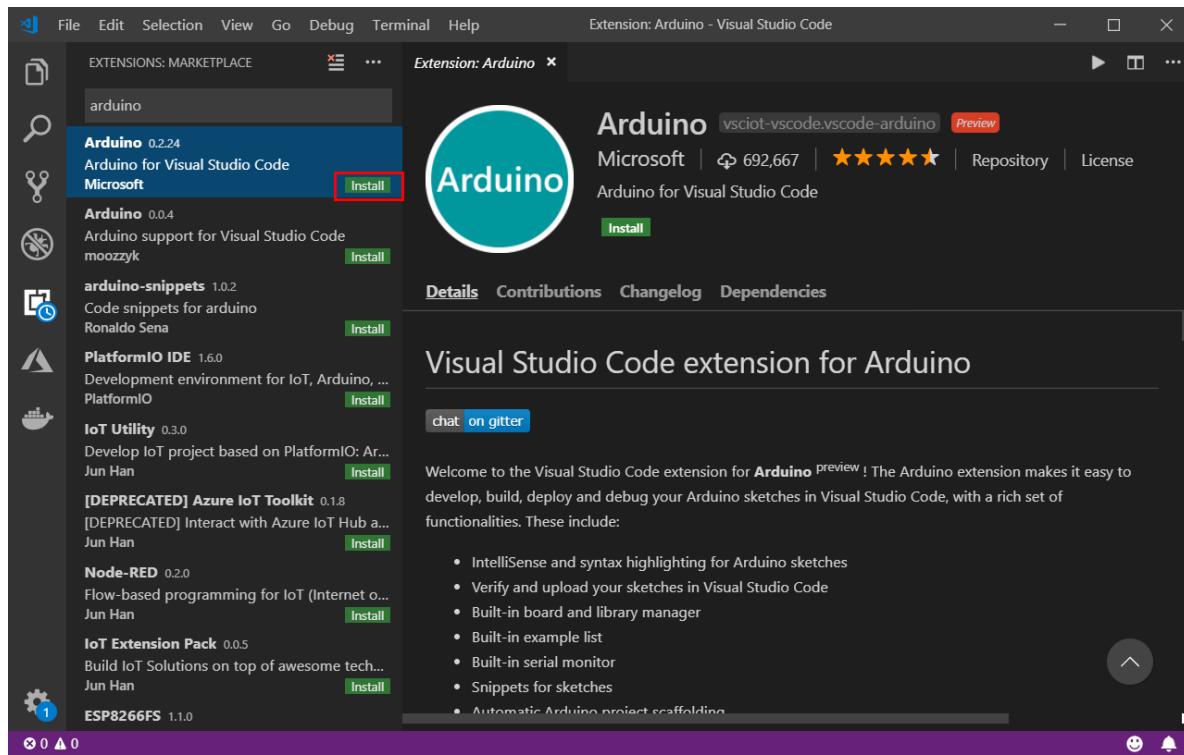
```
az iot hub monitor-events --hub-name YourIoTHubName --output table
```

Prepare the development environment

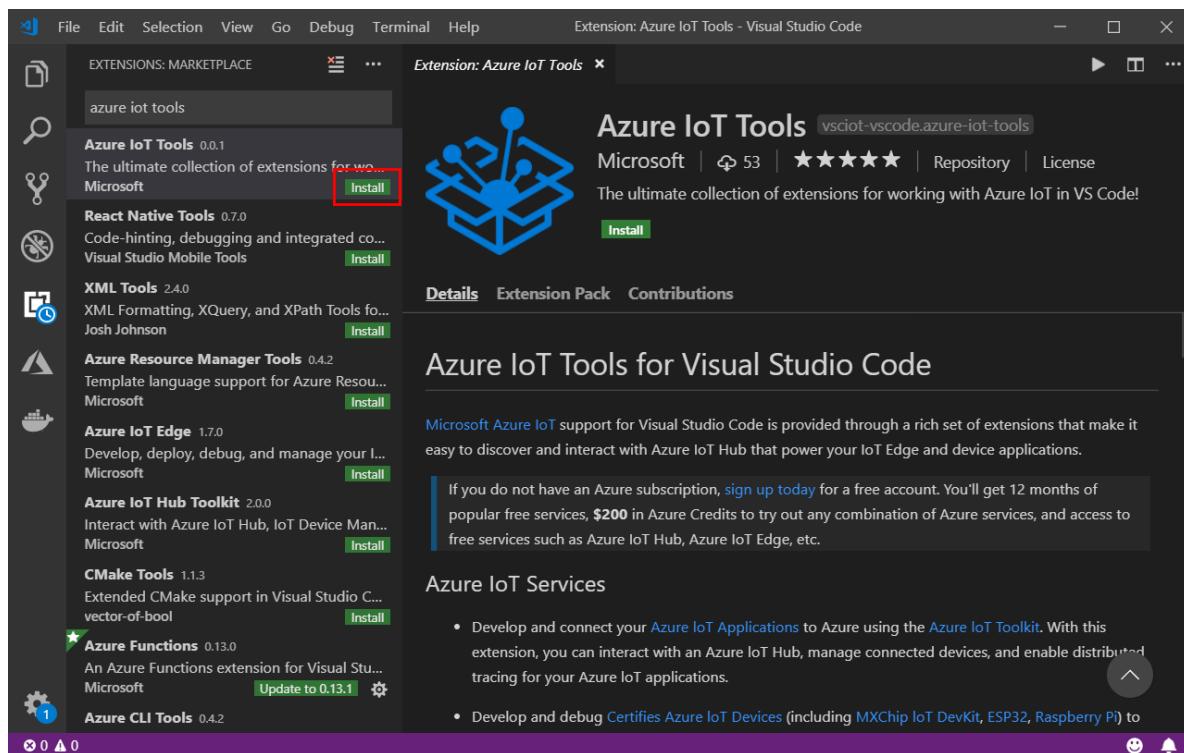
Follow these steps to prepare the development environment for the DevKit:

Install Visual Studio Code with Azure IoT Tools extension package

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 intellisense, code completion and debugging support as well as rich extensions can be installed from marketplace.
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.



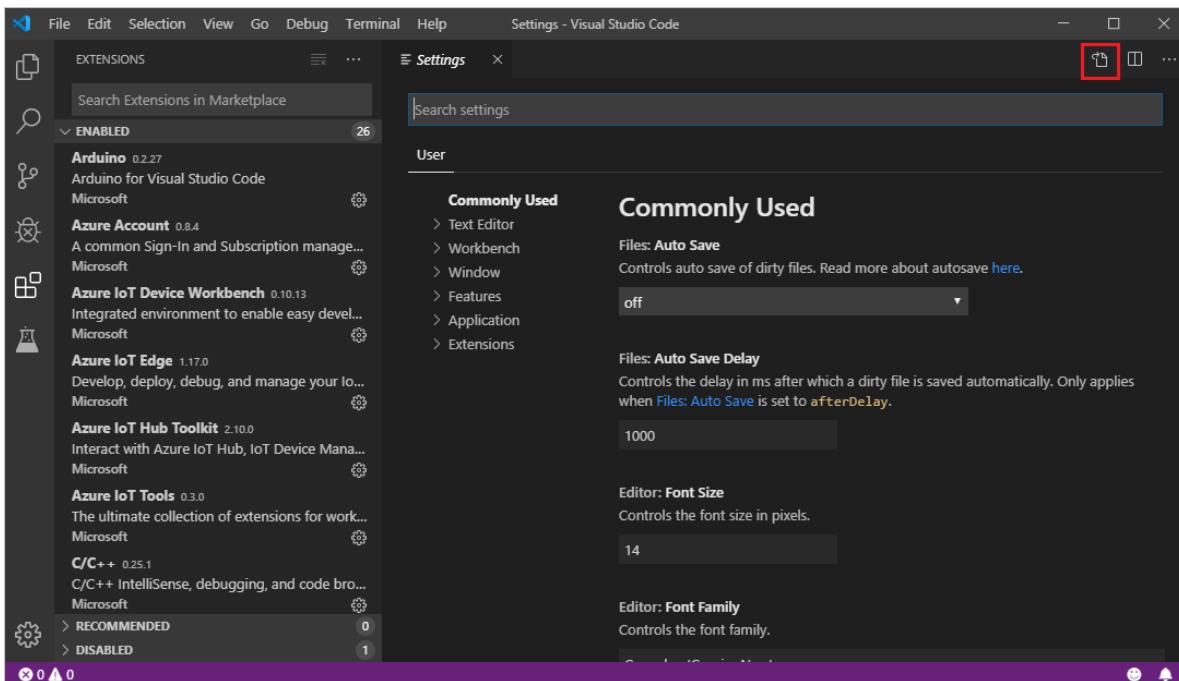
Or copy and paste this URL into a browser window: <vscode:extension/vsciot-vscode.azure-iot-tools>

NOTE

The Azure IoT Tools extension pack contains the [Azure IoT Device Workbench](#) which is used to develop and debug on various IoT devkit devices. The [Azure IoT Hub extension](#), also included with the Azure IoT Tools extension pack, is used to manage and interact with Azure IoT Hubs.

5. Configure VS Code with Arduino settings.

In Visual Studio Code, click **File > Preferences > Settings** (on macOS, **Code > Preferences > Settings**). Then click the **Open Settings (JSON)** icon in the upper-right corner of the *Settings* page.



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:

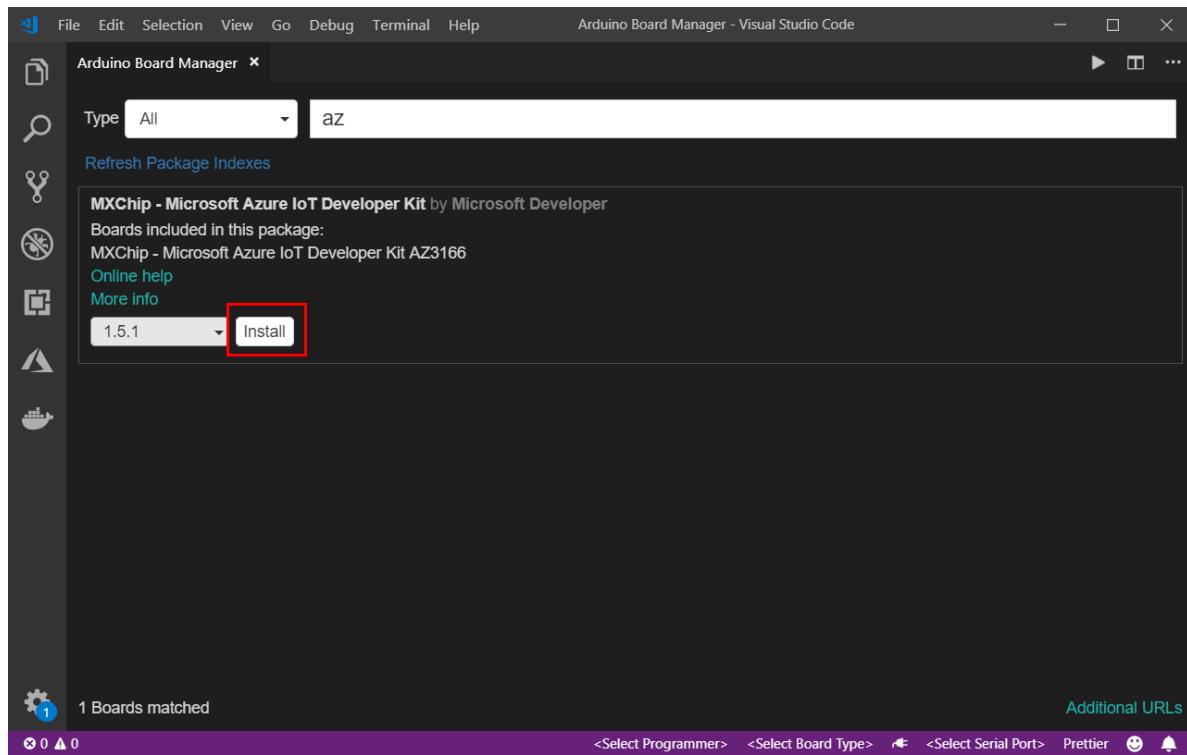
```
"arduino.path": "/Applications",
"arduino.additionalUrls":
"https://raw.githubusercontent.com/VSChina/azureiotdevkit_tools/master/package_azureboard_index.json"
```

- 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 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 commands in terminal and sign out and sign 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 GetStarted sample you just ran.

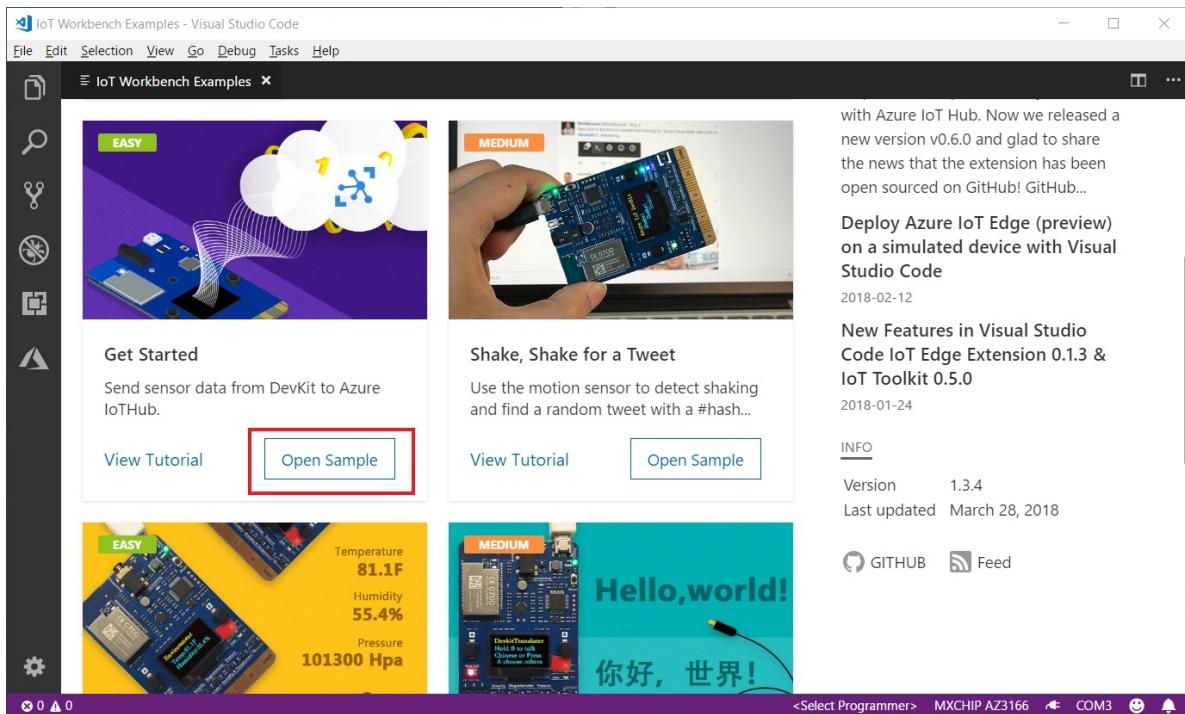
Build your first project

Open sample code from sample gallery

The IoT DevKit contains a rich gallery of samples that you can use to learn connect the DevKit to various Azure services.

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

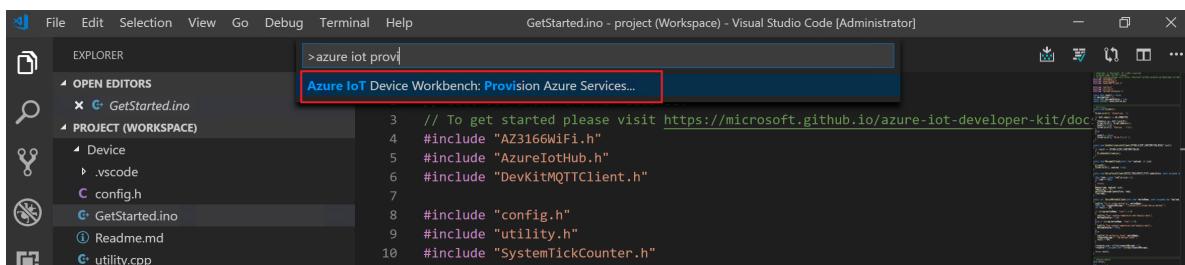
path to download the sample code.



Provision Azure IoT Hub and device

Instead of provisioning Azure IoT Hub and device from the Azure portal, you can do it in the VS Code without leaving the development environment.

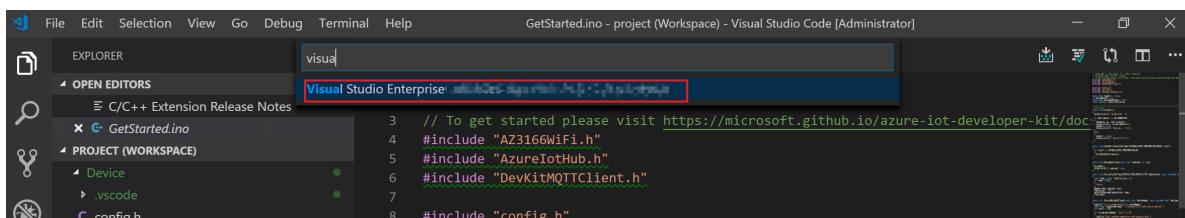
1. In the new opened project window, click **F1** to open the command palette, type and select **Azure IoT Device Workbench: Provision Azure Services....** Follow the step by step guide to finish provisioning your Azure IoT Hub and creating the IoT Hub device.



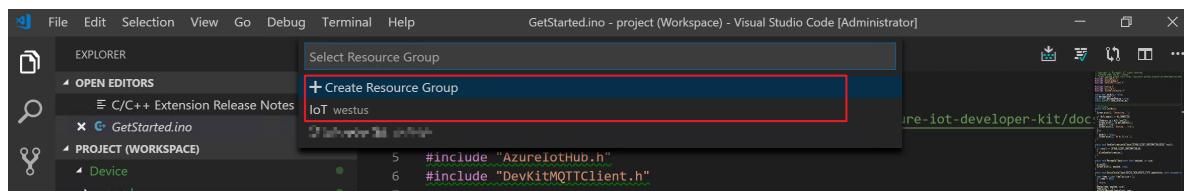
NOTE

If you have not signed in Azure. Follow the pop-up notification for signing in.

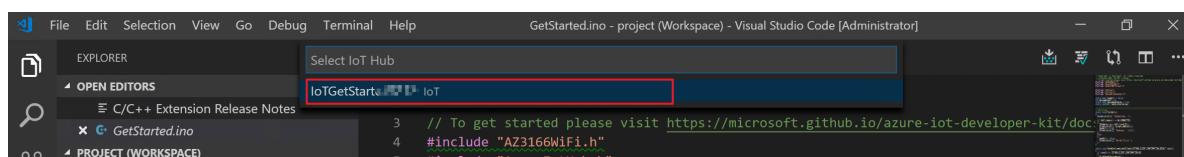
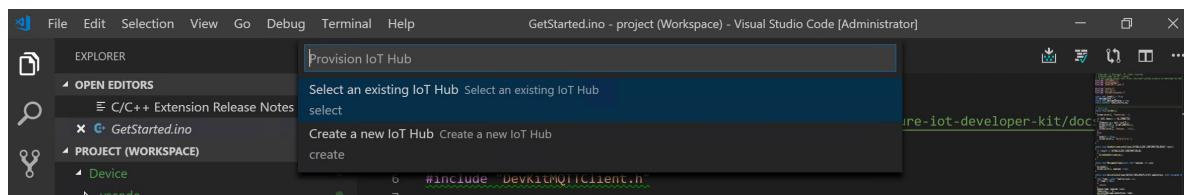
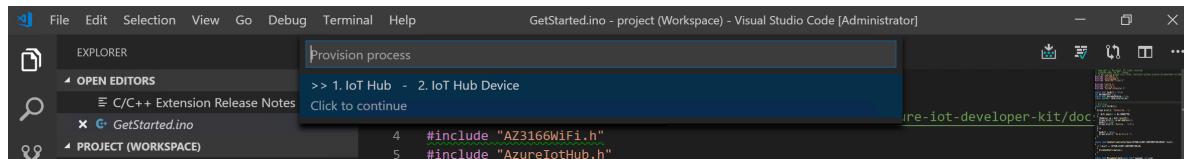
2. Select the subscription you want to use.



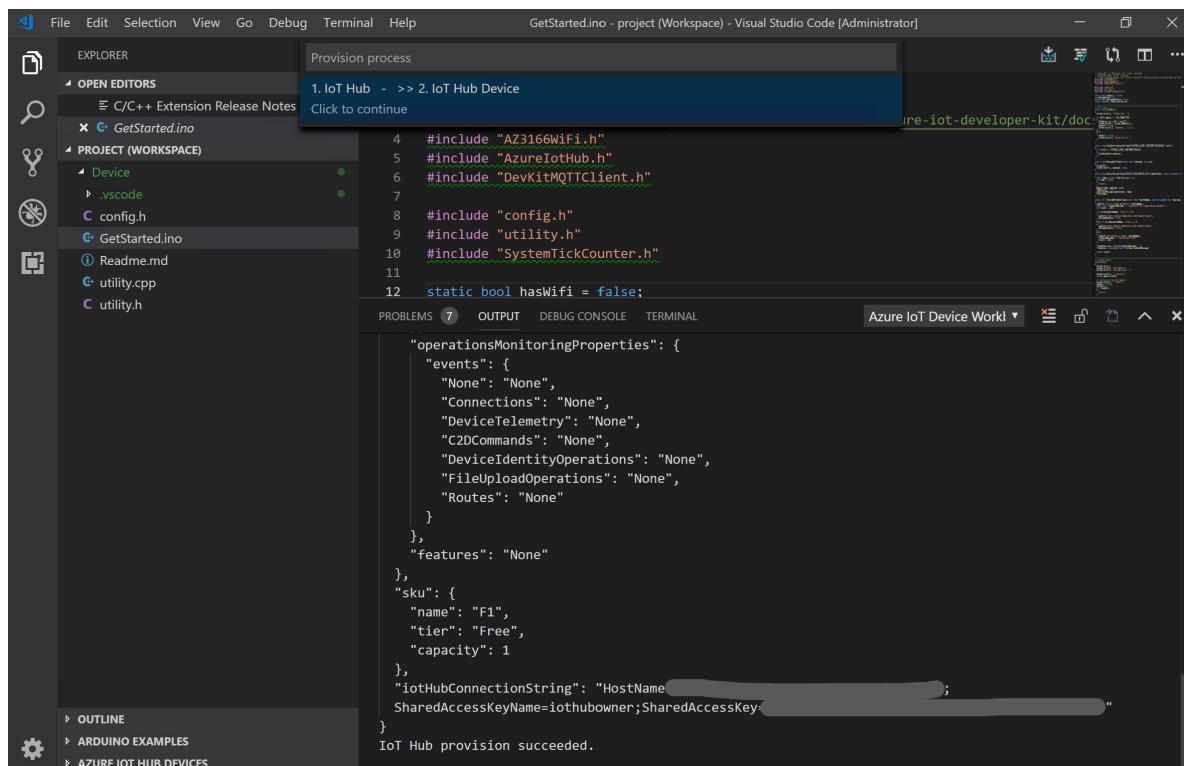
3. Then select or create a new **resource group**.



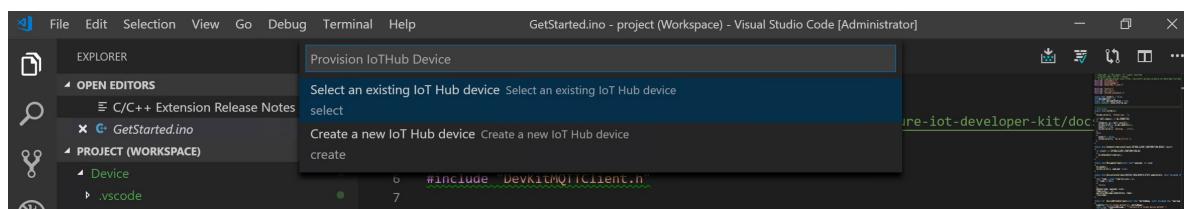
4. In the resource group you specified, follow the guide to select or create a new Azure IoT Hub.

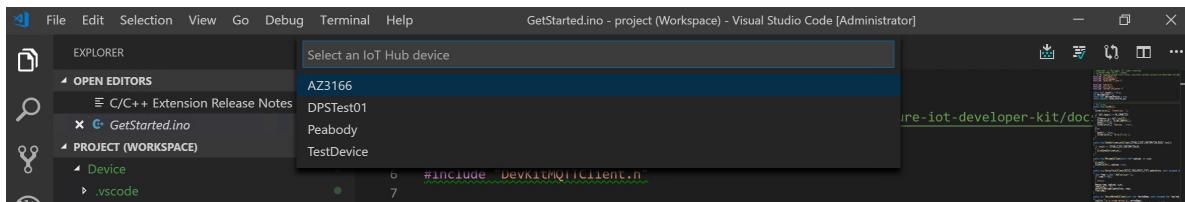


5. In the output window, you will see the Azure IoT Hub provisioned.

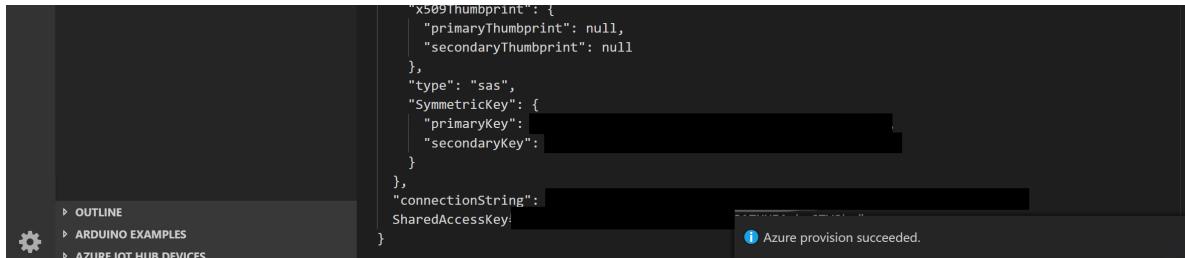


6. Select or create a new device in Azure IoT Hub you provisioned.



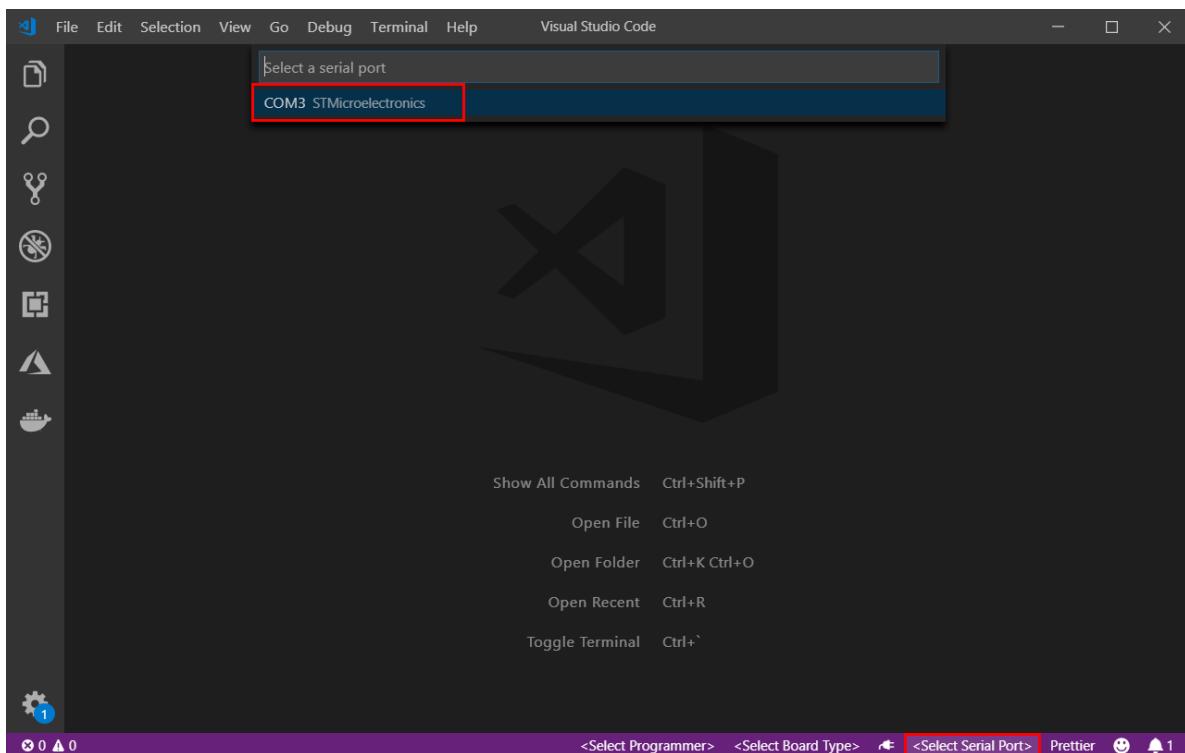


7. Now you have Azure IoT Hub provisioned and device created in it. Also the device connection string will be saved in VS Code for configuring the IoT DevKit later.

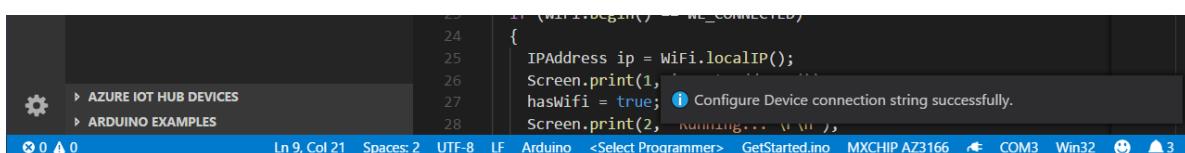


Configure and compile device code

1. In the bottom-right status bar, check the **MXCHIP AZ3166** is shown as selected board and serial port with **STMicroelectronics** is used.



2. 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**.
3. 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.



4. Click **F1** again, type and select **Azure IoT Device Workbench: Upload Device Code**. It starts compile and upload the code to DevKit.

The screenshot shows the Visual Studio Code interface with the Arduino extension installed. The Explorer sidebar displays the project structure. The main editor window shows the `GetStarted.ino` file content, which includes includes for WiFi, Azure IoT Hub, and DevKit MQTT Client, along with configuration and utility headers. The bottom status bar indicates a connection to 'Azure: liydu@microsoft.com' and shows a power icon with a '1'.

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 screenshot shows the Serial Monitor window in Visual Studio Code. It displays the message `20 with result = IOTHUB_CLIENT_CONFIRMATION_OK` and `[Done] Closed the serial port`. The status bar shows a power icon with a '1'.

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.

The screenshot shows the Visual Studio Code interface with the following details:

- File Bar:** File, Edit, Selection, View, Go, Debug, Terminal, Help.
- Explorer Sidebar:** OPEN EDITORS (GetStarted.ino), PROJECT (WORKSPACE) (Device, .vscode, config.h, GetStarted.ino, Readme.md, utility.cpp, utility.h).
- Code Editor:** The file "GetStarted.ino" is open, showing C++ code related to Azure IoT Hub and MQTT Client.
- Terminal:** Serial Monitor tab is active, showing logs from the IoT Hub client:

```
[Starting] Opening the serial port - COM3
[Info] Opened the serial port - COM3
2018-12-19 03:55:42 INFO: >>>IoTHubClient_LL_SendEventAsync accepted message for transmission to IoT Hub.
2018-12-19 03:55:42 INFO: >>>Confirmation[5] received for message tracking id = 5 with result = IOTHUB_CLIENT_CONFIRMATION_OK
2018-12-19 03:55:45 INFO: >>>IoTHubClient_LL_SendEventAsync accepted message for transmission to IoT Hub.
2018-12-19 03:55:45 INFO: >>>Confirmation[6] received for message tracking id = 6 with result = IOTHUB_CLIENT_CONFIRMATION_OK
```
- Bottom Status Bar:** Ln 20, Col 2, Spaces: 2, UTF-8, LF, C++, <Select Programmer>, GetStarted.ino, No line ending, MXCHIP AZ3166, 115200, COM3, 2 notifications.

NOTE

You might encounter an error during testing in which the LED isn't blinking, the Azure portal doesn't show incoming data from the device, but the device OLED screen shows as **Running....** To resolve the issue, in the Azure portal, go to the device in the IoT hub and send a message to the device. If you see the following response in the serial monitor in VS Code, it's possible that direct communication from the device is blocked at the router level. Check firewall and router rules that are configured for the connecting devices. Also, ensure that outbound port 1833 is open.

```
ERROR: mqtt_client.c (ln 454): Error: failure opening connection to endpoint
INFO: >>>Connection status: disconnected
ERROR: tlsio_mbedtls.c (ln 604): Underlying IO open failed
ERROR: mqtt_client.c (ln 1042): Error: io_open failed
ERROR: iothubtransport_mqtt_common.c (ln 2283): failure connecting to address atcslioithub.azure-devices.net.
INFO: >>>Re-connect.
INFO: IoThub Version: 1.3.6
```

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.

Some IoT Hub features are disabled due to the location of your resource. The following are the impacted features: Manual failover (preview)

Resource group (change) IoT

Status Active

Location West US

Subscription (change) Visual Studio Enterprise

Subscription ID

Hostname IoTGetStarted8735.azure-devices.net

Pricing and scale tier F1 - Free

Number of IoT Hub units 1

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 | service connect |
| device | device connect |
| registryRead | registry read |
| registryReadWrite | registry write |

Access policy name: iothubowner

Permissions:

- Registry read
- Registry write
- Service connect
- Device connect

Shared access keys:

Primary key: T3Y7KC/Wv+PoRq+18WsyA1a5IJf2XYI2... (Copy)

Secondary key: UTQzef7/zox6ltP+HoWtishDxpTseOB... (Copy)

Connection string—primary key:HostName=IoTGetStarted8735.azure-d... (Copy)

Connection string—secondary key:HostName=IoTGetStarted8735.azure-d... (Copy)

3. In VS Code, click **F1**, type and select **Azure IoT Hub: Set IoT Hub Connection String**. Copy the connection string into it.

```

File Edit Selection View Go Debug Terminal Help
GetStarted.ino - project (Workspace) - Visual Studio Code
EXPLORER >iot hub: set iot hub con
OPEN EDITORS GetStarted.ino
PROJECT (WORKSPACE)
Device .vscode
3 // To get started please visit https://microsoft.github.io/azure-iot-d...
4 #include "AZ3166WiFi.h"
5 #include "AzureIoTHub.h"
6 #include "DevKitMQTTClient.h"

```

4. Expand the AZURE IOT HUB DEVICES pane on the left, right click on the device name you created and select **Start Monitoring Built-in Event Endpoint**.

```

// Copyright (c) Microsoft. All rights reserved.
// Licensed under the MIT license.
// To get started please visit https://microsoft.github.io/azure-iot-device-sdk/arduino
#include "AZ3166WiFi.h"
#include "AzureIoTHub.h"
#include "DevKitMQTTClient.h"

#include "config.h"
#include "utility.h"
#include "SystemTickCounter.h"

bool hasWifi = false;
int messageCount = 1;
int ntMessageCount = 0;
bool messageSending = true;
uint64_t send_interval_ms;

float temperature;
float humidity;

void InitWifi()
{
    Screen.print(2, "Connecting ...");

    if (WiFi.begin() == WL_CONNECTED)
    {
        IPAddress ip = WiFi.localIP();
        Screen.print(1, ip.get_address());
        hasWifi = true;
        Screen.print(2, "Running ... \r\n");
    }
    else
    {
        Screen.print(2, "Failed");
    }
}

static void SendConfirmationOnSuccess(IOTHUB_CLIENT_C
{
    if (result == IOTHUB_CLIENT_CONNECTION_OK)
        blackboardConnection();
    sendMessageCount++;
}

Screen.print(1, "To IoT Hub");
sprayTimeCount = 500;
sendMessageCount = 0;
hasWifi = true;
sprayTimeCount = 1000;
hasWifi = true;
sprayTimeCount = 1500;
hasWifi = true;
messageCount++;

static void MessageCallback(const char payload, int
{
    blackboard();
    Screen.print(1, payload, true);
}

static void DeviceTwinCallback(DEVICE_TWIN_UPDATE_ST
{
    char *temp = (char *)malloc(sizeof(char));
    if (temp == NULL)
        return;
    memcpy(temp, payload, size);
    temp[0] = '\0';
    parseTwinMessage(updateState, temp);
    free(temp);
}

static int DeviceMethodCallback(const char methodN
{
    LogInfo("Try to invoke method %s", methodName);
    const char *methodName = "TemperatureAlert";
    int result = 200;
    if (strcmp(methodName, "start") == 0)
        LogInfo("Start sending temperature and humidity
    else if (strcmp(methodName, "stop") == 0)
        LogInfo("Stop sending temperature and humidity d
    messageSending = false;
    else
        LogInfo("No method to found", methodName);
    responseSize = strlen(responseMessage) + 5;
    responseMessage = (unsigned char *)malloc(responseMes
    return result;
}

// Arduino sketch

```

5. In OUTPUT pane, you can see the incoming D2C messages to the IoT Hub.

```

// Copyright (c) Microsoft. All rights reserved.
// Licensed under the MIT license.
// To get started please visit https://microsoft.github.io/azure-iot-device-sdk/arduino
#include "AZ3166WiFi.h"
#include "AzureIoTHub.h"
#include "DevKitMQTTClient.h"

#include "config.h"
#include "utility.h"
#include "SystemTickCounter.h"

bool hasWifi = false;

applicationProperties": {
    "temperatureAlert": "false"
}
}

[IoTHubMonitor] [12:06:55 PM] Message received from [AZ3166]:
{
    "body": {
        "messageId": 244,
        "humidity": 33.29999923706055
    },
    "applicationProperties": {
        "temperatureAlert": "false"
    }
}

```

Review the code

The `GetStarted.ino` is the main Arduino sketch file.

```
17 // Utilities
18 static void InitWifi()
19 {
20     Screen.print(2, "Connecting...");
21
22     if (WiFi.begin() == WL_CONNECTED)
23     {
24         IPAddress ip = WiFi.localIP();
25         Screen.print(1, ip.get_address());
26         hasWifi = true;
27         Screen.print(2, "Running... \r\n");
28     }
29     else
30     {
31         hasWifi = false;
32         Screen.print(1, "No Wi-Fi\r\n");
33     }
34 }
35
36 static void SendConfirmationCallback(IOTHUB_CLIENT_CONFIRMATION_RESULT result)
37 {
38     if (result == IOTHUB_CLIENT_CONFIRMATION_OK)
39     {
40         blinkSendConfirmation();
41     }
42 }
43
44 static void MessageCallback(const char* payload, int size)
45 {
46     blinkLED();
47     Screen.print(1, payload, true);
48 }
49
50 static void DeviceTwinCallback(DEVICE_TWIN_UPDATE_STATE updateState, const unsigned char*
51 {
52     char *temp = (char *)malloc(sizeof(char) * 100);
53 }
```

To see how device telemetry is sent to the Azure IoT Hub, open the `utility.cpp` file in the same folder. View [API Reference](#) to learn how to use sensors and peripherals on IoT DevKit.

The `DevKitMQTTClient` used is a wrapper of the `iothub_client` from the [Microsoft Azure IoT SDKs and libraries for C](#) to interact with Azure IoT Hub.

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 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](#)

Use IoT DevKit AZ3166 with Azure Functions and Cognitive Services to make a language translator

11/12/2019 • 3 minutes to read • [Edit Online](#)

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.

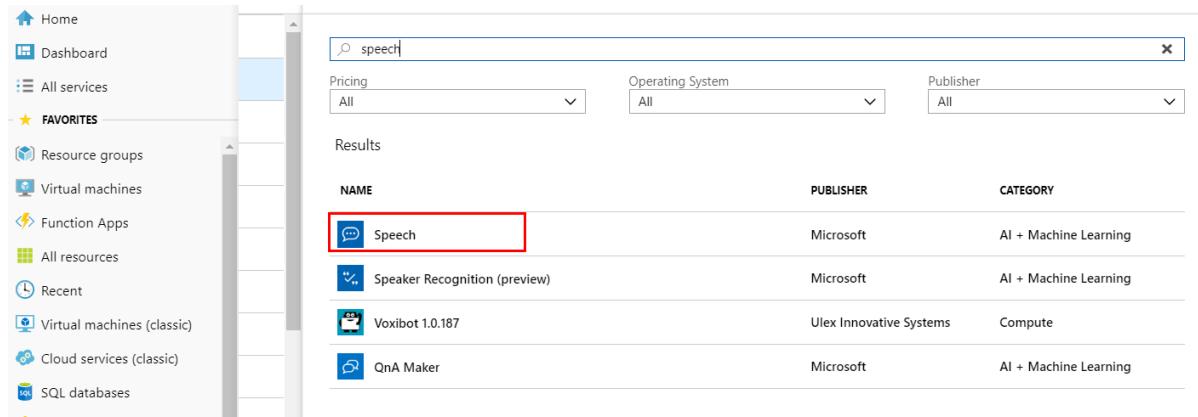
Before you begin

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

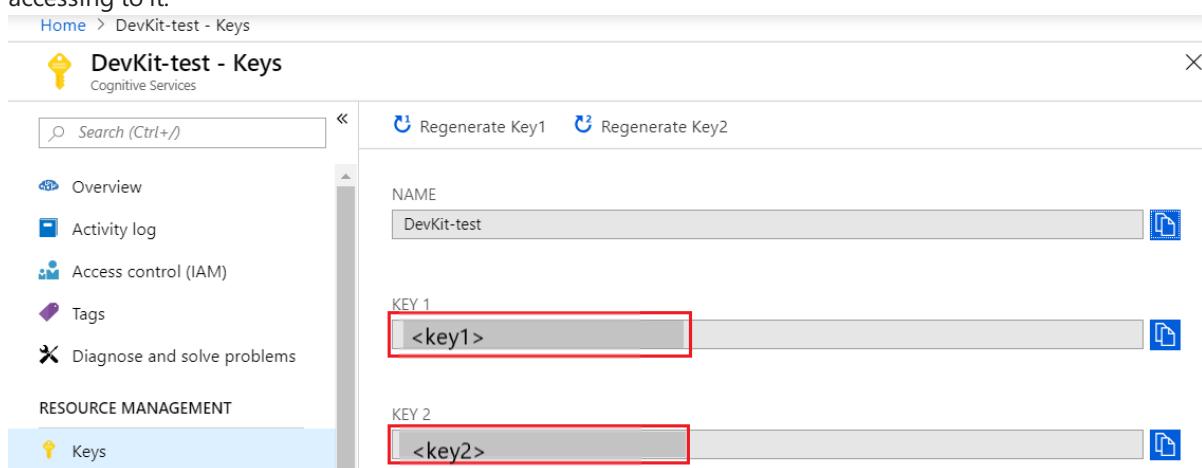
1. In the Azure portal, click **Create a resource** and search for **Speech**. Fill out the form to create the Speech Service.



A screenshot of 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 |

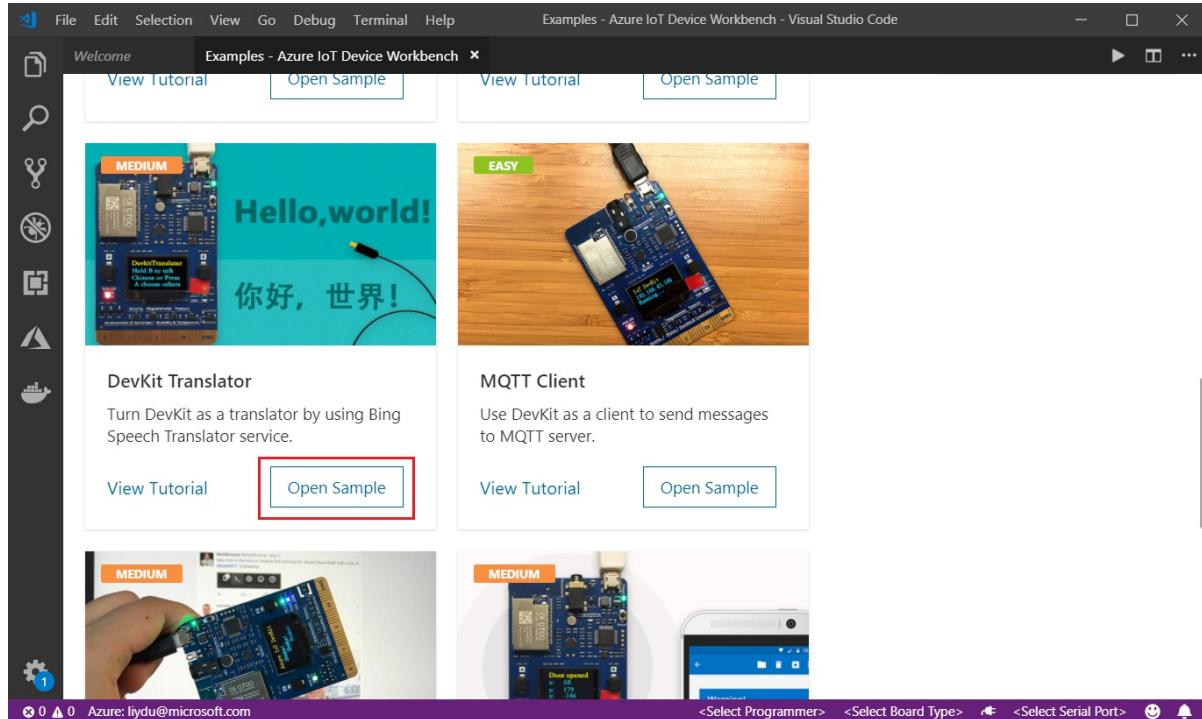
2. Go to the Speech service you just created, click **Keys** section to copy and note down the **Key1** for DevKit accessing to it.



A screenshot of the Azure portal showing the 'DevKit-test - Keys' page. The left sidebar includes links for Overview, Activity log, Access control (IAM), Tags, and Diagnose and solve problems. Under RESOURCE MANAGEMENT, the 'Keys' link is selected. The main area shows two key fields: 'KEY 1' containing '<key1>' and 'KEY 2' containing '<key2>'. Both fields have a red box highlighting them.

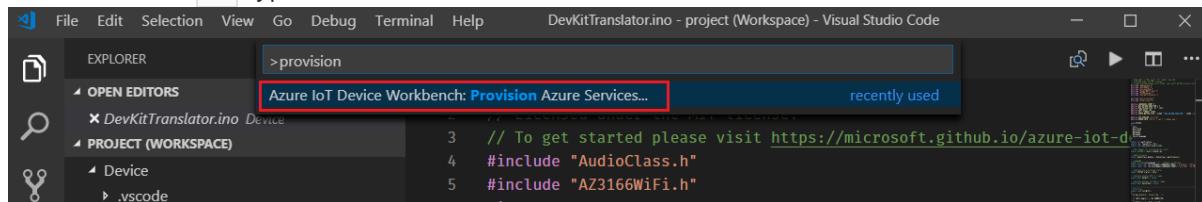
Open sample project

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.

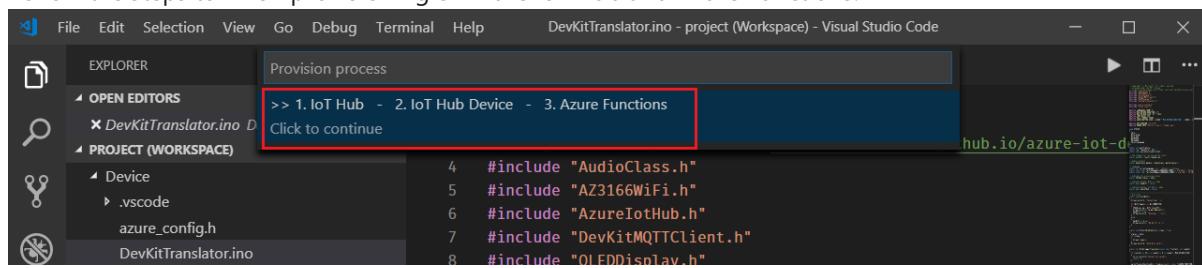


Use Speech Service with Azure Functions

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.

```

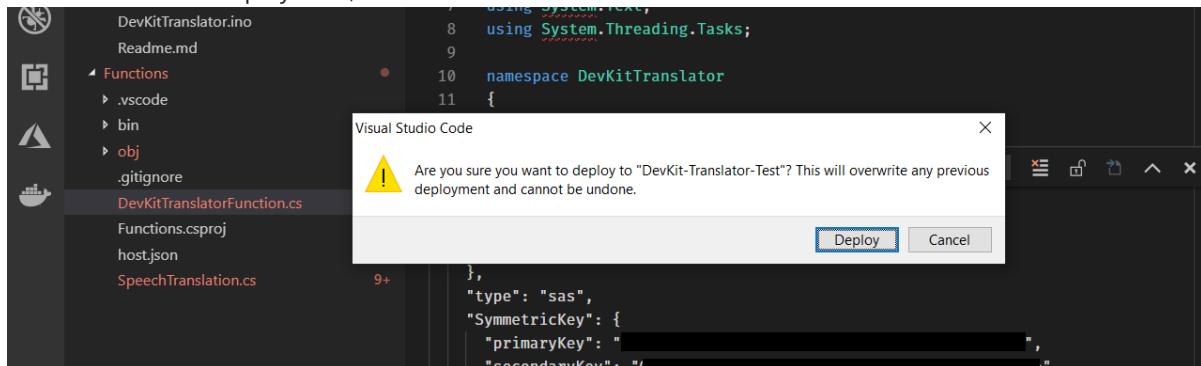
// Subscription Key of Speech Service
const string speechSubscriptionKey = "";

// Region of the speech service, see https://docs.microsoft.com/azure/cognitive-services/speech-
// service/regions for more details.
const string speechServiceRegion = "";

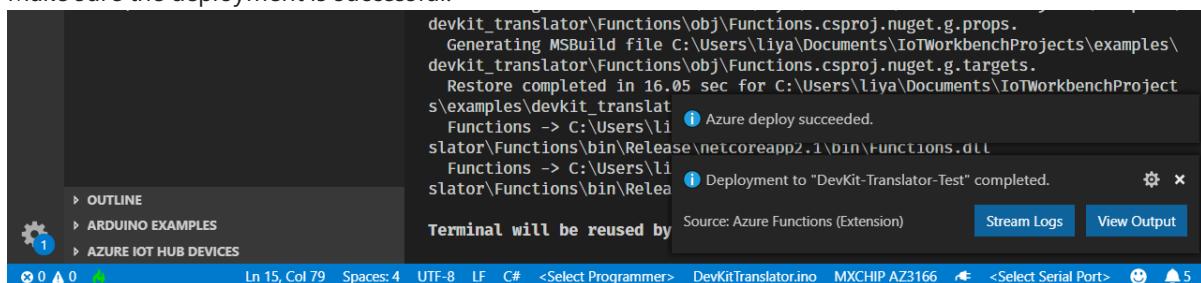
// Device ID
const string deviceName = "";

```

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.

```

{
  "generatedBy": "Microsoft.NET.Sdk.Functions-1.0.24",
  "configurationSource": "attributes",
  "bindings": [
    {
      "type": "httpTrigger",
      "methods": [
        "get",
        "post"
      ],
      "authLevel": "function",
      "name": "req"
    }
  ]
}

```

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

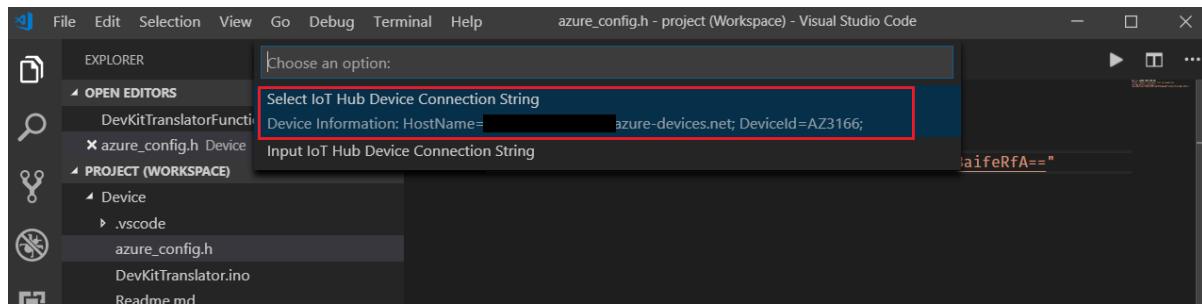
1. Switch the DevKit to **configuration mode** by:

- Hold down button A.
- Press and release **Reset** button.

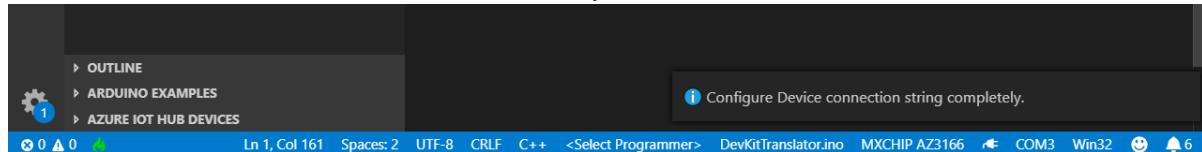
You will see the screen displays the DevKit ID and **Configuration**.



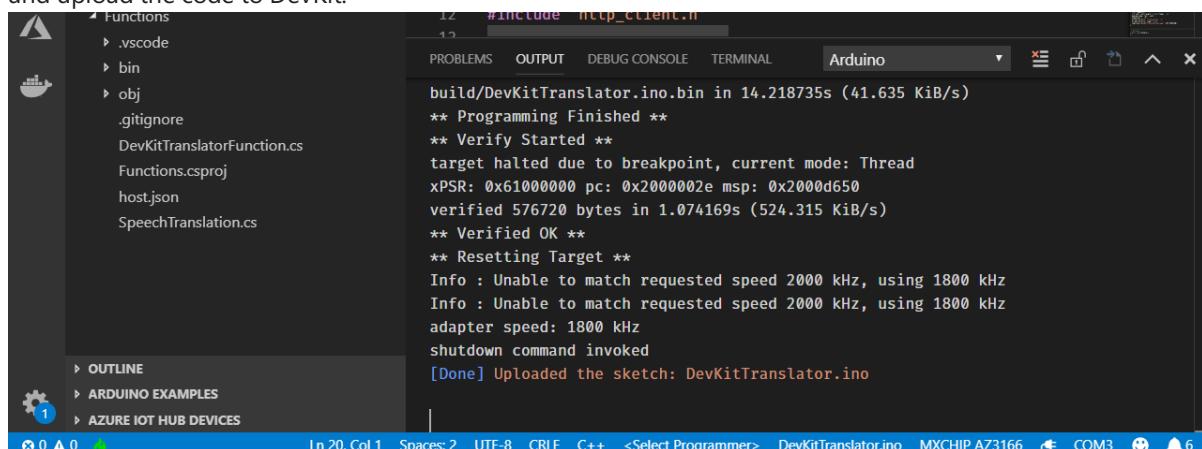
2. Click **F1**, type and select **Azure IoT Device Workbench: Configure Device Settings... > Config Device Connection String**. Select **Select IoT Hub Device Connection String** to configure it to the DevKit.



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.

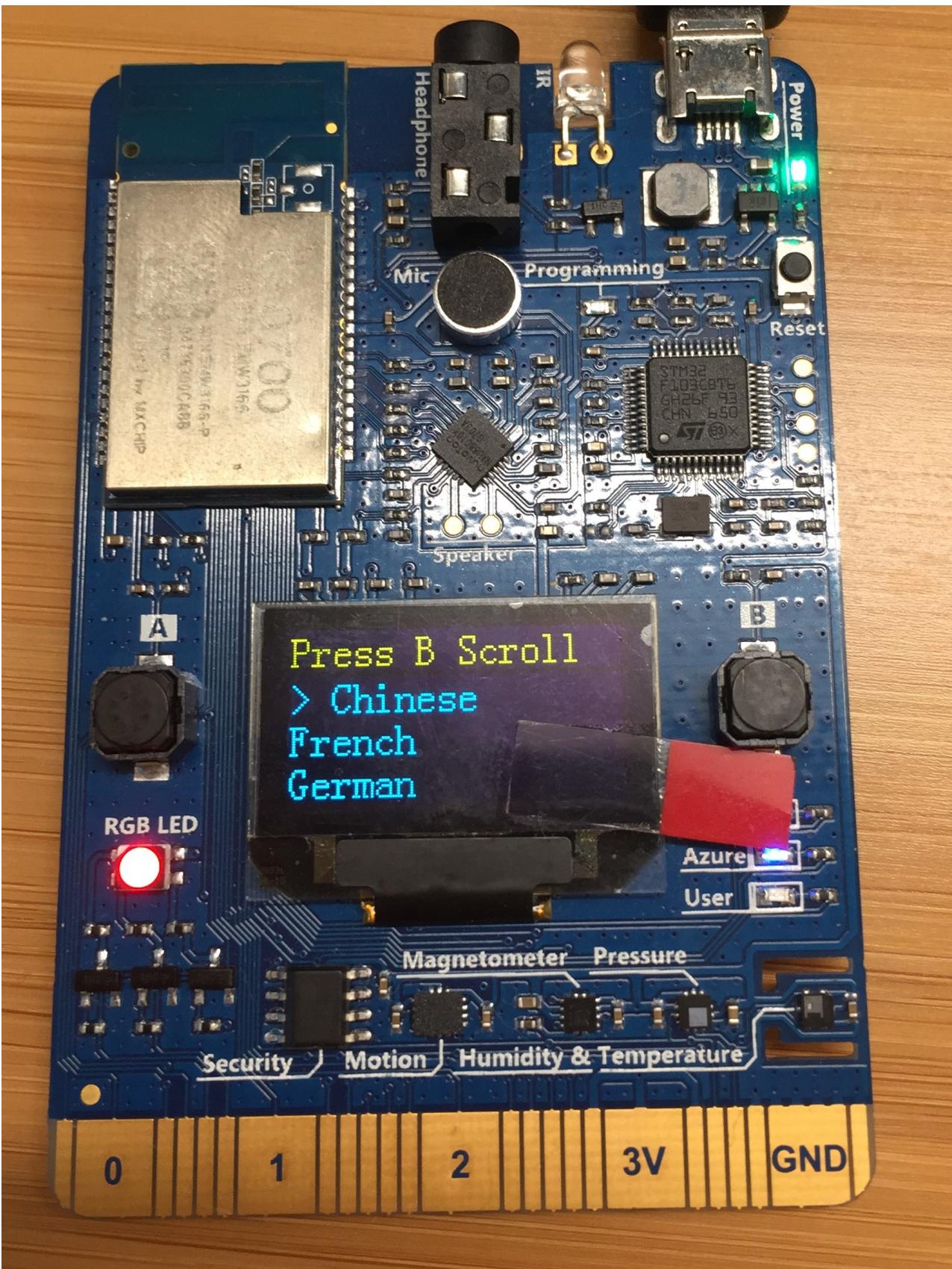


Test the project

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.

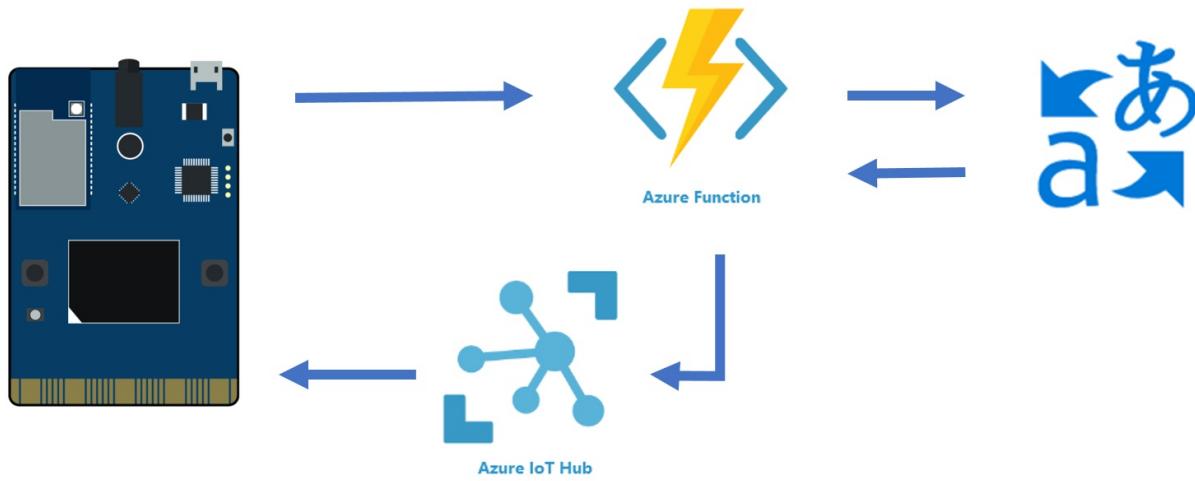




On the translation result screen, you can:

- 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

7/29/2020 • 5 minutes to read • [Edit Online](#)

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

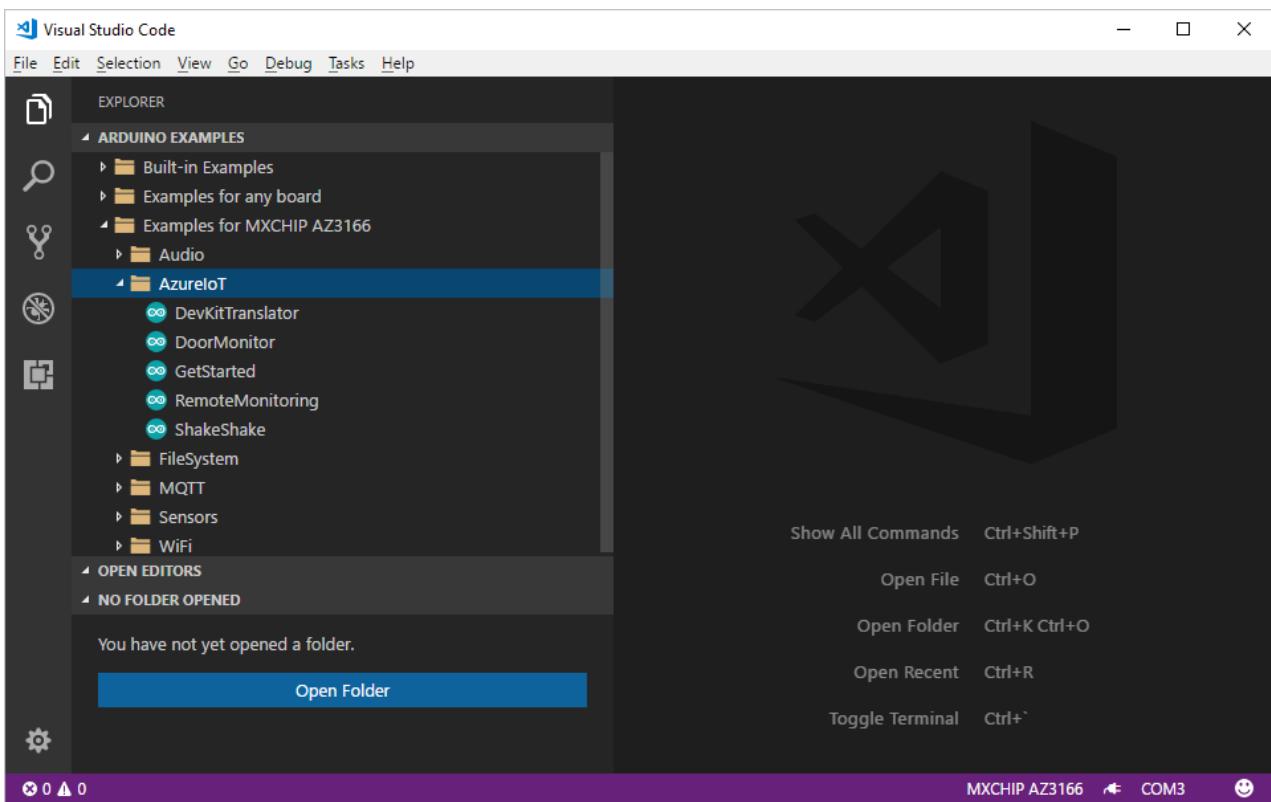
- Make sure your DevKit is connected to your computer.
- Start VS Code.
- Connect the DevKit to your computer.

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

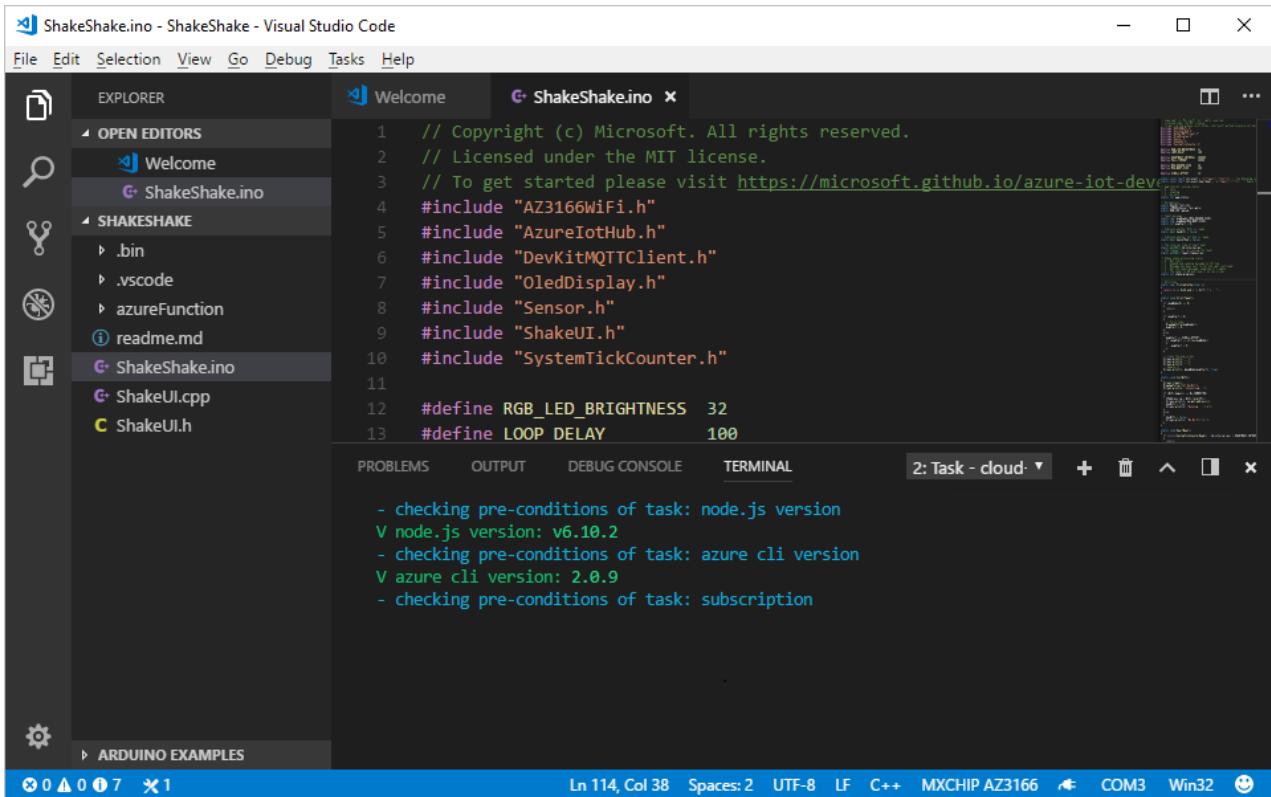


You can also open the sample project from command palette. Click `Ctrl+Shift+P` (macOS: `Cmd+Shift+P`) to open the command palette, type **Arduino**, and then find and select **Arduino: Examples**.

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

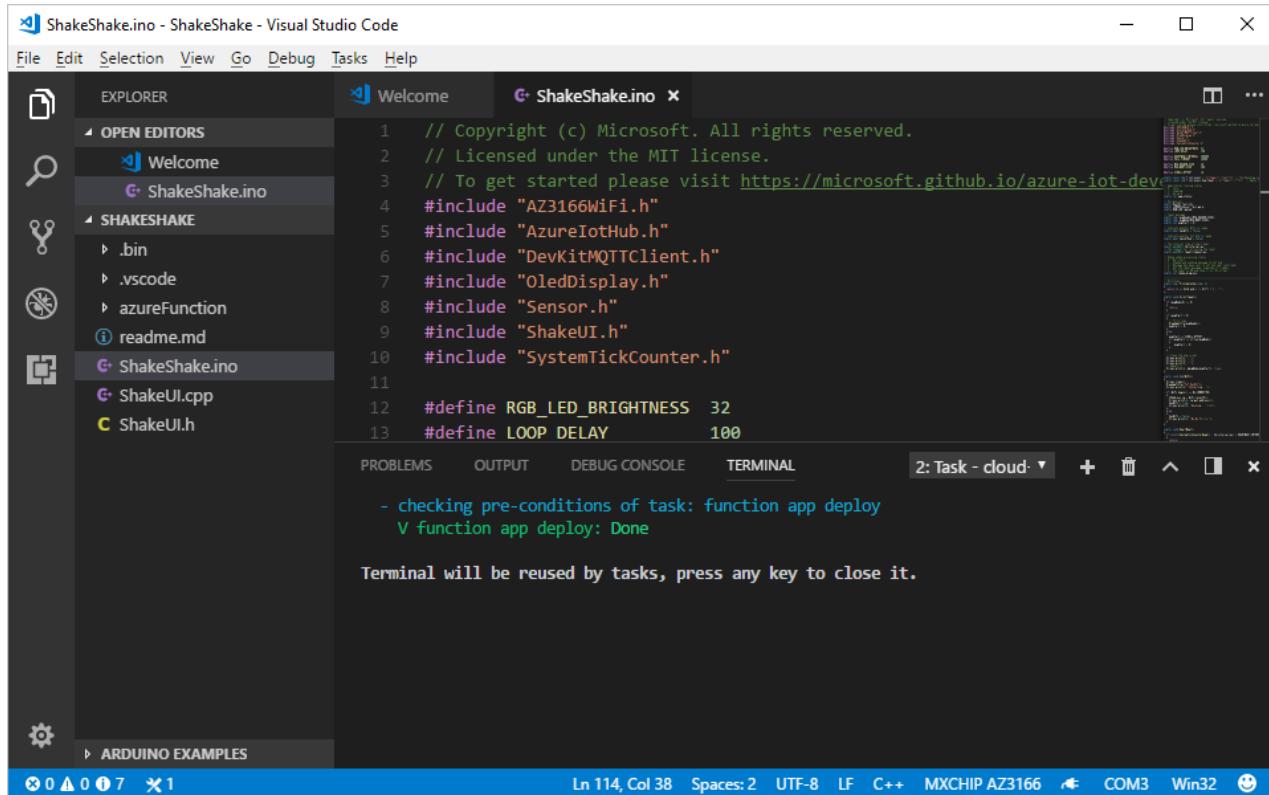
Open `ShakeShake.ino` and look for this line of code:

```
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:



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:

```
// 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

2: Task - device ▾ + 🗑️ ⌂ ✎ ×
```

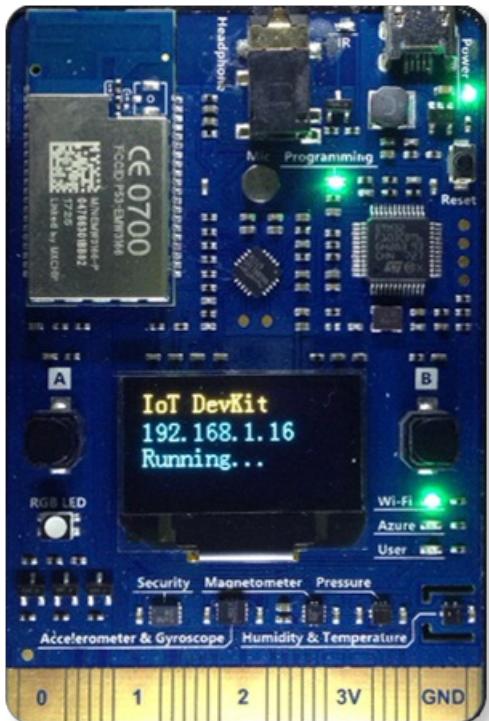
2. The DevKit reboots and starts running the code.

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](#).

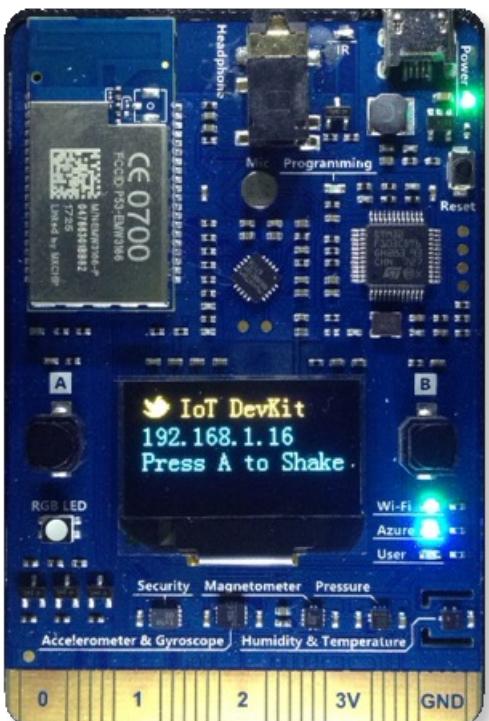
Test the project

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:

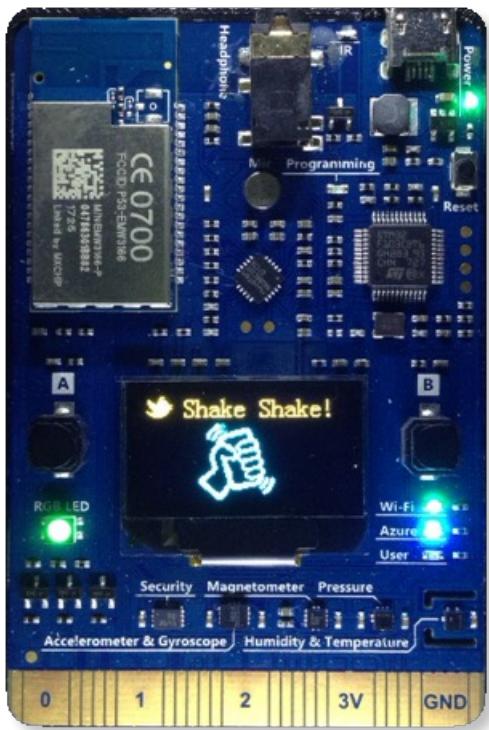
Arduino application initializing...



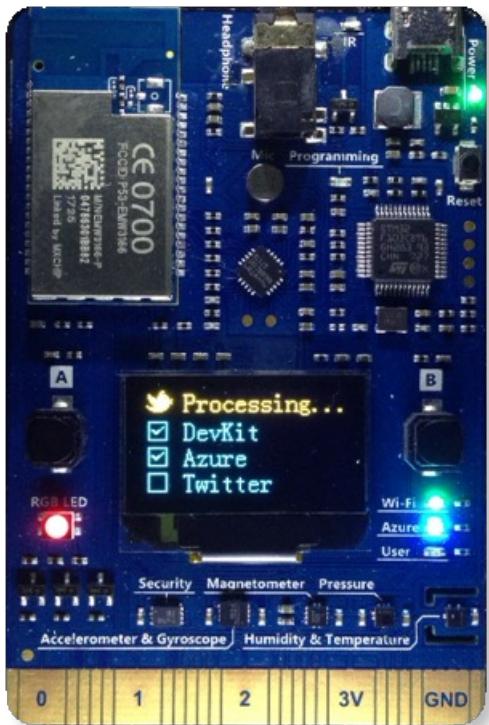
Press A to shake...



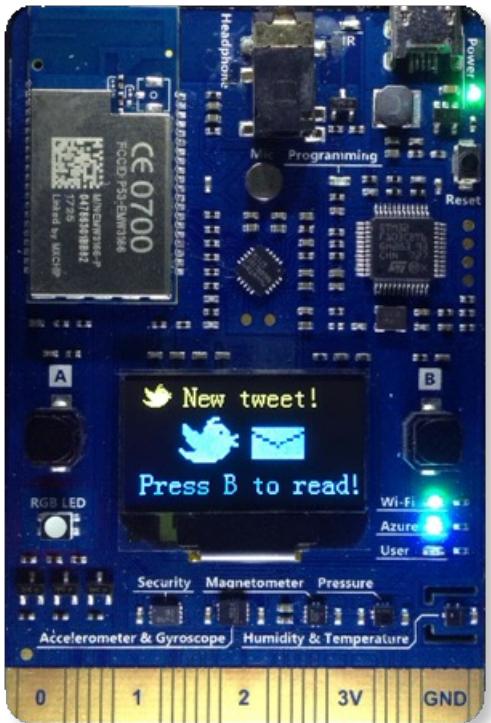
Ready to shake...



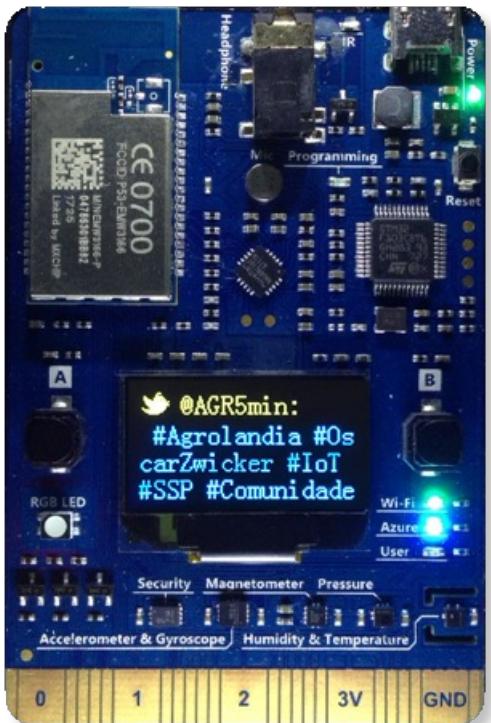
Processing...



Press B to read...

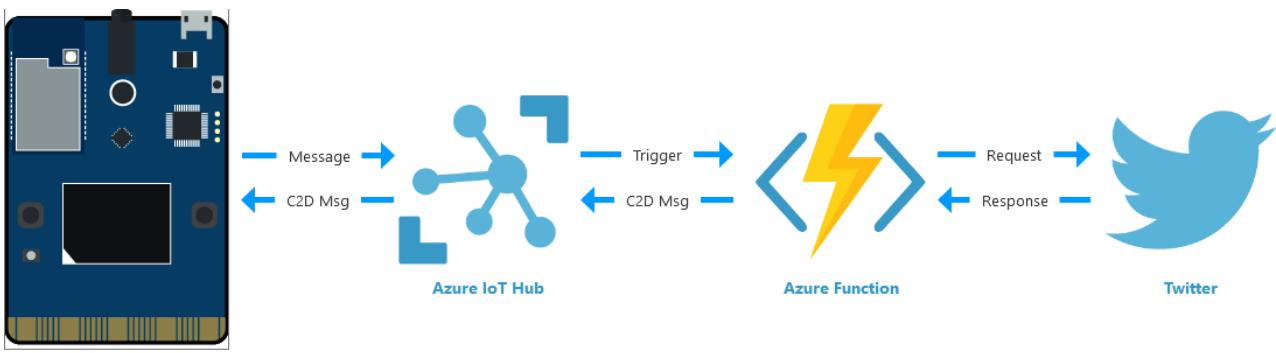


Display a random tweet...



- 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**: Filter dropdown.
- All locations**: Filter dropdown.
- No grouping**: Filter dropdown.
- 6 items**: List of resources:

| 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**: Editor dropdown.
- Function Apps**: List of apps.
- devkit-iot-hub-iohub-b0a4**: Selected function app.
- Functions**: Submenu with **shakeshake-cs** selected.
- Integrate**, **Manage**, **Monitor**: Function settings.
- Proxies (preview)**: Submenu.
- Slots (preview)**: Submenu.
- Code Editor (run.csx):**

```

string authHeader = "Bearer " + "[your own token]";

42   {
43     throw new ShakeShakeException($"Failed to deserialize message:{myEventHubMessage}");
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=%23";
57     string authHeader = "Bearer " + "AAAAAAAAAAAAAAAAGVU0AAAAAAUcpxA9aXc2T06";
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

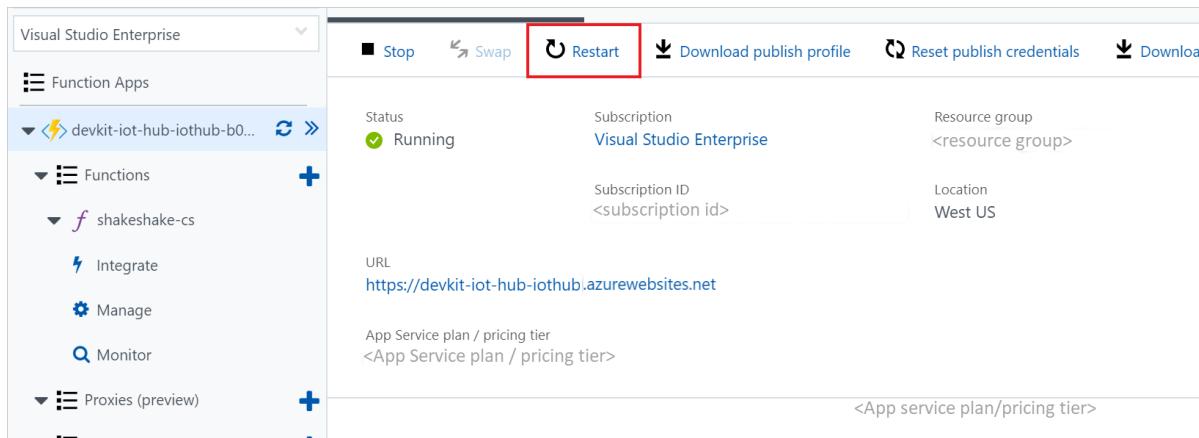
How to troubleshoot problems or provide feedback.

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'. Under 'Functions', there are options for 'Integrate', 'Manage', and 'Monitor'. At the bottom of this sidebar is a section for 'Proxies (preview)'. On the right, the main content area displays details for the 'shakeshake-cs' function. It shows the status as 'Running', subscription as 'Visual Studio Enterprise', resource group as '<resource group>', and location as 'West US'. It also lists the URL as 'https://devkit-iot-hub-iothub.azurewebsites.net' and the App Service plan / pricing tier as '<App Service plan / pricing tier>'. At the top of this content area, there are several buttons: 'Stop', 'Swap', 'Restart' (which is highlighted with a red box), 'Download publish profile', 'Reset publish credentials', and 'Download'.

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

7/29/2020 • 2 minutes to read • [Edit Online](#)

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

```

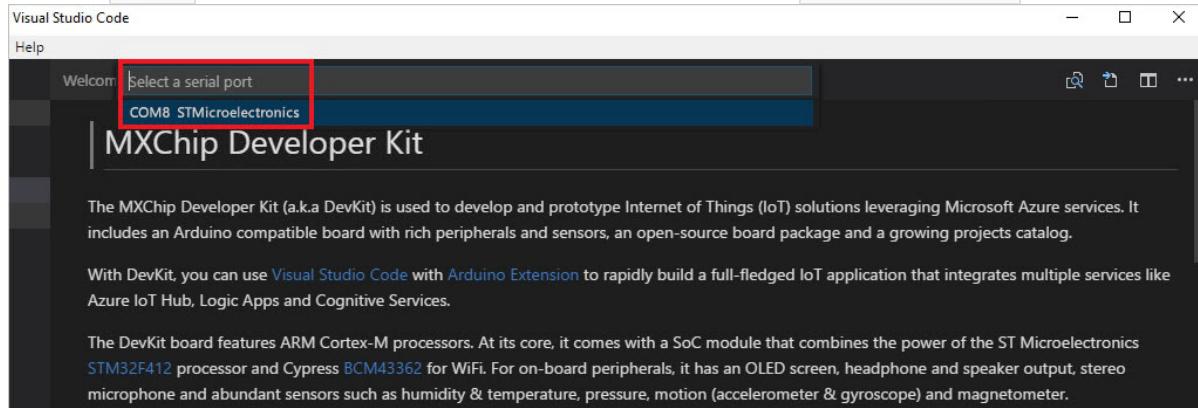
NOTE

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](#).

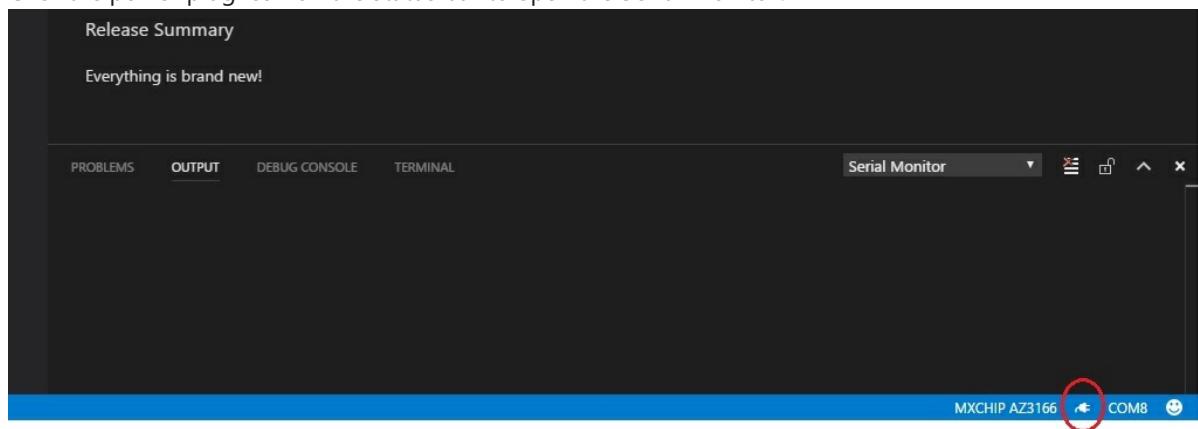
Test the project

In VS Code, follow this procedure to open and set up the Serial Monitor:

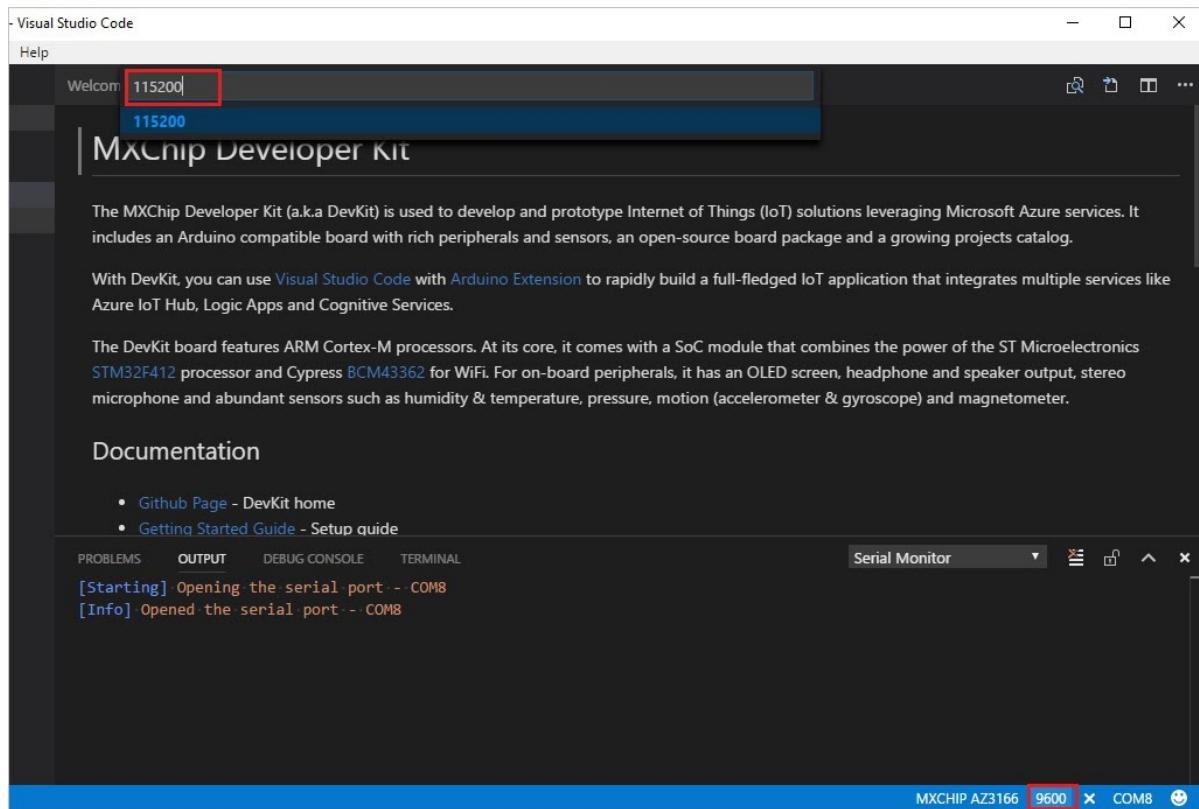
1. Click the `COM[X]` word on the status bar to set the right COM port with `STMicroelectronics`:



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 is the suggested next step: [Azure IoT Remote Monitoring solution accelerator overview](#)

Door Monitor -- Using Azure Functions and SendGrid, send email when a door is opened

7/29/2020 • 5 minutes to read • [Edit Online](#)

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 SendGrid Settings page for a 'DoorMonitorSendGrid' account. The left sidebar has a 'Manage' button highlighted with a red box. The main area displays 'Pricing Information' showing a 'FREE' tier with 25,000 emails/month. On the right, under 'Settings', there is a 'Support + Troubleshooting' section with an 'Activity log' link, and a 'GENERAL' section containing 'Properties', 'Configurations', and 'Contact Information'. Below these are sections for 'RESOURCE MANAGEMENT' like 'Tags', 'Locks', 'Users', and 'Automation script'.

On the SendGrid page, click **Settings** > **API Keys** > **Create API Key**.

The screenshot shows the 'API Keys' creation page. The left sidebar has a 'Settings' button highlighted with a red box, and an 'API Keys' button also highlighted with a red box. The main area features a 'Create API Key' button highlighted with a red box. A key icon and the text 'Get started creating API Keys' are displayed, along with a descriptive paragraph about API keys.

On the **Create API Key** page, input the **API Key Name** and click **Create & View**.

Create API Key

API Key Name • ⓘ

API Key Permissions • ⓘ

 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

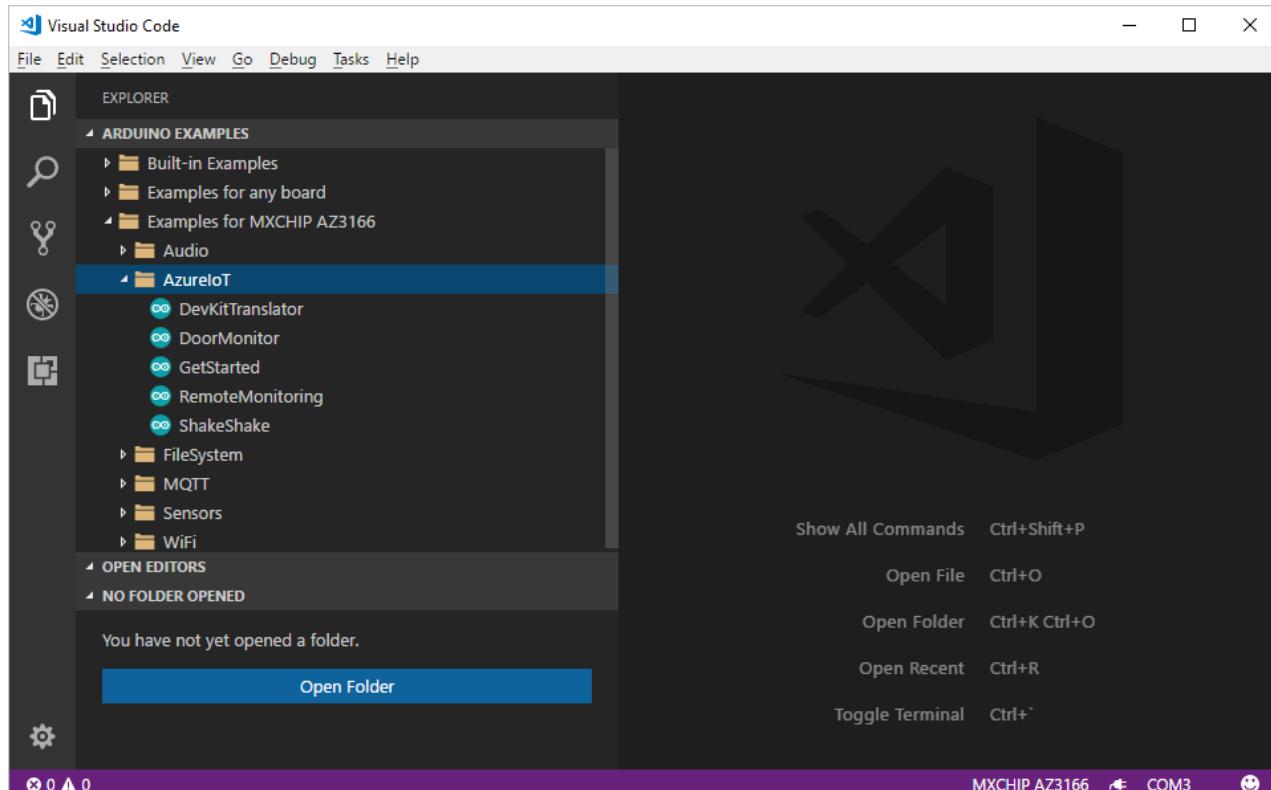
- Make sure your DevKit is **not** connected to your computer.
- Start VS Code.
- Connect the DevKit to your computer.

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.

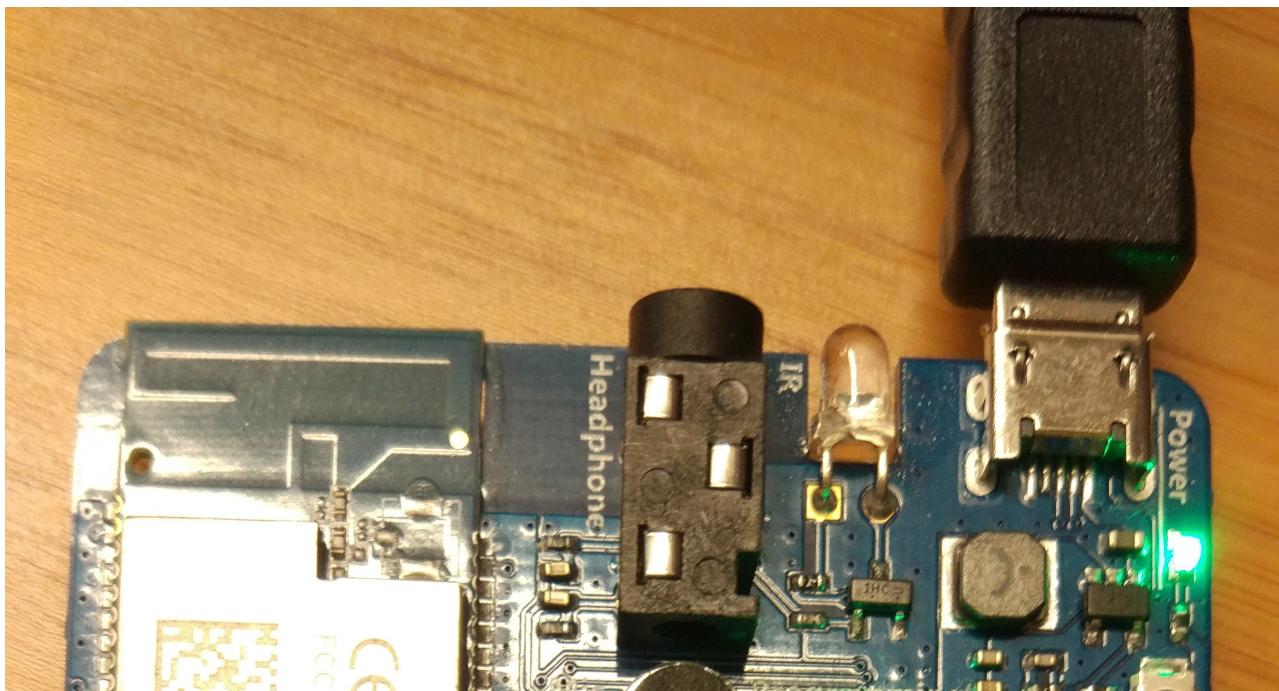
NOTE

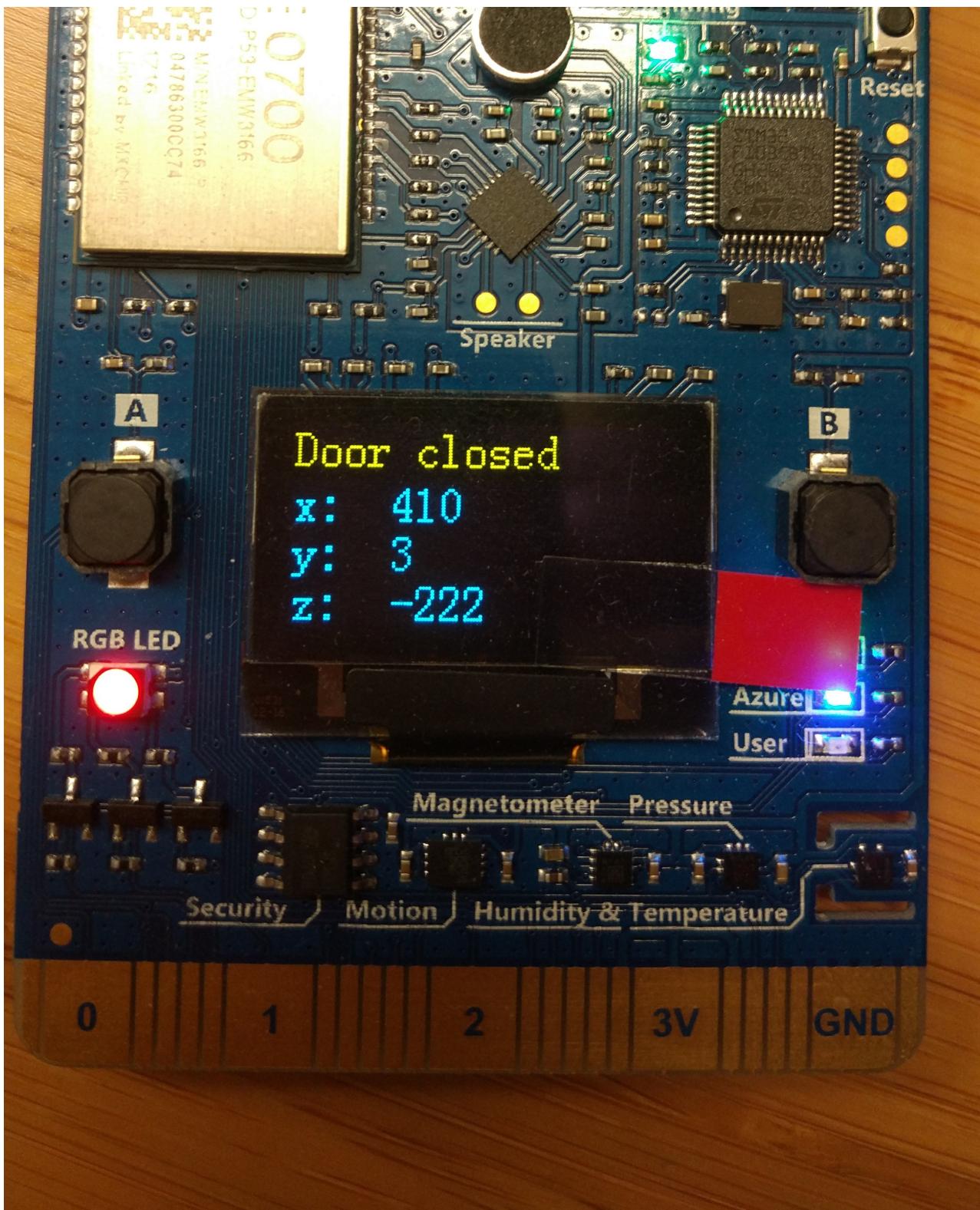
Occasionally, you may receive an "Error: AZ3166: Unknown package" error message. This error occurs when the board package index is not refreshed correctly. To resolve this error, refer to the [development section of the IoT DevKit FAQ](#).

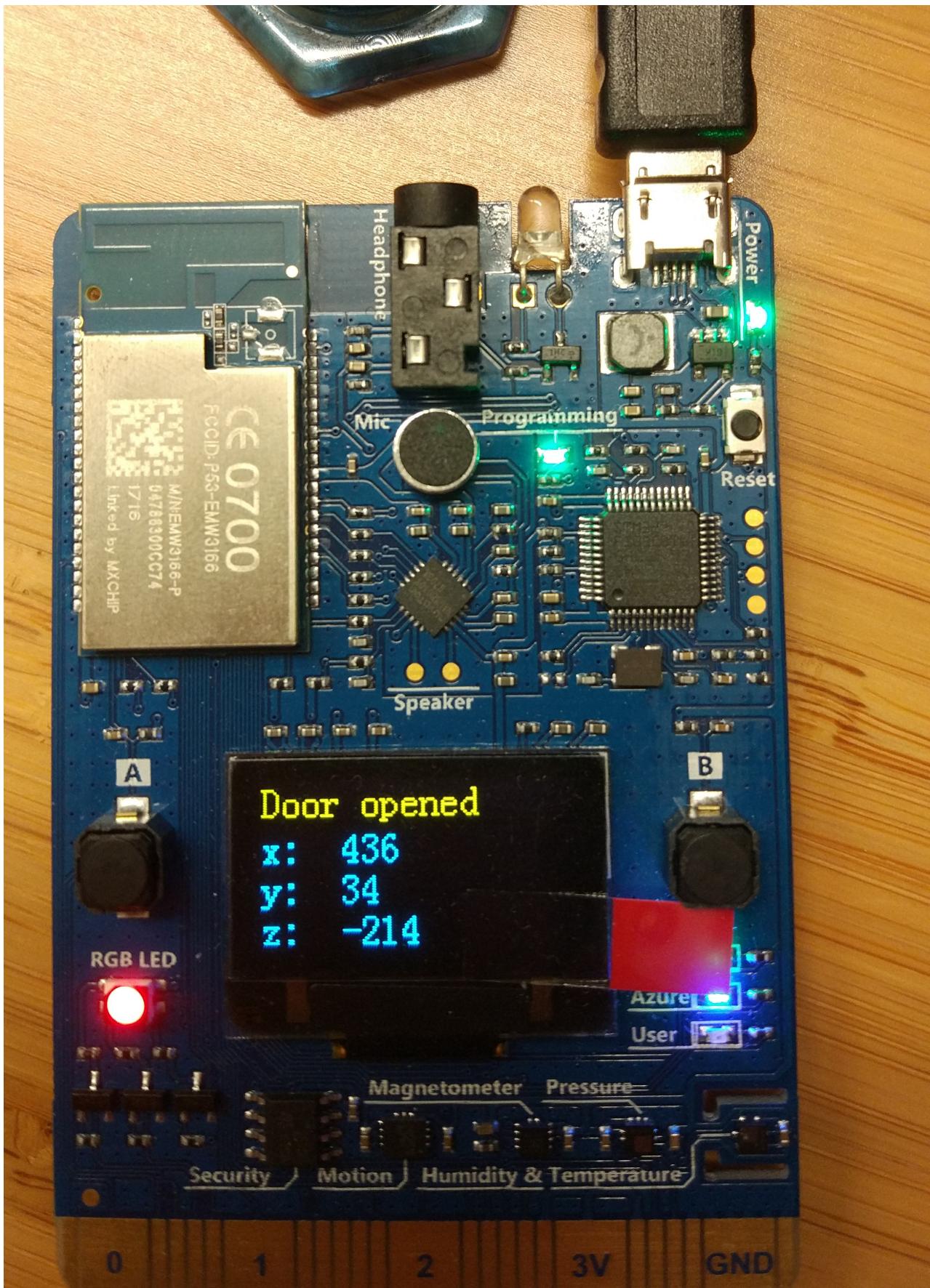
Test the project

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:

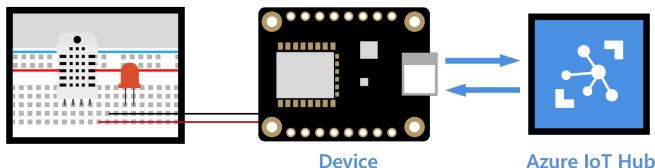
- [Gitter.im](#)
- [Stack Overflow](#)

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 is the suggested next step:[Azure IoT Remote Monitoring solution accelerator overview](#)

Use Azure IoT Tools for Visual Studio Code to send and receive messages between your device and IoT Hub

7/29/2020 • 2 minutes to read • [Edit Online](#)



[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](#).

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](#) or copy and paste this URL into a browser window:
`vscode://extension/vsciot-vscode.azure-iot-tools`

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 **iothubowner** policy 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 Built-in Event Endpoint**.
2. The monitored messages will be shown in **OUTPUT > Azure IoT Hub** view.
3. To stop monitoring, right-click the **OUTPUT** view and select **Stop Monitoring Built-in Event Endpoint**.

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** view.

Next steps

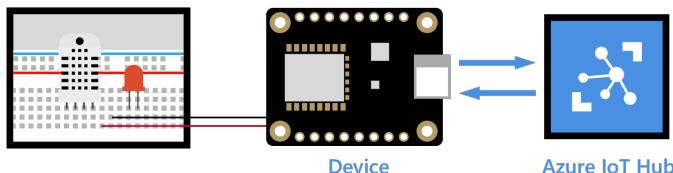
You've learned how to monitor device-to-cloud messages and send cloud-to-device messages between your IoT device and Azure IoT Hub.

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 extension for Visual Studio Code](#)
- [Manage devices with Azure IoT Hub extension for Visual Studio Code](#)
- [Set up message routing](#)
- [Use Power BI to visualize real-time sensor data from your IoT hub](#)
- [Use a web app 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

11/14/2019 • 2 minutes to read • [Edit Online](#)



[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 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 learn

In this article, you 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 do

In this article, you do the following tasks:

- 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

You need the following prerequisites:

- An active Azure subscription.
- An Azure IoT Hub under your subscription.
- Microsoft Visual Studio 2017 Update 9 or later. This article uses [Visual Studio 2019](#).
- The Cloud Explorer component from Visual Studio Installer, which is selected by default with Azure Workload.

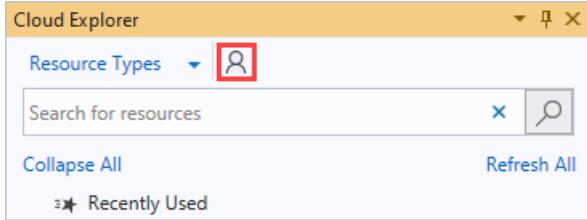
Update Cloud Explorer to latest version

The Cloud Explorer component from Visual Studio Installer for Visual Studio 2017 only supports monitoring device-to-cloud and cloud-to-device messages. To use Visual Studio 2017, download and install the latest [Cloud Explorer](#).

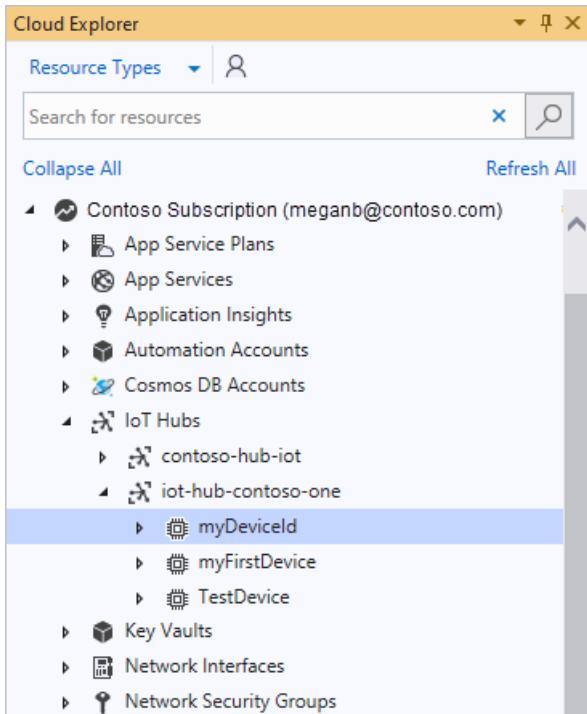
Sign in to access your hub

To access your hub, follow these steps:

1. In Visual Studio, select **View > Cloud Explorer** to open Cloud Explorer.
2. Select the Account Management icon to show your subscriptions.



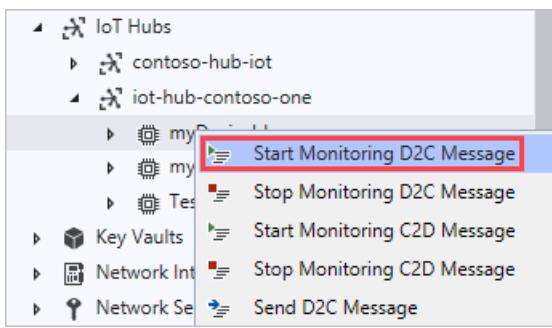
3. If you are signed in to Azure, your accounts appear. To sign into Azure for the first time, choose **Add an account**.
4. Select the Azure subscriptions you want to use and choose **Apply**.
5. Expand your subscription, then expand **IoT Hubs**. Under each hub, you can see your devices for that hub.



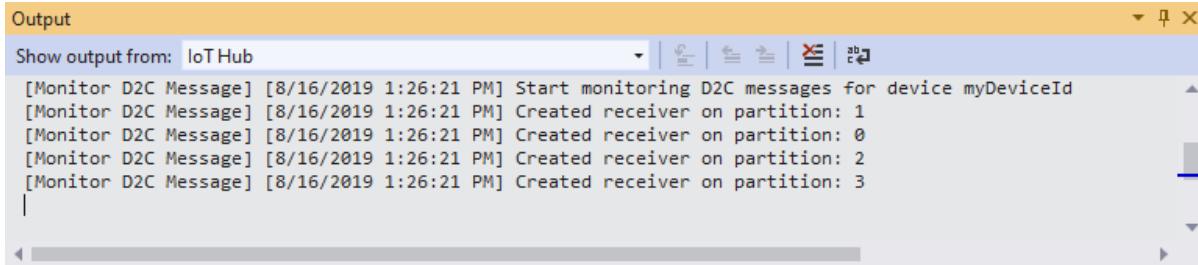
Monitor device-to-cloud messages

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 appear under **Output**.

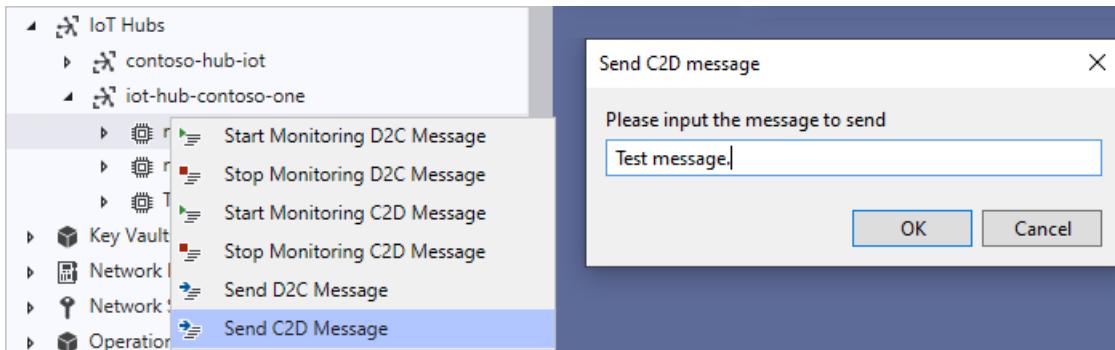


3. To stop monitoring, right-click on any IoT Hub or device and select **Stop Monitoring D2C Message**.

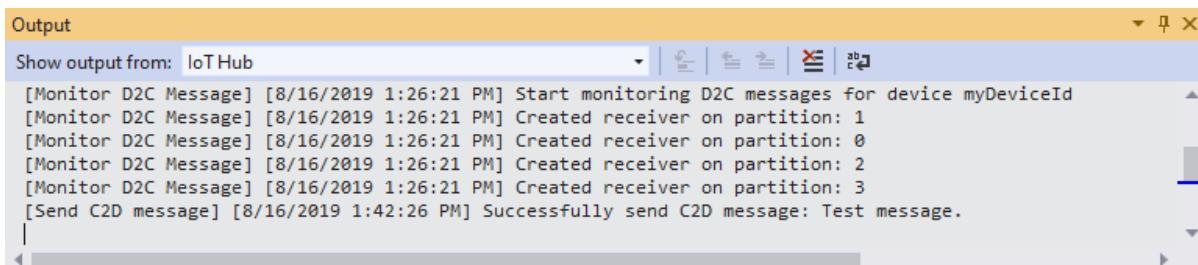
Send cloud-to-device messages

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.



Results appear under **Output**.



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.

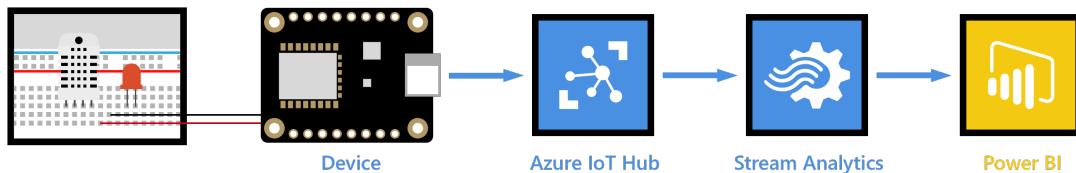
To continue to get started with Azure IoT Hub and to explore all extended IoT scenarios, see the following:

- [Manage cloud device messaging with Azure IoT Hub extension for Visual Studio Code](#)

- Manage devices with Azure IoT Hub extension for Visual Studio Code
- Set up message routing
- Use Power BI to visualize real-time sensor data from your IoT hub
- Use a web app 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

7/29/2020 • 6 minutes to read • [Edit Online](#)



NOTE

Before you start this tutorial, complete the [Raspberry Pi online simulator](#) tutorial or one of the device tutorials; for example, [Raspberry Pi with node.js](#). In these articles, 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 a web app, see [Use a web app 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

- Complete the [Raspberry Pi online simulator](#) tutorial or one of the device tutorials; for example, [Raspberry Pi with node.js](#). These articles cover 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](#) provide independent views into the event stream that enable apps and Azure services to independently consume data from the same Event Hub endpoint. In this section, you add a consumer group to

your IoT hub's built-in endpoint that is used later in this tutorial to pull data from the endpoint.

To add a consumer group to your IoT hub, follow these steps:

1. In the [Azure portal](#), open your IoT hub.
2. On the left pane, select **Built-in endpoints**, select **Events** on the right pane, and enter a name under **Consumer groups**. Select **Save**.

The screenshot shows the 'contoso-hub-1 - Built-in endpoints' blade in the Azure portal. The left sidebar lists various IoT Hub settings like Overview, Activity log, Access control (IAM), Tags, Events, and more. Under 'Built-in endpoints', the 'Events' section is selected. It displays configuration for the 'Events' endpoint, including the number of partitions (4), event hub-compatible name (contoso-hub-1), event hub-compatible endpoint (Endpoint=sb://iothub-ns-contoso-hu-1477762-53f6652df9.servicebus.windows.net/SharedAccessKeyName=iothubowner;SharedAccessKe...), and retention settings. A red box highlights the 'Create new consumer group' input field under the 'CONSUMER GROUPS' section, which currently contains '\$Default'. Below this, the 'Cloud to device messaging' section is shown with message retention and delivery count settings.

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](#), select **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.

New Stream Analytics job

This will create a new Stream Analytics job. You will be charged according to Azure Stream Analytics billing model. Learn more. →

Job name *

 ✓

Subscription *

 ✓

Resource group *

 ✓

[Create new](#)

Location *

 ✓

Hosting environment ⓘ

Cloud Edge

Streaming units (1 to 192) ⓘ

3

[Create](#)

3. Select **Create**.

Add an input to the Stream Analytics job

1. Open the Stream Analytics job.
2. Under **Job topology**, select **Inputs**.
3. In the **Inputs** pane, select **Add stream input**, then select **IoT Hub** from the drop-down list. On the new input pane, enter the following information:

Input alias: Enter a unique alias for the input.

Select IoT Hub from your subscription: Select this radio button.

Subscription: Select the Azure subscription you're using for this tutorial.

IoT Hub: Select the IoT Hub you're using for this tutorial.

Endpoint: Select **Messaging**.

Shared access policy name: Select the name of the shared access policy you want the Stream Analytics job to use for your IoT hub. For this tutorial, you can select *service*. The *service* policy is created by default on new IoT hubs and grants permission to send and receive on cloud-side endpoints exposed by the IoT hub. To learn more, see [Access control and permissions](#).

Shared access policy key: This field is auto-filled based on your selection for the shared access policy name.

Consumer group: Select the consumer group you created previously.

Leave all other fields at their defaults.

The screenshot shows the Azure Stream Analytics Job Overview page for a job named "PowerBiVisualizationJob". The left sidebar has sections like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings (Properties, Locks), Job topology (Inputs selected), Functions, Query, Outputs, Configure (Storage account settings, Scale, Locale, Event ordering, Error policy, Compatibility level), and Help. The main area shows an "Inputs" section with a table for "Add stream input" and "Add reference input". A "New input" card for "IoT Hub" is open, showing fields: Input alias * (PowerBiVisualizationInput), Provide IoT Hub settings manually (radio button), Select IoT Hub from your subscriptions (radio button selected), Subscription (Azure subscription 1), IoT Hub (contoso-hub-1), Endpoint (Messaging), Shared access policy name (service), Shared access policy key (redacted), Consumer group (power-bi-visualization-cg), Partition Key (empty), and Event serialization format * (JSON). A "Save" button is at the bottom right.

4. Select Save.

Add an output to the Stream Analytics job

1. Under Job topology, select Outputs.
2. In the Outputs pane, select Add and Power BI.
3. On the Power BI - New output pane, select Authorize and follow the prompts to sign in to your Power BI account.
4. After you've signed in to Power BI, enter the following information:

Output alias: A unique alias for the output.

Group workspace: Select your target group workspace.

Dataset name: Enter a dataset name.

Table name: Enter a table name.

Authentication mode: Leave at the default.

Power Bi

New output

Currently authorized as **Abe User** ([auser@contosotest1960.onmicrosoft.com](#))

Output alias * ✓

Group workspace ▼

Dataset name * ✓

Table name * ✓

Authentication mode ▼

Note: You are granting this output permanent access to your Power BI dashboard. Should you need to revoke this access in the future you can do one of the following:
 1. Change the user account password.
 2. Delete this output.
 3. Delete this job.

Save

5. Select **Save**.

Configure the query of the Stream Analytics job

- Under **Job topology**, select **Query**.
- Replace **[YourInputAlias]** with the input alias of the job.
- Replace **[YourOutputAlias]** with the output alias of the job.

PowerBiVisualizationJob | Query

Inputs (1)
PowerBiVisualizationInput

Outputs (1)
PowerBiVisualizationOutput

```

1  SELECT
2  *
3  INTO
4  PowerBiVisualizationOutput
5  FROM
6  PowerBiVisualizationInput

```

Test results

Test query to show results here

4. Select **Save query**.

Run the Stream Analytics job

In the Stream Analytics job, select **Overview**, then select **Start > Now > Start**. Once the job successfully starts, the job status changes from **Stopped** to **Running**.

PowerBiVisualizationJob

Stream Analytics job

Search (Ctrl+ /)

Start Stop Delete

Your job is running. Learn how to setup alerts and monitor your job using metrics. →

Resource group (change)
contoso-hub-rgrp

Status
Running

Location
West US

Subscription (change)
Azure subscription 1

Subscription ID

Send feedback
UserVoice

Created
Wednesday, June 3, 2020, 2:16:48 PM

Started
Wednesday, June 3, 2020, 3:28:31 PM

Output watermark
Wednesday, June 3, 2020, 3:30:15 PM

Hosting environment
Cloud

Overview

Inputs
1
PowerBiVisualizationInput IoT Hub

Outputs
1
PowerBiVisualizationOutput Power BI

Query

```
1 SELECT
2 *
3 INTO
4 PowerBiVisualizationOutput
5 FROM
6 PowerBiVisualizationInput
```

Create and publish a Power BI report to visualize the data

The following steps show you how to create and publish a report using the Power BI service. You can follow these steps, with some modification, if you want to use the "new look" in Power BI. To understand the differences and how to navigate in the "new look", see [The 'new look' of the Power BI service](#).

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. Select the workspace you used, [My Workspace](#).
4. Select **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, select **Add Report** (the first icon to the right of the dataset name).

The screenshot shows the Power BI desktop application. On the left, there's a navigation pane with sections like Home, Favorites, Recent, Apps, Shared with me, Learn, Workspaces, and My workspace. Under My workspace, there are sections for Reports, Workbooks, and Datasets. The Datasets section is currently active, showing one item: 'PowerBiVisualizationDataSet'. The main area displays the dataset details with columns for NAME, ENDORSER, and ACTIONS. The ACTIONS column for the dataset has five icons: a chart (highlighted with a red box), a person, a pencil, a delete, and three dots.

6. Create a line chart to show real-time temperature over time.

- On the **Visualizations** pane of the report creation page, select the line chart icon to add a line chart.
- On the **Fields** pane, expand the table that you specified when you created the output for the Stream Analytics job.
- 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.

This screenshot shows the Power BI report creation interface. The left sidebar includes Home, Favorites, Recent, Apps, Shared with me, Learn, Workspaces, and My workspace. The main area features a canvas with a line chart titled 'Temperature by EventEnqueuedUtcTime'. To the right are the **Visualizations** and **FIELDS** panes. The **Visualizations** pane shows the chart settings with 'EventEnqueuedUtcTime' assigned to the axis and 'temperature' assigned to the values. The **FIELDS** pane shows the 'PowerBiVisualizationDataSet' table with two fields selected: 'EventEnqueuedUtcTime' and 'temperature', both highlighted with red boxes.

7. Create another line chart to show real-time humidity over time. To do this, click on a blank part of the canvas and follow the same steps above to place **EventEnqueuedUtcTime** on the x-axis and **humidity**

on the y-axis.

The screenshot shows the Power BI desktop interface. On the left, the navigation pane is visible with sections like Home, Favorites, Recent, Apps, Shared with me, Learn, Workspaces, and My workspace. Under My workspace, there are Reports, Workbooks, and Datasets. The main area displays two line charts side-by-side. The left chart is titled 'Temperature by EventEnqueuedUtcTime' and the right chart is titled 'Humidity by EventEnqueuedUtcTime'. Both charts have 'EventEnqueuedUtcTime' on the x-axis and 'Temperature' and 'Humidity' on the y-axis. To the right of the charts, the 'Visualizations' pane is open, showing various chart types. The 'FIELDS' pane is also open, displaying fields from the dataset: deviceId, EventEnqueuedUtcTime (selected), EventProcessedUtcTime, humidity (selected), IoTHub, messageId, PartitionId, and temperature. The 'Values' section of the Fields pane has 'humidity' selected. A red box highlights the 'EventEnqueuedUtcTime' field in the Visualizations pane and the 'humidity' field in the Fields pane.

8. Select **Save** to save the report.

9. Select **Reports** on the left pane, and then select the report that you just created.

10. Select **File > Publish to web**.

The screenshot shows the Power BI desktop interface with the 'File' menu open. The 'File' menu contains several options: Save as, Print, Publish to web (which is highlighted with a red box), Export to PowerPoint, Export to PDF, and Download report (Preview). To the right of the menu, a preview of the report is shown, featuring two line charts: 'Humidity by EventEnqueuedUtcTime' and 'Temperature by EventEnqueuedUtcTime'. The x-axis for both charts is 'EventEnqueuedUtcTime' and the y-axis is 'humidity' (as set in the previous step). The 'Filters' pane is also visible on the right side of the screen.

NOTE

If you get a notification to contact your administrator to enable embed code creation, you may need to contact them. Embed code creation must be enabled before you can complete this step.

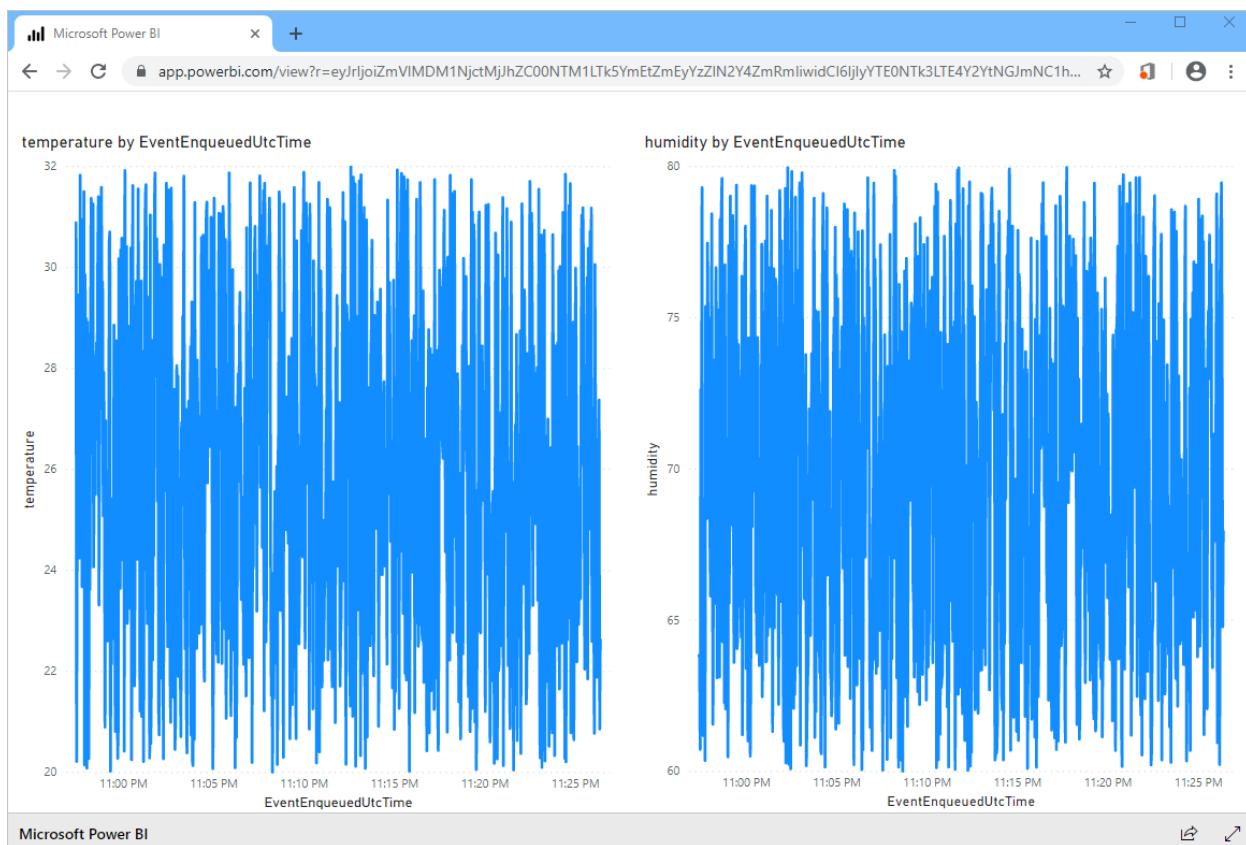
Contact your admin to enable embed code creation

To publish this report on the web, ask your Power BI admin if they will allow you to create new publish to web embed codes. Once they turn that on, you will be able to publish this report to the web. [Learn more](#)

OK

11. Select **Create embed code**, and then select **Publish**.

You're provided the report link that you can share with anyone for report access and a code snippet that you can use 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.

For another way to visualize data from Azure IoT Hub, see [Use a web app to visualize real-time sensor data from Azure IoT Hub](#).

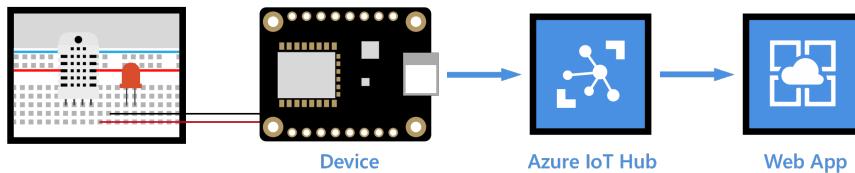
To continue to get started with Azure IoT Hub and to explore all extended IoT scenarios, see the following:

- [Manage cloud device messaging with Azure IoT Hub extension for Visual Studio Code](#)
- [Manage devices with Azure IoT Hub extension for Visual Studio Code](#)

- Set up message routing
- Use Power BI to visualize real-time sensor data from your IoT hub
- Use a web app 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 in a web application

7/29/2020 • 12 minutes to read • [Edit Online](#)



NOTE

Before you start this tutorial, complete the [Raspberry Pi online simulator](#) tutorial or one of the device tutorials; for example, [Raspberry Pi with node.js](#). In these articles, 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 with a node.js web app running on your local computer. After running the web app locally, you can optionally follow steps to host the web app in Azure App Service. 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

- Add a consumer group to your IoT hub that the web application will use to read sensor data
- Download the web app code from GitHub
- Examine the web app code
- Configure environment variables to hold the IoT Hub artifacts needed by your web app
- Run the web app on your development machine
- Open a web page to see real-time temperature and humidity data from your IoT hub
- (Optional) Use Azure CLI to host your web app in Azure App Service

What you need

- Complete the [Raspberry Pi online simulator](#) tutorial or one of the device tutorials; for example, [Raspberry Pi with node.js](#). These cover 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](#)
- The steps in this article assume a Windows development machine; however, you can easily perform these

steps on a Linux system in your preferred shell.

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|--|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal. |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the Copy button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

Run the following command to add the Microsoft Azure IoT Extension for Azure CLI to your Cloud Shell instance. The IOT Extension adds IoT Hub, IoT Edge, and IoT Device Provisioning Service (DPS) specific commands to Azure CLI.

```
az extension add --name azure-iot
```

Add a consumer group to your IoT hub

Consumer groups provide independent views into the event stream that enable apps and Azure services to independently consume data from the same Event Hub endpoint. In this section, you add a consumer group to your IoT hub's built-in endpoint that the web app will use to read data from.

Run the following command to add a consumer group to the built-in endpoint of your IoT hub:

```
az iot hub consumer-group create --hub-name YourIoTHubName --name YourConsumerGroupName
```

Note down the name you choose, you'll need it later in this tutorial.

Get a service connection string for your IoT hub

IoT hubs are created with several default access policies. One such policy is the **service** policy, which provides

sufficient permissions for a service to read and write the IoT hub's endpoints. Run the following command to get a connection string for your IoT hub that adheres to the service policy:

```
az iot hub show-connection-string --hub-name YourIoTHub --policy-name service
```

The connection string should look similar to the following:

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

Note down the service connection string, you'll need it later in this tutorial.

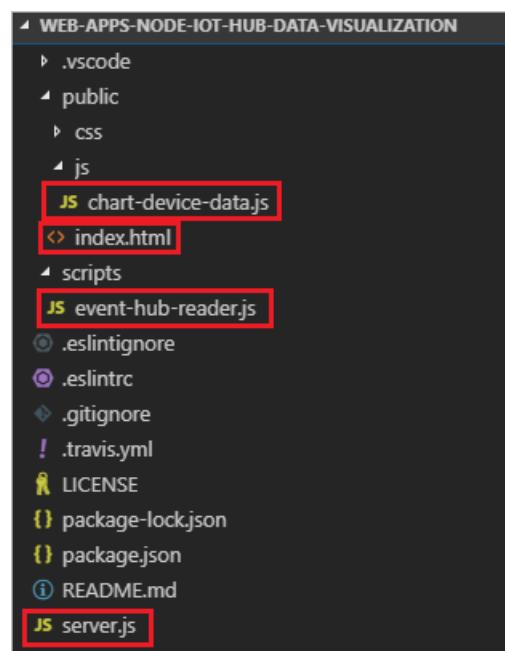
Download the web app from GitHub

Open a command window, and enter the following commands to download the sample from GitHub and change to the sample directory:

```
git clone https://github.com/Azure-Samples/web-apps-node-iot-hub-data-visualization.git  
cd web-apps-node-iot-hub-data-visualization
```

Examine the web app code

From the web-apps-node-iot-hub-data-visualization directory, open the web app in your favorite editor. The following shows the file structure viewed in VS Code:



Take a moment to examine the following files:

- **Server.js** is a service-side script that initializes the web socket and the Event Hub wrapper class. It provides a callback to the Event Hub wrapper class that the class uses to broadcast incoming messages to the web socket.
- **Event-hub-reader.js** is a service-side script that connects to the IoT hub's built-in endpoint using the specified connection string and consumer group. It extracts the DeviceId and EnqueuedTimeUtc from metadata on incoming messages and then relays the message using the callback method registered by server.js.

- `Chart-device-data.js` is a client-side script that listens on the web socket, keeps track of each DeviceId, and stores the last 50 points of incoming data for each device. It then binds the selected device data to the chart object.
- `Index.html` handles the UI layout for the web page and references the necessary scripts for client-side logic.

Configure environment variables for the web app

To read data from your IoT hub, the web app needs your IoT hub's connection string and the name of the consumer group that it should read through. It gets these strings from the process environment in the following lines in `server.js`:

```
const iotHubConnectionString = process.env.IotHubConnectionString;
const eventHubConsumerGroup = process.env.EventHubConsumerGroup;
```

Set the environment variables in your command window with the following commands. Replace the placeholder values with the service connection string for your IoT hub and the name of the consumer group you created previously. Don't quote the strings.

```
set IotHubConnectionString=YourIoTHubConnectionString
set EventHubConsumerGroup=YourConsumerGroupName
```

Run the web app

1. Make sure that your device is running and sending data.
2. In the command window, run the following lines to download and install referenced packages and start the website:

```
npm install
npm start
```

3. You should see output in the console that indicates that the web app has successfully connected to your IoT hub and is listening on port 3000:

```
C:\code3\web-apps-node-iot-hub-data-visualization>npm install
audited 823 packages in 1.513s
found 0 vulnerabilities

C:\code3\web-apps-node-iot-hub-data-visualization>npm start
> webapp@0.0.1 start C:\code3\web-apps-node-iot-hub-data-visualization
> node server.js

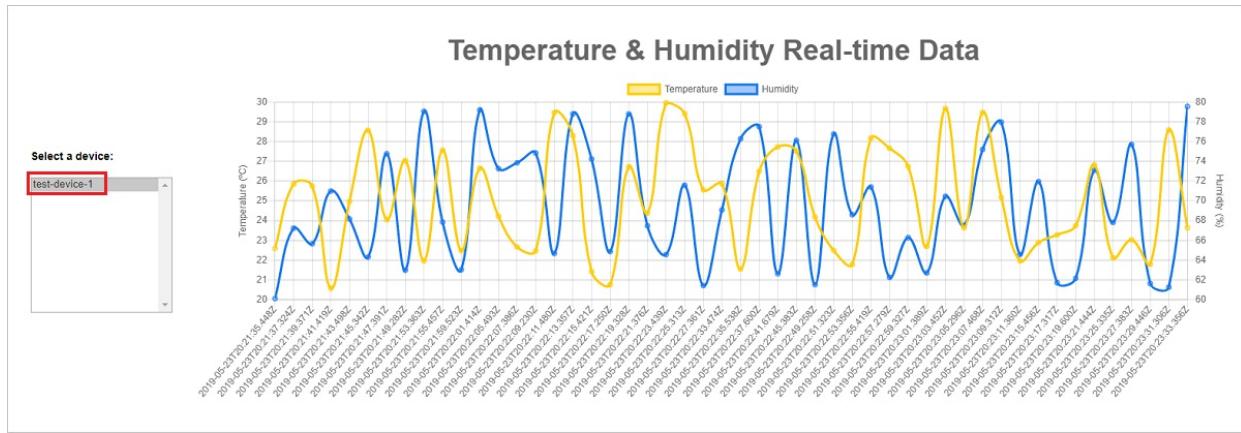
Listening on 3000.
Successfully created the EventHub Client from IoT Hub connection string.
The partition ids are: [ '0', '1', '2', '3' ]
```

Open a web page to see data from your IoT hub

Open a browser to <http://localhost:3000>.

In the **Select a device** list, select your device to see a running plot of the last 50 temperature and humidity data

points sent by the device to your IoT hub.



You should also see output in the console that shows the messages that your web app is broadcasting to the browser client:

```
C:\code3\web-apps-node-iot-hub-data-visualization>npm start
> webapp@0.0.1 start C:\code3\web-apps-node-iot-hub-data-visualization
> node server.js

Listening on 3000.
Successfully created the EventHub Client from IoT Hub connection string.
The partition ids are: [ '0', '1', '2', '3' ]
Broadcasting data {"TotData":{"messageId":926,"deviceId":"Raspberry Pi Web Client","temperature":20.45068803911039,"humidity":60.180029032336961,"MessageDate":"2019-05-23T20:19:07.496Z","DeviceId":"test-device-1"}
Broadcasting data {"TotData":{"messageId":927,"deviceId":"Raspberry Pi Web Client","temperature":20.901972115666005,"humidity":74.81110020178921,"MessageDate":"2019-05-23T20:19:09.543Z","DeviceId":"test-device-1"}
Broadcasting data {"TotData":{"messageId":928,"deviceId":"Raspberry Pi Web Client","temperature":26.38555844478955,"humidity":77.438785052053821,"MessageDate":"2019-05-23T20:19:11.388Z","DeviceId":"test-device-1"}
Broadcasting data {"TotData":{"messageId":929,"deviceId":"Raspberry Pi Web Client","temperature":22.61710143059446,"humidity":65.360859789382471,"MessageDate":"2019-05-23T20:19:13.249Z","DeviceId":"test-device-1"}
Broadcasting data {"TotData":{"messageId":930,"deviceId":"Raspberry Pi Web Client","temperature":27.06864202102564,"humidity":67.363973995619521,"MessageDate":"2019-05-23T20:19:15.311Z","DeviceId":"test-device-1"}
Broadcasting data {"TotData":{"messageId":931,"deviceId":"Raspberry Pi Web Client","temperature":28.809119733220534,"humidity":65.849480174461581,"MessageDate":"2019-05-23T20:19:17.358Z","DeviceId":"test-device-1"}
Broadcasting data {"TotData":{"messageId":932,"deviceId":"Raspberry Pi Web Client","temperature":25.814147212226988,"humidity":72.942902658042581,"MessageDate":"2019-05-23T20:19:19.421Z","DeviceId":"test-device-1"}
Broadcasting data {"TotData":{"messageId":934,"deviceId":"Raspberry Pi Web Client","temperature":24.41547756986796,"humidity":75.883074850617831,"MessageDate":"2019-05-23T20:19:23.313Z","DeviceId":"test-device-1"}
```

Host the web app in App Service

The [Web Apps feature of Azure App Service](#) provides a platform as a service (PAAS) for hosting web applications. Web applications hosted in Azure App Service can benefit from powerful Azure features like additional security, load balancing, and scalability as well as Azure and partner DevOps solutions like continuous deployment, package management, and so on. Azure App Service supports web applications developed in many popular languages and deployed on Windows or Linux infrastructure.

In this section, you provision a web app in App Service and deploy your code to it by using Azure CLI commands. You can find details of the commands used in the [az webapp](#) documentation. Before starting, make sure you've completed the steps to [add a resource group to your IoT hub](#), [get a service connection string for your IoT hub](#), and [download the web app from GitHub](#).

1. An [App Service plan](#) defines a set of compute resources for an app hosted in App Service to run. In this tutorial, we use the Developer/Free tier to host the web app. With the Free tier, your web app runs on shared Windows resources with other App Service apps, including apps of other customers. Azure also offers App Service plans to deploy web apps on Linux compute resources. You can skip this step if you already have an App Service plan that you want to use.

To create an App Service plan using the Windows free tier, run the following command. Use the same resource group your IoT hub is in. Your service plan name can contain upper and lower case letters, numbers, and hyphens.

```
az appservice plan create --name <app service plan name> --resource-group <your resource group name>  
--sku FREE
```

- Now provision a web app in your App Service plan. The `--deployment-local-git` parameter enables the web app code to be uploaded and deployed from a Git repository on your local machine. Your web app name must be globally unique and can contain upper and lower case letters, numbers, and hyphens. Be sure to specify Node version 10.6 or later for the `--runtime` parameter, depending on the version of the Node.js runtime you are using. You can use the `az webapp list-runtimes` command to get a list of supported runtimes.

```
az webapp create -n <your web app name> -g <your resource group name> -p <your app service plan name>  
--runtime "node|10.6" --deployment-local-git
```

- Now add Application Settings for the environment variables that specify the IoT hub connection string and the Event hub consumer group. Individual settings are space delimited in the `-settings` parameter. Use the service connection string for your IoT hub and the consumer group you created previously in this tutorial. Don't quote the values.

```
az webapp config appsettings set -n <your web app name> -g <your resource group name> --settings  
EventHubConsumerGroup=<your consumer group> IoTHubConnectionString=<your IoT hub connection string>
```

- Enable the Web Sockets protocol for the web app and set the web app to receive HTTPS requests only (HTTP requests are redirected to HTTPS).

```
az webapp config set -n <your web app name> -g <your resource group name> --web-sockets-enabled true  
az webapp update -n <your web app name> -g <your resource group name> --https-only true
```

- To deploy the code to App Service, you'll use your [user-level deployment credentials](#). Your user-level deployment credentials are different from your Azure credentials and are used for Git local and FTP deployments to a web app. Once set, they're valid across all of your App Service apps in all subscriptions in your Azure account. If you've previously set user-level deployment credentials, you can use them.

If you haven't previously set user-level deployment credentials or you can't remember your password, run the following command. Your deployment user name must be unique within Azure, and it must not contain the '@' symbol for local Git pushes. When you're prompted, enter and confirm your new password. The password must be at least eight characters long, with two of the following three elements: letters, numbers, and symbols.

```
az webapp deployment user set --user-name <your deployment user name>
```

- Get the Git URL to use to push your code up to App Service.

```
az webapp deployment source config-local-git -n <your web app name> -g <your resource group name>
```

- Add a remote to your clone that references the Git repository for the web app in App Service. For <Git clone URL>, use the URL returned in the previous step. Run the following command in your command window.

```
git remote add webapp <Git clone URL>
```

8. To deploy the code to App Service, enter the following command in your command window. If you are prompted for credentials, enter the user-level deployment credentials that you created in step 5. Make sure that you push to the master branch of the App Service remote.

```
git push webapp master:master
```

9. The progress of the deployment will update in your command window. A successful deployment will end with lines similar to the following output:

```
remote:  
remote: Finished successfully.  
remote: Running post deployment command(s)...  
remote: Deployment successful.  
To https://contoso-web-app-3.scm.azurewebsites.net/contoso-web-app-3.git  
 6b132dd..7cbc994  master -> master
```

10. Run the following command to query the state of your web app and make sure it is running:

```
az webapp show -n <your web app name> -g <your resource group name> --query state
```

11. Navigate to <https://<your web app name>.azurewebsites.net> in a browser. A web page similar to the one you saw when you ran the web app locally displays. Assuming that your device is running and sending data, you should see a running plot of the 50 most recent temperature and humidity readings sent by the device.

Troubleshooting

If you come across any issues with this sample, try the steps in the following sections. If you still have problems, send us feedback at the bottom of this topic.

Client issues

- If a device does not appear in the list, or no graph is being drawn, make sure the device code is running on your device.
- In the browser, open the developer tools (in many browsers the F12 key will open it), and find the console. Look for any warnings or errors printed there.
- You can debug client-side script in /js/chat-device-data.js.

Local website issues

- Watch the output in the window where you launched node for console output.
- Debug the server code, specifically server.js and /scripts/event-hub-reader.js.

Azure App Service issues

- In Azure portal, go to your web app. Under **Monitoring** in the left pane, select **App Service logs**. Turn **Application Logging (File System)** to on, set **Level** to Error, and then select **Save**. Then open **Log stream** (under **Monitoring**).
- From your web app in Azure portal, under **Development Tools** select **Console** and validate node and npm versions with `node -v` and `npm -v`.
- If you see an error about not finding a package, you may have run the steps out of order. When the site is deployed (with `git push`) the app service runs `npm install`, which runs based on the current version of node it has configured. If that is changed in configuration later, you'll need to make a meaningless change

to the code and push again.

Next steps

You've successfully used your web app to visualize real-time sensor data from your IoT hub.

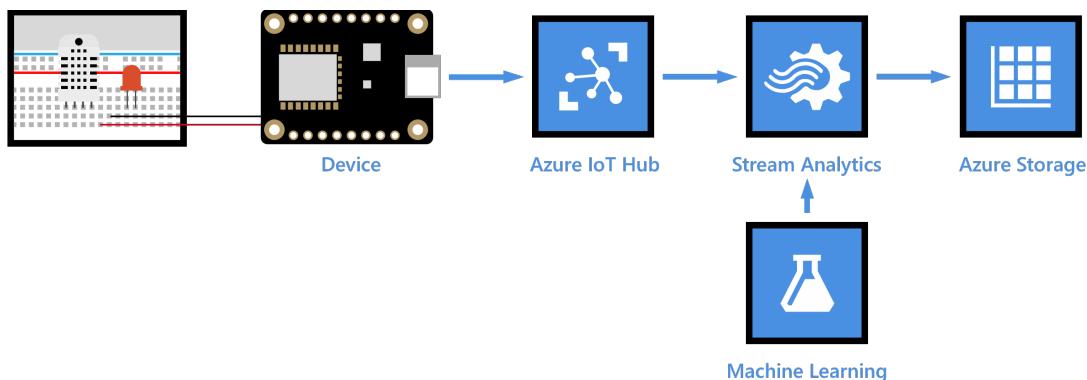
For another way to visualize data from Azure IoT Hub, see [Use Power BI to visualize real-time sensor data from your IoT hub](#).

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 extension for Visual Studio Code](#)
- [Manage devices with Azure IoT Hub extension for Visual Studio Code](#)
- [Set up message routing](#)
- [Use Power BI to visualize real-time sensor data from your IoT hub](#)
- [Use a web app 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

5/21/2020 • 7 minutes to read • [Edit Online](#)



NOTE

Before you start this tutorial, complete the [Raspberry Pi online simulator](#) tutorial or one of the device tutorials; for example, [Raspberry Pi with node.js](#). In these articles, 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

- Complete the [Raspberry Pi online simulator](#) tutorial or one of the device tutorials; for example, [Raspberry Pi with node.js](#). These cover 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 \(classic\)](#) account.

Deploy the weather prediction model as a web service

In this section you get the weather prediction model from the Azure AI Library. Then you add an R-script module to the model to clean the temperature and humidity data. Lastly, you deploy the model as a predictive web service.

Get the weather prediction model

In this section you get the weather prediction model from the Azure AI Gallery and open it in Azure Machine Learning Studio (classic).

1. Go to the [weather prediction model page](#).

The screenshot shows the Azure AI Gallery interface. A specific experiment card for a 'Weather prediction model' is displayed. The card includes a profile picture, the experiment name, the author's name (Yuwei Zhou), the date (March 3, 2017), and a '27 likes' count. Below the card are two buttons: 'Summary' and 'Description'. To the right of the card is a preview image of a gauge with three faces: green, yellow, and red. At the bottom right of the card is a large green button with white text that reads 'Open in Studio (classic)'. This button is highlighted with a red rectangular border.

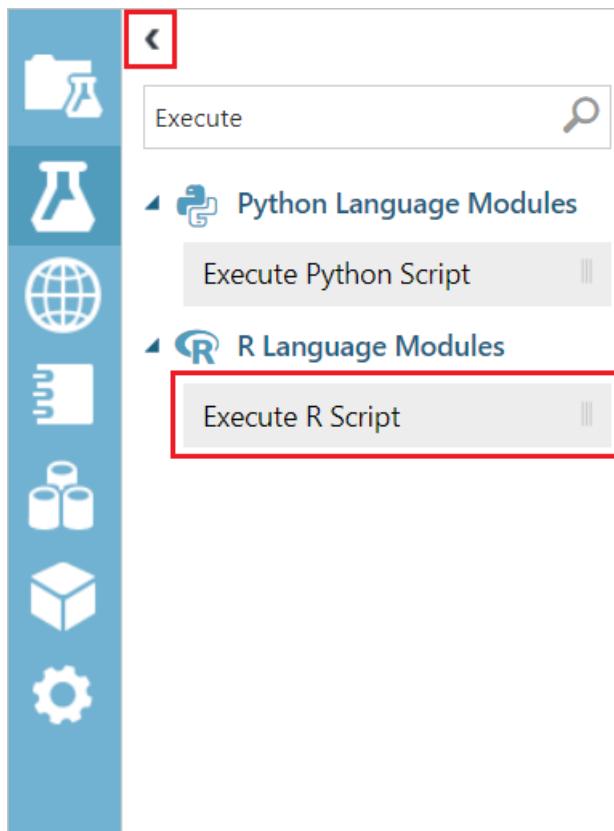
2. Click Open in Studio (classic) to open the model in Microsoft Azure Machine Learning Studio (classic).

The screenshot shows the Microsoft Azure Machine Learning Studio (classic) interface. On the left, there is a vertical toolbar with icons for Datasets, Modules, Trained Models, and Transforms. The main workspace displays the 'Weather prediction model' experiment. The experiment consists of several connected modules: 'Two-Class Logistic Regression', 'Split Data', 'Train Model', 'Score Model', and 'Web service output'. A 'Mini Map' panel on the left shows a detailed view of the entire experiment flow. On the right side of the screen, there are several panels: 'Properties' (Experiment Properties, Status: InDraft), 'Summary' (empty text area), 'Description' (empty text area), and 'Quick Help'. At the bottom of the screen is a navigation bar with buttons for 'NEW', 'RUN HISTORY', 'SAVE', 'DISCARD CHANGES', 'RUN', 'SET UP WEB SERVICE', and 'PUBLISH TO GALLERY'.

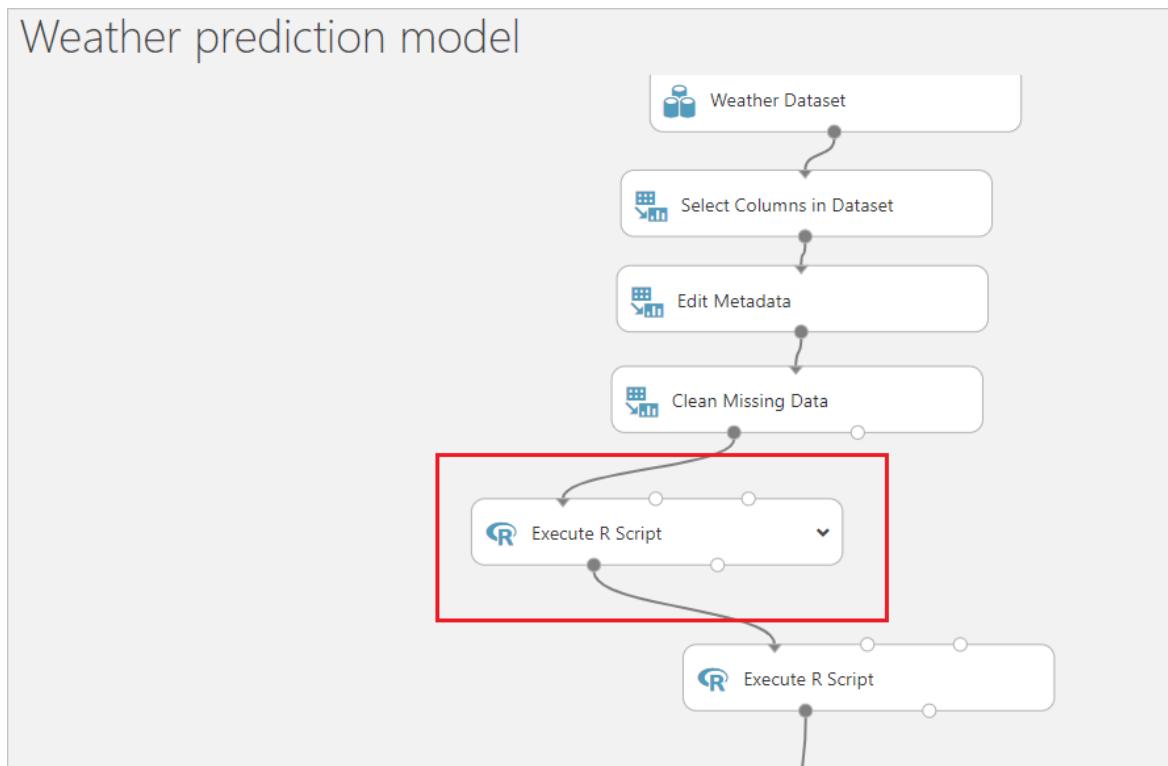
Add an R-script module to clean temperature and humidity data

For the model to behave correctly, the temperature and humidity data must be convertible to numeric data. In this section, you add an R-script module to the weather prediction model that removes any rows that have data values for temperature or humidity that cannot be converted to numeric values.

1. On the left-side of the Azure Machine Learning Studio window, click the arrow to expand the tools panel. Enter "Execute" into the search box. Select the **Execute R Script** module.



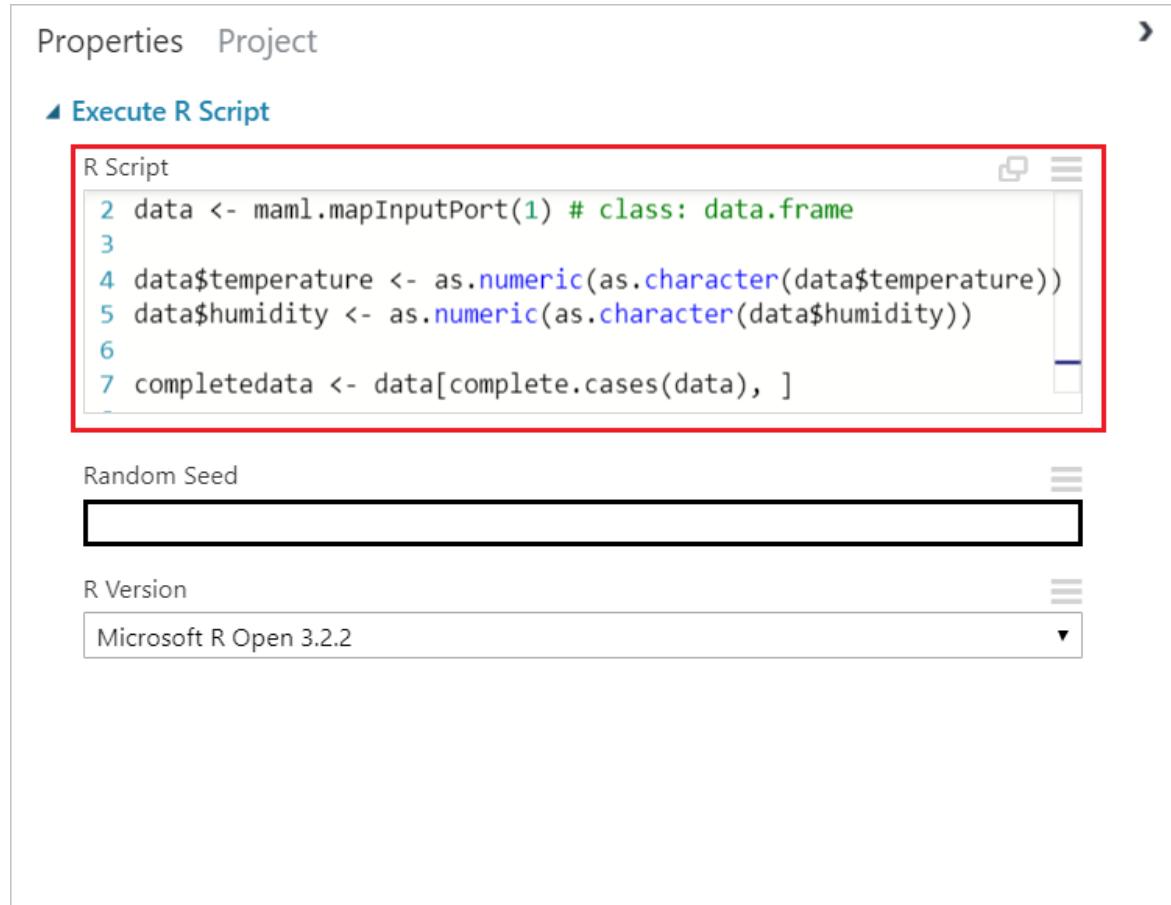
2. Drag the **Execute R Script** module near the **Clean Missing Data** module and the existing **Execute R Script** module on the diagram. Delete the connection between the **Clean Missing Data** and the **Execute R Script** modules and then connect the inputs and outputs of the new module as shown.



3. Select the new **Execute R Script** module to open its properties window. Copy and paste the following code into the **R Script** box.

```
# Map 1-based optional input ports to variables  
data <- maml.mapInputPort(1) # class: data.frame  
  
data$temperature <- as.numeric(as.character(data$temperature))  
data$humidity <- as.numeric(as.character(data$humidity))  
  
completedata <- data[complete.cases(data), ]  
  
maml.mapOutputPort('completedata')
```

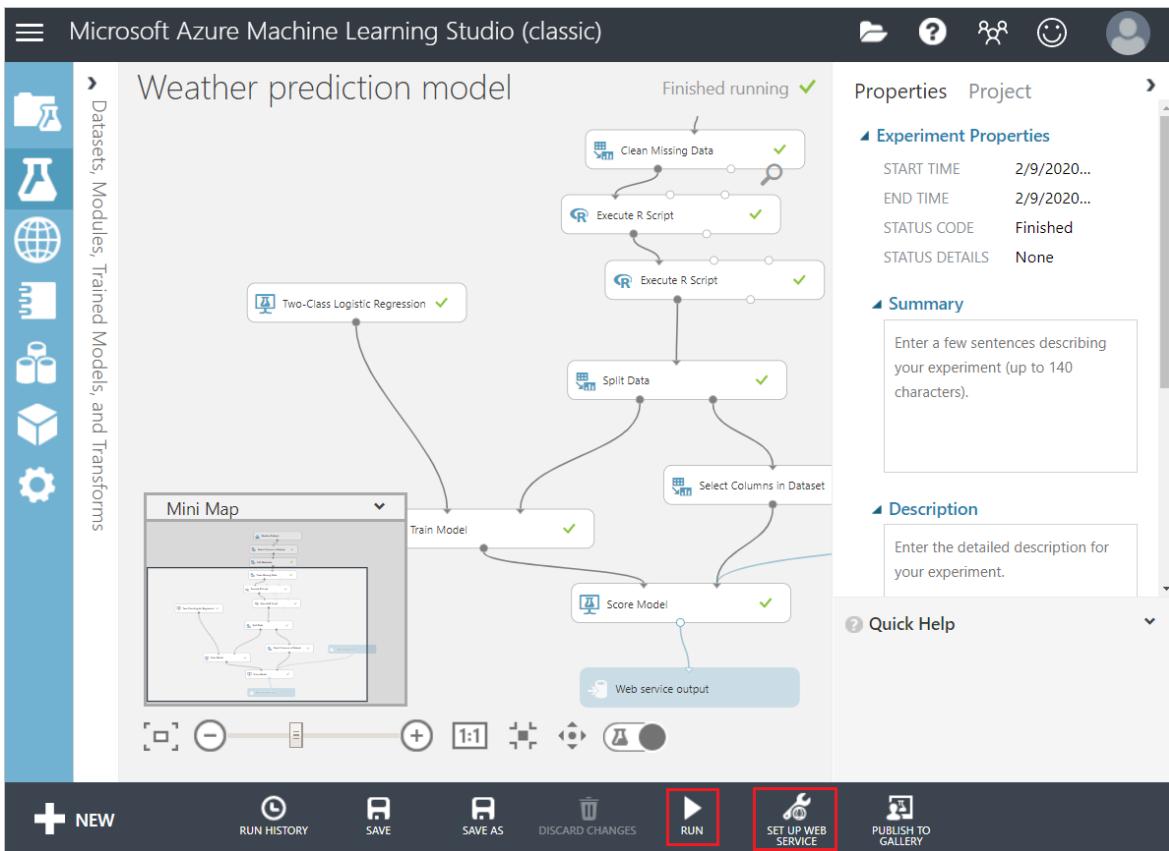
When you're finished, the properties window should look similar to the following:



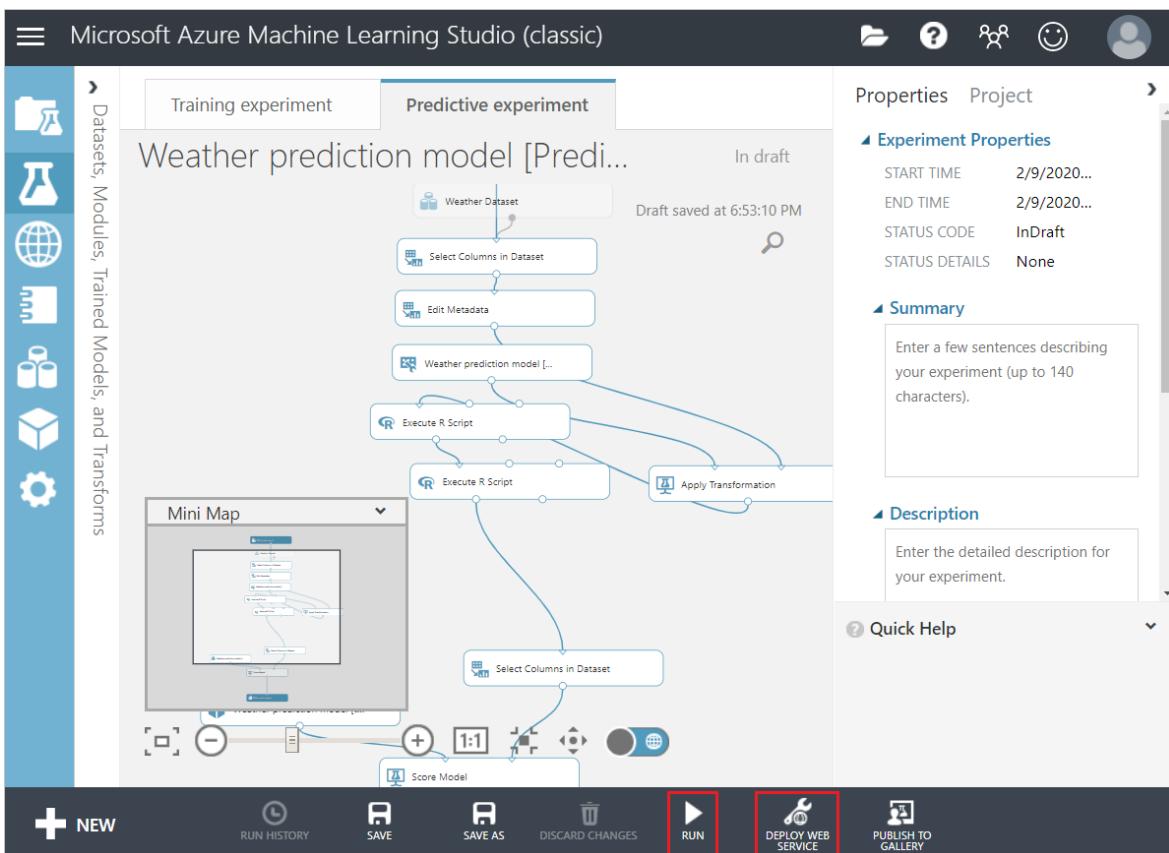
Deploy predictive web service

In this section, you validate the model, set up a predictive web service based on the model, and then deploy the web service.

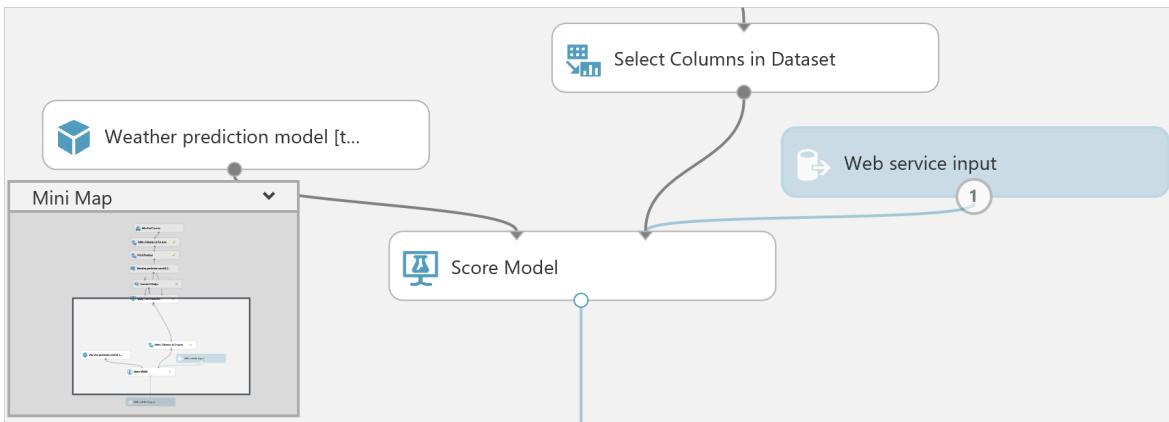
1. Click **Run** to validate the steps in the model. This step might take a few minutes to complete.



2. Click **SET UP WEB SERVICE** > **Predictive Web Service**. The predictive experiment diagram opens.



3. In the predictive experiment diagram, delete the connection between the **Web service input** module and the **Weather Dataset** at the top. Then drag the **Web service input** module somewhere near the **Score Model** module and connect it as shown:



4. Click **RUN** to validate the steps in the model.
5. Click **DEPLOY WEB SERVICE** to deploy the model as a web service.
6. On the dashboard of the model, download the **Excel 2010 or earlier** workbook for **REQUEST/RESPONSE**.

NOTE

Make sure that you download the **Excel 2010 or earlier** workbook even if you are running a later version of Excel on your computer.

| REQUEST/RESPONSE | Test | Excel 2013 or later | Excel 2010 or earlier | LAST UPDATED |
|------------------|------|---------------------|-----------------------|---------------------|
| BATCH EXECUTION | Test | Excel 2013 or later | Excel 2010 or earlier | 2/9/2020 7:17:06 PM |

7. 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](#) provide independent views into the event stream that enable apps and Azure services to independently consume data from the same Event Hub endpoint. In this section, you add a consumer group to your IoT hub's built-in endpoint that is used later in this tutorial to pull data from the endpoint.

To add a consumer group to your IoT hub, follow these steps:

1. In the [Azure portal](#), open your IoT hub.
2. On the left pane, select **Built-in endpoints**, select **Events** on the right pane, and enter a name under **Consumer groups**. Select **Save**.

The screenshot shows the Azure IoT Hub interface for 'contoso-hub-1'. The left sidebar has 'Built-in endpoints' selected. The main pane shows the 'Events' configuration. It includes fields for 'Event Hub-compatible name' (set to 'contoso-hub-1'), 'Event Hub-compatible endpoint' (set to 'Endpoint=sb://iothub-ns-contoso-hu-1477762-53f6652df9.servicebus.windows.net/SharedAccessKeyName=iothubowner;SharedAccessKe...'), 'Partitions' (set to 4), 'Retain for' (set to 1 Day), and a 'Create new consumer group' input field which is highlighted with a red box. Below this is the 'Cloud to device messaging' section with settings for 'Default TTL', 'Feedback retention time', and 'Maximum delivery count' (set to 10 attempts).

Create, configure, and run a Stream Analytics job

Create a Stream Analytics job

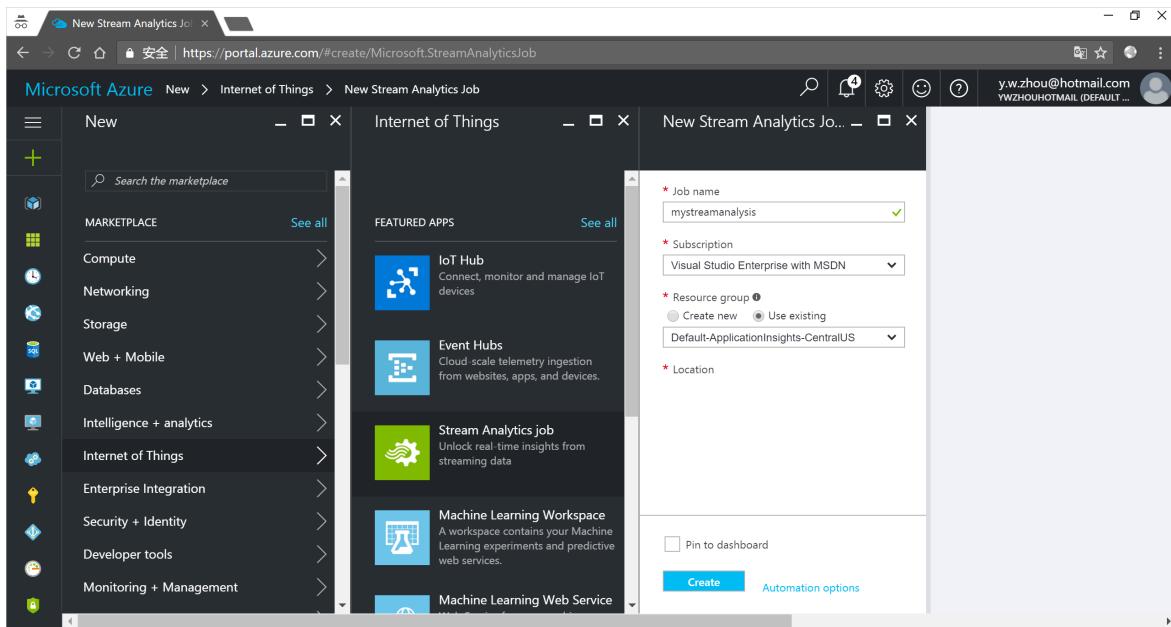
1. In the [Azure portal](#), click **Create a resource** > **Internet of Things** > **Stream Analytics job**.
2. Enter the following information for the job.

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.



3. Click **Create**.

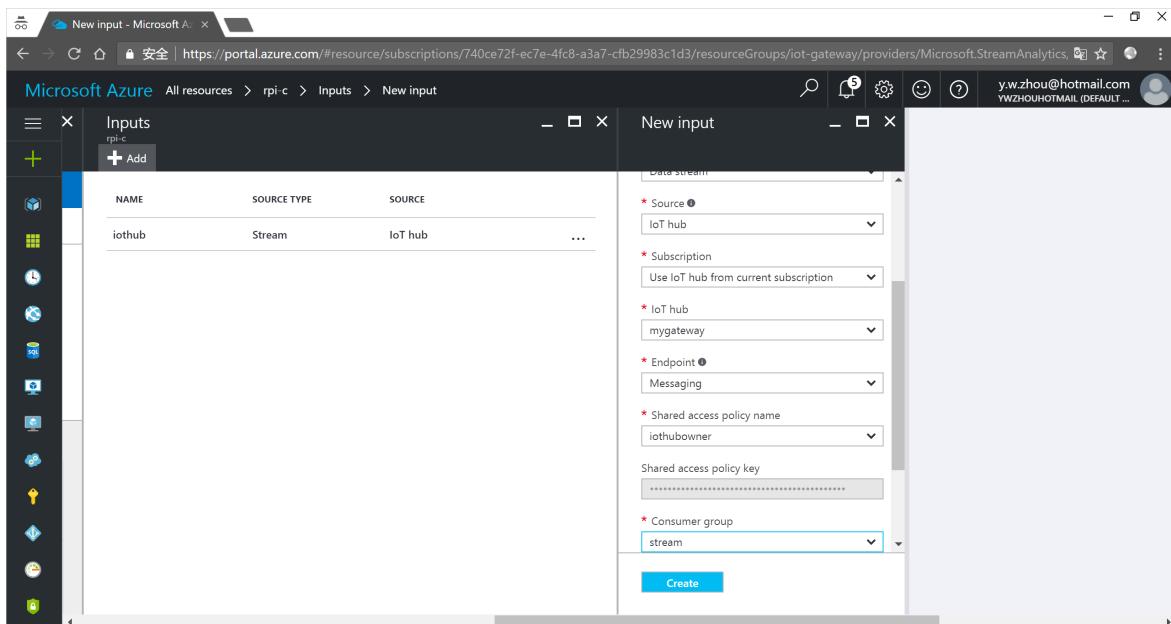
Add an input to the Stream Analytics job

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.



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:

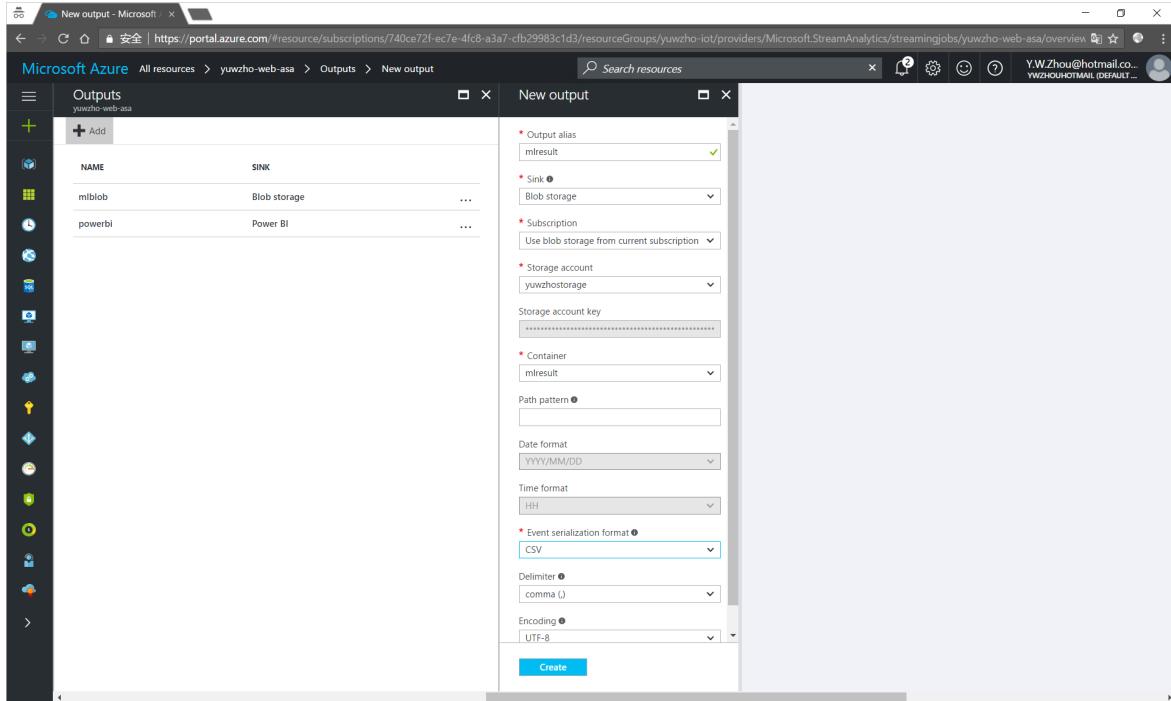
Output alias: The unique alias for the output.

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:

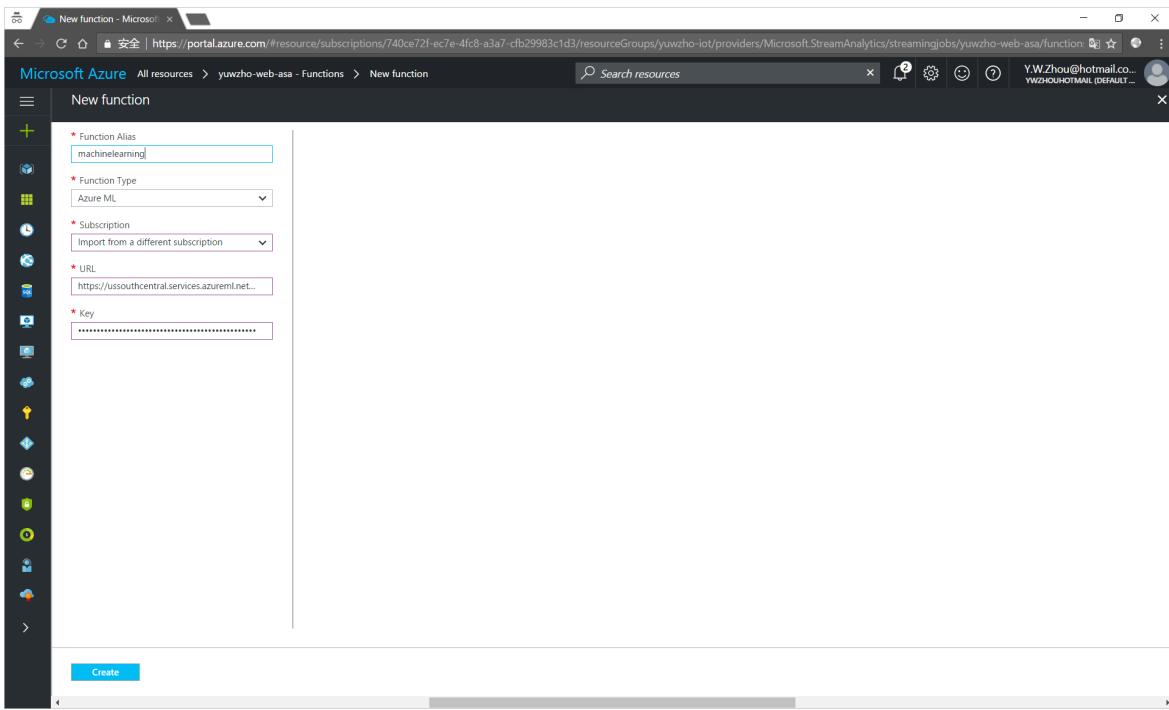
Function Alias: Enter `machinelearning`.

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
```

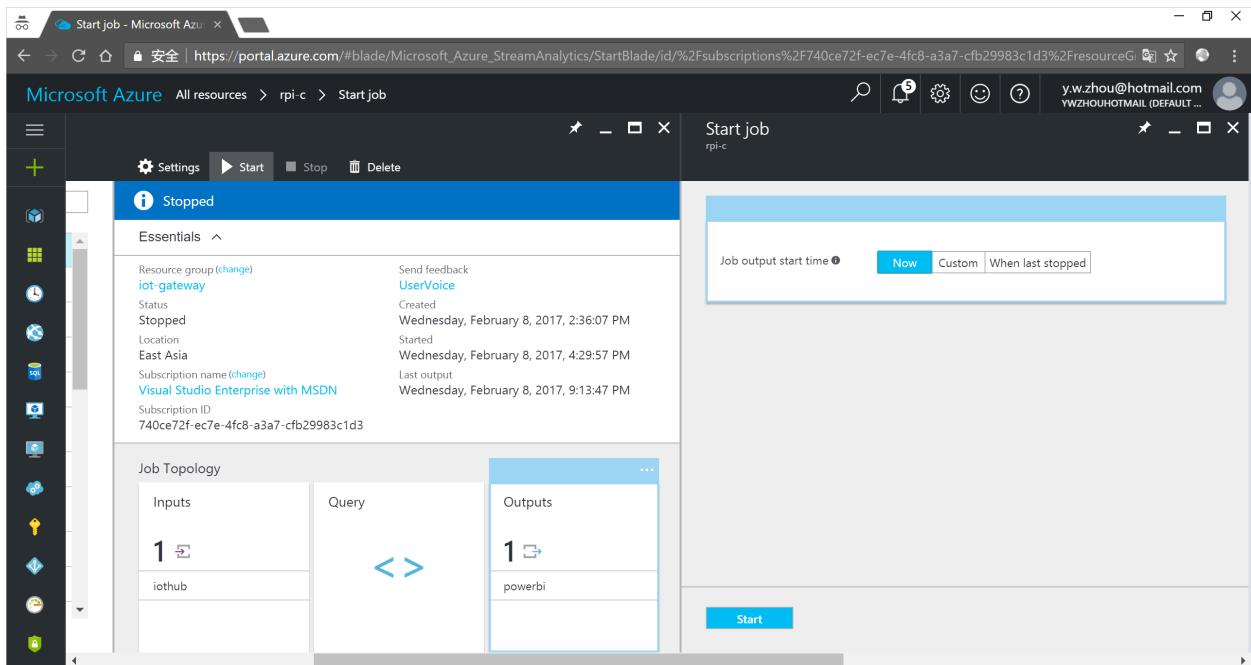
Replace `[YourInputAlias]` with the input alias of the job.

Replace `[YourOutputAlias]` with the output alias of the job.

3. Click **Save**.

Run the Stream Analytics job

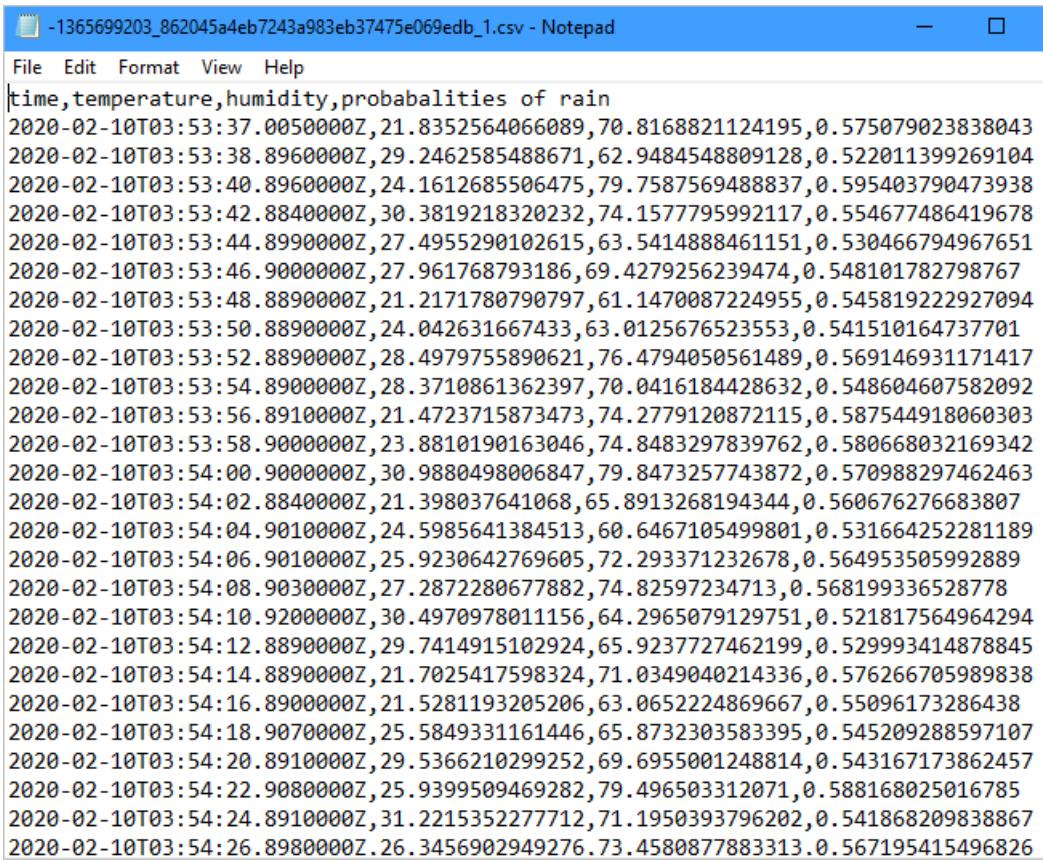
In the Stream Analytics job, click **Start > Now > Start**. Once the job successfully starts, the job status changes from **Stopped** to **Running**.



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. Download a .csv file to see the result. The last column records the chance of rain.



The screenshot shows a Notepad window with a CSV file titled '-1365699203_862045a4eb7243a983eb37475e069edb_1.csv'. The file contains a header row 'time,temperature,humidity,probabalities of rain' followed by approximately 40 data rows. Each row represents a timestamp and three sensor values: temperature, humidity, and probability of rain.

| time | temperature | humidity | probabalities of rain |
|------------------------------|------------------|------------------|-----------------------|
| 2020-02-10T03:53:37.0050000Z | 21.8352564066089 | 70.8168821124195 | 0.575079023838043 |
| 2020-02-10T03:53:38.8960000Z | 29.2462585488671 | 62.9484548809128 | 0.522011399269104 |
| 2020-02-10T03:53:40.8960000Z | 24.1612685506475 | 79.7587569488837 | 0.595403790473938 |
| 2020-02-10T03:53:42.8840000Z | 30.3819218320232 | 74.1577795992117 | 0.554677486419678 |
| 2020-02-10T03:53:44.8990000Z | 27.4955290102615 | 63.5414888461151 | 0.530466794967651 |
| 2020-02-10T03:53:46.9000000Z | 27.961768793186 | 69.4279256239474 | 0.548101782798767 |
| 2020-02-10T03:53:48.8890000Z | 21.2171780790797 | 61.1470087224955 | 0.545819222927094 |
| 2020-02-10T03:53:50.8890000Z | 24.042631667433 | 63.0125676523553 | 0.541510164737701 |
| 2020-02-10T03:53:52.8890000Z | 28.4979755890621 | 76.4794050561489 | 0.569146931171417 |
| 2020-02-10T03:53:54.8900000Z | 28.3710861362397 | 70.0416184428632 | 0.548604607582092 |
| 2020-02-10T03:53:56.8910000Z | 21.4723715873473 | 74.2779120872115 | 0.587544918060303 |
| 2020-02-10T03:53:58.9000000Z | 23.8810190163046 | 74.8483297839762 | 0.580668032169342 |
| 2020-02-10T03:54:00.9000000Z | 30.9880498006847 | 79.8473257743872 | 0.570988297462463 |
| 2020-02-10T03:54:02.8840000Z | 21.398037641068 | 65.8913268194344 | 0.560676276683807 |
| 2020-02-10T03:54:04.9010000Z | 24.5985641384513 | 60.6467105499801 | 0.531664252281189 |
| 2020-02-10T03:54:06.9010000Z | 25.9230642769605 | 72.293371232678 | 0.564953505992889 |
| 2020-02-10T03:54:08.9030000Z | 27.2872280677882 | 74.82597234713 | 0.568199336528778 |
| 2020-02-10T03:54:10.9200000Z | 30.4970978011156 | 64.2965079129751 | 0.521817564964294 |
| 2020-02-10T03:54:12.8890000Z | 29.7414915102924 | 65.9237727462199 | 0.529993414878845 |
| 2020-02-10T03:54:14.8890000Z | 21.7025417598324 | 71.0349040214336 | 0.576266705989838 |
| 2020-02-10T03:54:16.8900000Z | 21.5281193205206 | 63.0652224869667 | 0.55096173286438 |
| 2020-02-10T03:54:18.9070000Z | 25.5849331161446 | 65.8732303583395 | 0.545209288597107 |
| 2020-02-10T03:54:20.8910000Z | 29.5366210299252 | 69.6955001248814 | 0.543167173862457 |
| 2020-02-10T03:54:22.9080000Z | 25.9399509469282 | 79.496503312071 | 0.588168025016785 |
| 2020-02-10T03:54:24.8910000Z | 31.2215352277712 | 71.1950393796202 | 0.541868209838867 |
| 2020-02-10T03:54:26.8980000Z | 26.3456902949276 | 73.4580877883313 | 0.567195415496826 |

Summary

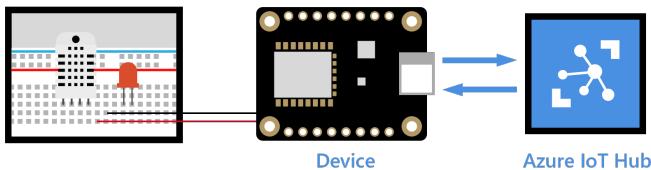
You've successfully used Azure Machine Learning to produce the chance of rain based on the temperature and humidity data that your IoT hub receives.

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 extension for Visual Studio Code](#)
- [Manage devices with Azure IoT Hub extension for Visual Studio Code](#)
- [Set up message routing](#)
- [Use Power BI to visualize real-time sensor data from your IoT hub](#)
- [Use a web app 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

4/20/2020 • 3 minutes to read • [Edit Online](#)



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 available only in the standard tier of IoT Hub. For more information about the basic and standard/free IoT Hub tiers, see [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. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module 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](#) or copy this URL and paste it into a browser window:
`vscode:extension/vsciot-vscode.azure-iot-tools`

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 `iothubowner` policy 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** 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** view.

Next steps

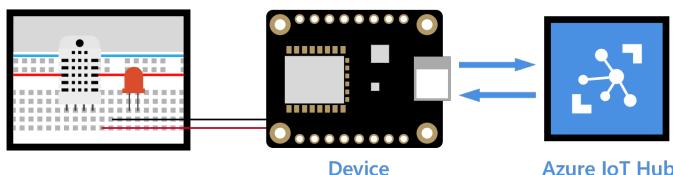
You've learned how to use Azure IoT Tools extension for Visual Studio Code with various management options.

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 extension for Visual Studio Code](#)
- [Manage devices with Azure IoT Hub extension for Visual Studio Code](#)
- [Set up message routing](#)
- [Use Power BI to visualize real-time sensor data from your IoT hub](#)
- [Use a web app 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

11/12/2019 • 3 minutes to read • [Edit Online](#)



[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 available only in the standard tier of IoT Hub. For more information about the basic and standard/free IoT Hub tiers, see [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, including 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

In this article, you learn how to use the Cloud Explorer for Visual Studio with various management options on your development computer.

What you do

In this article, run Cloud Explorer for Visual Studio with various management options.

What you need

You need the following prerequisites:

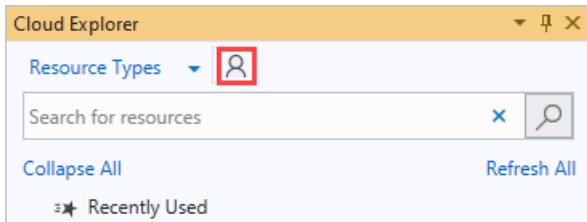
- An active Azure subscription.
- An Azure IoT Hub under your subscription.
- Microsoft Visual Studio 2017 Update 9 or later. This article uses [Visual Studio 2017 or Visual Studio 2019](#).
- Cloud Explorer component from Visual Studio Installer, which selected by default with Azure Workload.

Update Cloud Explorer to latest version

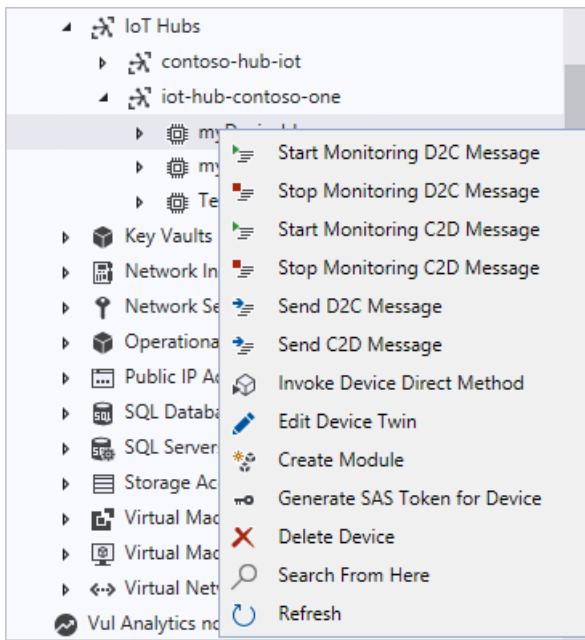
The Cloud Explorer component from Visual Studio Installer for Visual Studio 2017 only supports monitoring device-to-cloud and cloud-to-device messages. To use Visual Studio 2017, download and install the latest [Cloud Explorer](#).

Sign in to access your hub

1. In Visual Studio, select **View > Cloud Explorer** to open Cloud Explorer.
2. Select the Account Management icon to show your subscriptions.



3. If you are signed in to Azure, your accounts appear. To sign into Azure for the first time, choose **Add an account**.
4. Select the Azure subscriptions you want to use and choose **Apply**.
5. Expand your subscription, then expand **IoT Hubs**. Under each hub, you can see your devices for that hub. Right-click one device to access the management options.



Direct methods

To use direct methods, do the following steps:

1. Right-click your device and select **Invoke Device Direct Method**.
2. Enter the method name and payload in **Invoke Direct Method**, and then select **OK**.

Results appear in **Output**.

Update device twin

To edit a device twin, do the following steps:

1. Right-click your device and select **Edit Device Twin**.

An **azure-iot-device-twin.json** file opens with the content of device twin.

2. Make some edits of **tags** or **properties.desired** fields to the **azure-iot-device-twin.json** file.
3. Press **Ctrl+S** to update the device twin.

Results appear in **Output**.

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 **Send C2D message** and select **OK**.

Results appear in **Output**.

Next steps

You've learned how to use Cloud Explorer for Visual Studio with various management options.

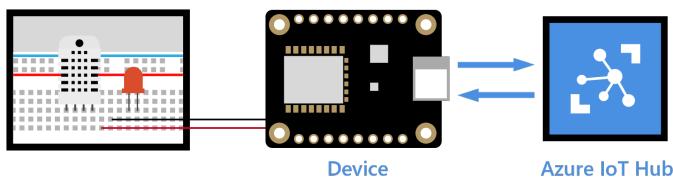
To continue to get started with Azure IoT Hub and to explore all extended IoT scenarios, see the following:

- [Manage cloud device messaging with Azure IoT Hub extension for Visual Studio Code](#)

- Manage devices with Azure IoT Hub extension for Visual Studio Code
- Set up message routing
- Use Power BI to visualize real-time sensor data from your IoT hub
- Use a web app 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

3/25/2020 • 5 minutes to read • [Edit Online](#)



NOTE

Before you start this tutorial, complete the [Raspberry Pi online simulator](#) tutorial or one of the device tutorials; for example, [Raspberry Pi with node.js](#). In these articles, 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 an 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

This article uses the newest version of the Azure IoT extension, called `azure-iot`. The legacy version is called `azure-cli-iot-ext`. You should only have one version installed at a time. You can use the command `az extension list` to validate the currently installed extensions.

Use `az extension remove --name azure-cli-iot-ext` to remove the legacy version of the extension.

Use `az extension add --name azure-iot` to add the new version of the extension.

To see what extensions you have installed, use `az extension list`.

NOTE

The features described in this article are available only in the standard tier of IoT Hub. For more information about the basic and standard/free IoT Hub tiers, see [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. |

| MANAGEMENT OPTION | TASK |
|--------------------------|---|
| 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 twins 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 to use 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 [Raspberry Pi online simulator](#) tutorial or one of the device tutorials; for example, [Raspberry Pi with node.js](#). These items cover 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 2.0.70 or above. Use `az --version` to validate.

NOTE

This article uses the newest version of the Azure IoT extension, called `azure-iot`. The legacy version is called `azure-cli-iot-ext`. You should only have one version installed at a time. You can use the command `az extension list` to validate the currently installed extensions.

Use `az extension remove --name azure-cli-iot-ext` to remove the legacy version of the extension.

Use `az extension add --name azure-iot` to add the new version of the extension.

To see what extensions you have installed, use `az extension list`.

- Install the IoT extension. The simplest way is to run `az extension add --name azure-iot`. [The IoT extension readme](#) describes several ways to install the extension.

Sign in to your Azure account

Sign 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

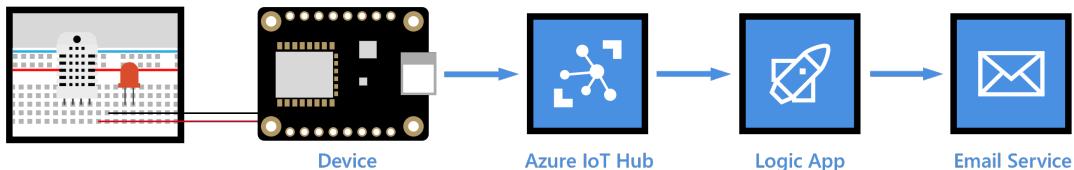
You've learned how to monitor device-to-cloud messages and send cloud-to-device messages between your IoT device and Azure IoT Hub.

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 extension for Visual Studio Code](#)
- [Manage devices with Azure IoT Hub extension for Visual Studio Code](#)
- [Set up message routing](#)
- [Use Power BI to visualize real-time sensor data from your IoT hub](#)
- [Use a web app 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

4/20/2020 • 8 minutes to read • [Edit Online](#)



NOTE

Before you start this tutorial, complete the [Raspberry Pi online simulator](#) tutorial or one of the device tutorials; for example, [Raspberry Pi with node.js](#). In these articles, 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 can help you orchestrate workflows across on-premises and cloud services, one or more enterprises, and across various protocols. A logic app begins with a trigger, which is then followed by one or more actions that can be sequenced using built-in controls, such as conditions and iterators. This flexibility makes Logic Apps an ideal IoT solution for IoT monitoring scenarios. For example, the arrival of telemetry data from a device at an IoT Hub endpoint can initiate logic app workflows to warehouse the data in an Azure Storage blob, send email alerts to warn of data anomalies, schedule a technician visit if a device reports a failure, and so on.

What you learn

You learn how to create a logic app that connects your IoT hub and your mailbox for temperature monitoring and notifications.

The client code running on your device sets an application property, `temperatureAlert`, on every telemetry message it sends to your IoT hub. When the client code detects a temperature above 30 C, it sets this property to `true`; otherwise, it sets the property to `false`.

Messages arriving at your IoT hub look similar to the following, with the telemetry data contained in the body and the `temperatureAlert` property contained in the application properties (system properties are not shown):

```
{
  "body": {
    "messageId": 18,
    "deviceId": "Raspberry Pi Web Client",
    "temperature": 27.796111770668457,
    "humidity": 66.77637926438427
  },
  "applicationProperties": {
    "temperatureAlert": "false"
  }
}
```

To learn more about IoT Hub message format, see [Create and read IoT Hub messages](#).

In this topic, you set up routing on your IoT hub to send messages in which the `temperatureAlert` property is `true` to a Service Bus endpoint. You then set up a logic app that triggers on the messages arriving at the Service Bus endpoint and sends you an email notification.

What you do

- Create a Service Bus namespace and add a Service Bus queue to it.
- Add a custom endpoint and a routing rule to your IoT hub to route messages that contain a temperature alert to the Service Bus queue.
- Create, configure, and test a logic app to consume messages from your Service Bus queue and send notification emails to a desired recipient.

What you need

- Complete the [Raspberry Pi online simulator](#) tutorial or one of the device tutorials; for example, [Raspberry Pi with node.js](#). These cover the following requirements:
 - An active Azure subscription.
 - An Azure IoT hub under your subscription.
 - A client application running on your device that sends telemetry messages to your Azure IoT hub.

Create Service Bus namespace and queue

Create a Service Bus namespace and queue. Later in this topic, you create a routing rule in your IoT hub to direct messages that contain a temperature alert to the Service Bus queue, where they will be picked up by a logic app and trigger it to send a notification email.

Create a Service Bus namespace

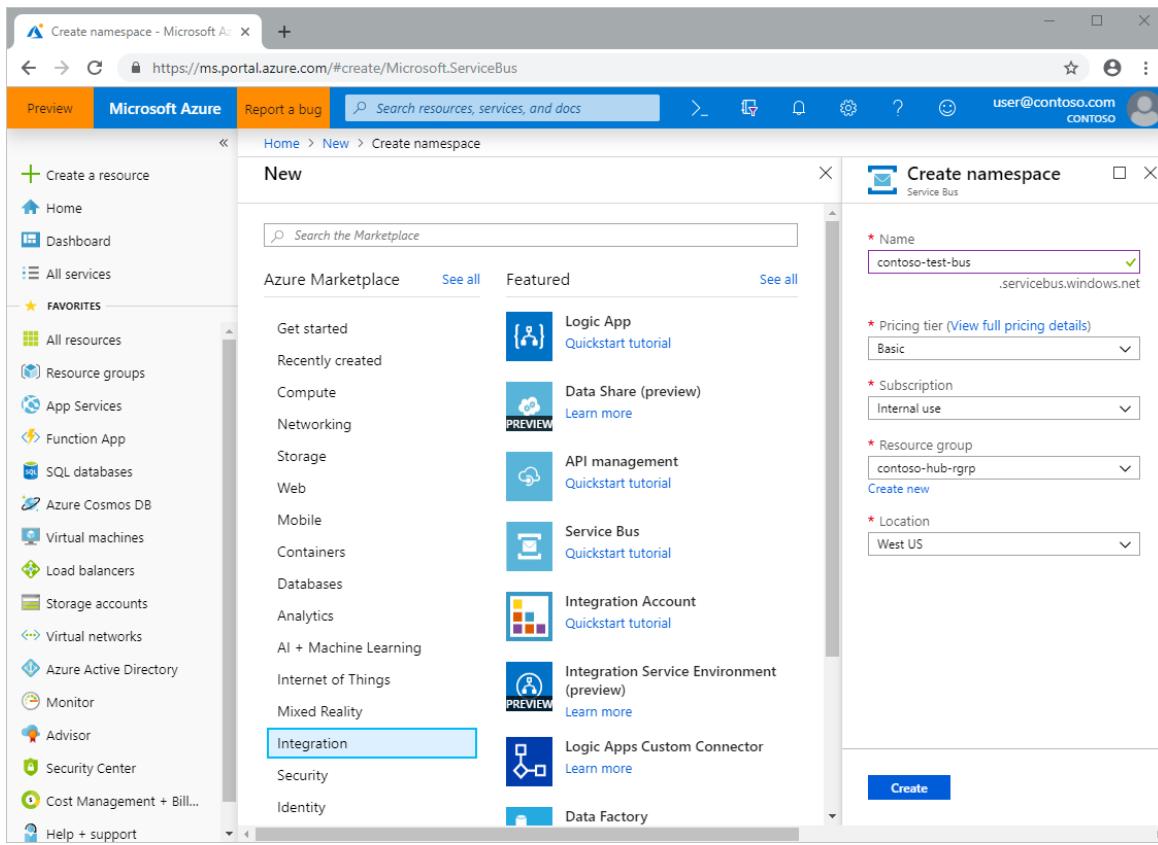
1. On the [Azure portal](#), select **+ Create a resource** > **Integration** > **Service Bus**.
2. On the **Create namespace** pane, provide the following information:

Name: The name of the service bus namespace. The namespace must be unique across Azure.

Pricing tier: Select **Basic** from the drop-down list. 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. Select **Create**. Wait for the deployment to complete before moving on to the next step.

Add a Service Bus queue to the namespace

1. Open the Service Bus namespace. The easiest way to get to the Service Bus namespace is to select **Resource groups** from the resource pane, select your resource group, then select the Service Bus namespace from the list of resources.
2. On the **Service Bus Namespace** pane, select **+ Queue**.
3. Enter a name for the queue and then select **Create**. When the queue has been successfully created, the **Create queue** pane closes.

4. Back on the **Service Bus Namespace** pane, under **Entities**, select **Queues**. Open the Service Bus queue from the list, and then select **Shared access policies > + Add**.
5. Enter a name for the policy, check **Manage**, and then select **Create**.

The screenshot shows the Azure portal interface for managing a Service Bus queue. On the left, there's a sidebar with options like Overview, Diagnose and solve problems, Settings, Shared access policies (which is selected), Metrics (preview), Properties, Locks, Export template, Support + troubleshooting, and New support request. The main area shows the 'Shared access policies' blade for the 'contoso-test-queue'. It has a search bar, an 'Add' button, and a table with columns 'POLICY' and 'CLAIMS'. A note says 'no policies have been set up yet.' A modal window titled 'Add SAS Policy' is overlaid, asking for a 'Policy name' (set to 'contoso-test-policy') and checkboxes for 'Manage', 'Send', and 'Listen', all of which are checked. A 'Create' button is at the bottom right of the modal.

Add a custom endpoint and routing rule to your IoT hub

Add a custom endpoint for the Service Bus queue to your IoT hub and create a message routing rule to direct messages that contain a temperature alert to that endpoint, where they will be picked up by your logic app. The routing rule uses a routing query, `temperatureAlert = "true"`, to forward messages based on the value of the `temperatureAlert` application property set by the client code running on the device. To learn more, see [Message routing query based on message properties](#).

Add a custom endpoint

1. Open your IoT hub. The easiest way to get to the IoT hub is to select **Resource groups** from the resource pane, select your resource group, then select your IoT hub from the list of resources.
2. Under **Messaging**, select **Message routing**. On the **Message routing** pane, select the **Custom endpoints** tab and then select **+ Add**. From the drop-down list, select **Service bus queue**.

The screenshot shows the Azure portal interface for managing an IoT hub. On the left, there's a sidebar with options like Locks, Export template, Explorers (Query explorer, IoT devices), Automatic Device Management (IoT Edge, IoT device configuration), Messaging (File upload, Message routing, which is selected), and Resiliency. The main area shows the 'Message routing' blade for the 'contoso-hub-1' IoT hub. It has a search bar, a note about sending data to endpoints, and tabs for 'Routes' and 'Custom endpoints' (which is selected). Below that, it says 'Choose which Azure services will receive your messages. You can add up to 10 endpoints to an IoT hub.' There's a '+ Add' button, a 'Synchronize keys' button, and a 'Delete' button. A dropdown menu is open, showing options: Event hubs, Service bus queue (highlighted with a red box), Service bus topic, Blob storage, and Service Bus topic/Blob storage.

3. On the **Add a service bus endpoint** pane, enter the following information:

Endpoint name: The name of the endpoint.

Service bus namespace: Select the namespace you created.

Service bus queue: Select the queue you created.



Add a service bus endpoint



Route your telemetry and device messages to Azure Service Bus and add publisher and subscriber capability.

* Endpoint name i

 ✓

Choose an existing service bus

Add an existing service bus queue or topic that shares a subscription with this IoT hub.

* Service bus namespace i

 ▼

* Service bus queue i

 ▼

Create

4. Select **Create**. After the endpoint is successfully created, proceed to the next step.

Add a routing rule

1. Back on the Message routing pane, select the **Routes** tab and then select **+ Add**.
2. On the **Add a route** pane, enter the following information:

Name: The name of the routing rule.

Endpoint: Select the endpoint you created.

Data source: Select **Device Telemetry Messages**.

Routing query: Enter `temperatureAlert = "true"`.

The screenshot shows the 'Add a route' dialog in the Azure portal. The 'Name' field is set to 'contoso-test-route'. The 'Endpoint' dropdown contains 'contoso-test-endpoint' with an 'Add' button next to it. The 'Data source' dropdown is set to 'Device Telemetry Messages'. The 'Enable route' switch is set to 'Enable'. Below these fields is a note: 'Create a query to filter messages before data is routed to an endpoint. [Learn more](#)'. Under the 'Routing query' heading, there is a code editor containing the query '1 temperatureAlert = "true"'. At the bottom of the dialog are 'Save' and 'Test' buttons.

3. Select **Save**. You can close the **Message routing** pane.

Create and configure a Logic App

In the preceding section, you set up your IoT hub to route messages containing a temperature alert to your Service Bus queue. Now, you set up a logic app to monitor the Service Bus queue and send an e-mail notification whenever a message is added to the queue.

Create a logic app

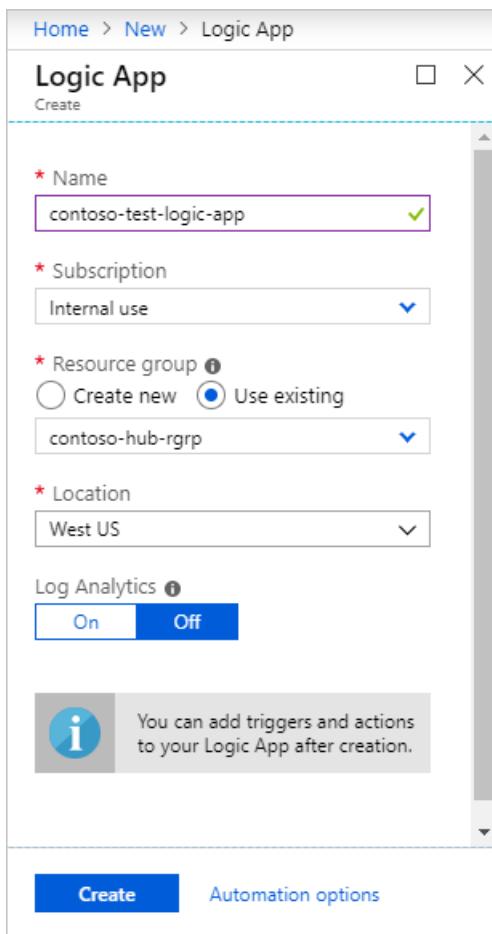
1. Select **Create a resource > Integration > Logic App**.

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.



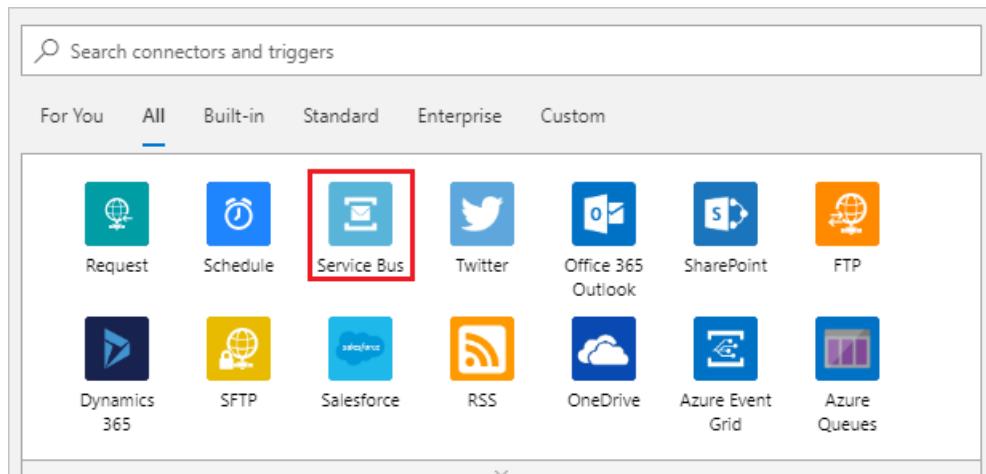
3. Select **Create**.

Configure the logic app trigger

1. Open the logic app. The easiest way to get to the logic app is to select **Resource groups** from the resource pane, select your resource group, then select your logic app from the list of resources. When you select the logic app, the Logic Apps Designer opens.
2. In the Logic Apps Designer, scroll down to **Templates** and select **Blank Logic App**.

The screenshot shows the Logic Apps Designer interface. The 'Templates' section is open, displaying a list of available templates. The 'Blank Logic App' template is highlighted with a red border. Other visible templates include 'Azure Monitor - Metrics Alert Handler' and 'Cancel runs by tracking id'.

3. Select the **All** tab and then select **Service Bus**.



4. Under Triggers, select When one or more messages arrive in a queue (auto-complete).

The screenshot shows the 'Search connectors and triggers' search bar at the top. Below it, a navigation bar includes 'Triggers' (which is selected) and 'Actions'. A list of triggers under 'Service Bus' is shown:

- When a message is received in a queue (auto-complete)
- When a message is received in a queue (peek-lock)
- When a message is received in a topic subscription (auto-complete)
- When a message is received in a topic subscription (peek-lock)
- When one or more messages arrive in a queue (auto-complete) (highlighted with a red box)
- When one or more messages arrive in a queue (peek-lock)
- When one or more messages arrive in a topic (auto-complete)
- When one or more messages arrive in a topic (peek-lock)

5. Create a service bus connection.

- a. Enter a connection name and select your Service Bus namespace from the list. The next screen opens.

When one or more messages arrive in a queue (auto-complete) ...

* Connection Name contoso-test-sb-connection

* Service Bus Namespace

| Name | Resource Group | Location |
|------------------|------------------|----------|
| contoso-test-bus | contoso-hub-rgrp | West US |
| vstegr | vstegrGroup | West US |

Create

[Manually enter connection information](#)

b. Select the service bus policy (RootManageSharedAccessKey). Then select **Create**.

When one or more messages arrive in a queue (auto-complete) ...

* Connection Name contoso-test-sb-connection

* Service Bus Policy

| Name | Rights |
|---------------------------|----------------------|
| RootManageSharedAccessKey | Listen, Manage, Send |

Create

[Manually enter connection information](#)

c. On the final screen, for **Queue name**, select the queue that you created from the drop-down.

Enter **175** for **Maximum message count**.

When one or more messages arrive in a queue (auto-complete) ⓘ ...

* Queue name contoso-test-queue

Maximum message count 175

Queue type Main

How often do you want to check for items?

| * Interval | * Frequency |
|------------|-------------|
| 3 | Minute |

Add new parameter

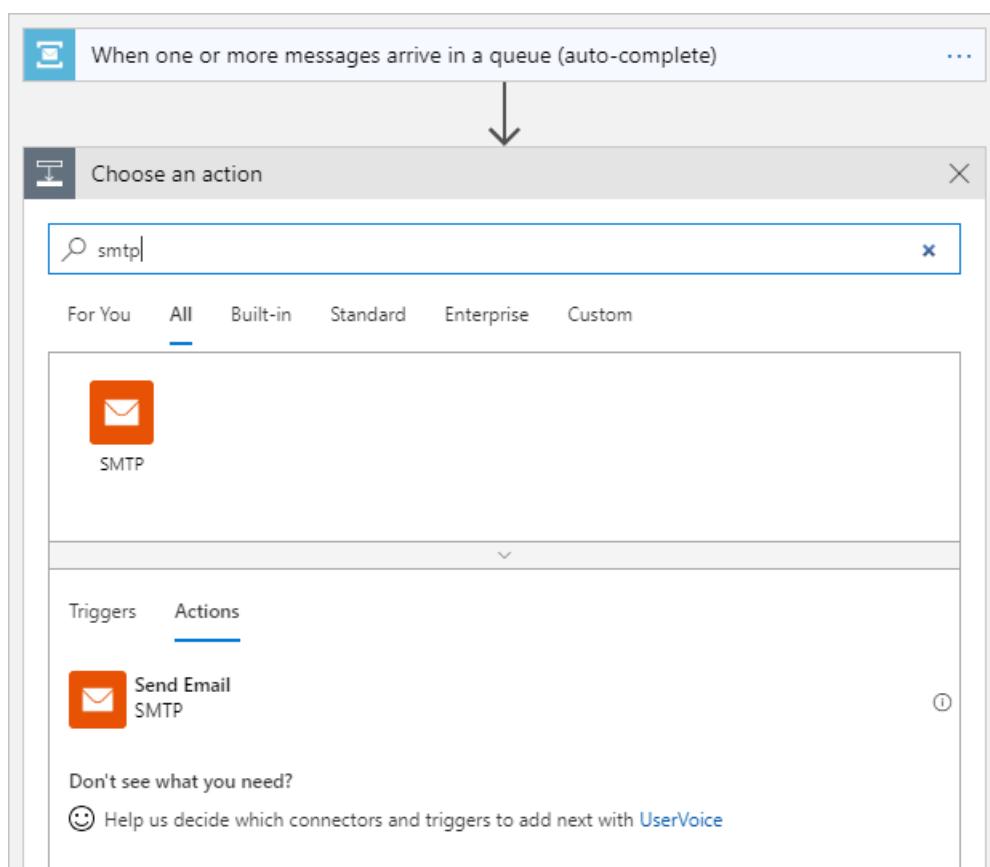
Connected to contoso-test-sb-connection. [Change connection](#).

d. Select **Save** on the menu at the top of the Logic Apps Designer to save your changes.

Configure the logic app action

1. Create an SMTP service connection.

- a. Select **New step**. In **Choose an action**, select the **All** tab.
- b. Type `smtp` in the search box, select the **SMTP** service in the search result, and then select **Send Email**.



- c. Enter the SMTP information for your mailbox, and then select **Create**.

The screenshot shows the 'SMTP' configuration dialog box. At the top, it says 'When one or more messages arrive in a queue (auto-complete)'. Below that is a large orange button labeled 'SMTP'. The form fields are as follows:

- *Connection Name: A text input field with placeholder text 'Enter name for connection'.
- *SMTP Server Address: A text input field with placeholder text 'SMTP Server Address'.
- *User Name: A text input field with placeholder text 'User Name'.
- *Password: A text input field with placeholder text 'Password'.
- SMTP Server Port: A text input field with placeholder text 'SMTP Port Number (example: 587)'.
- Enable SSL?: A checkbox labeled 'Enable SSL? (True/False)' with an info icon.

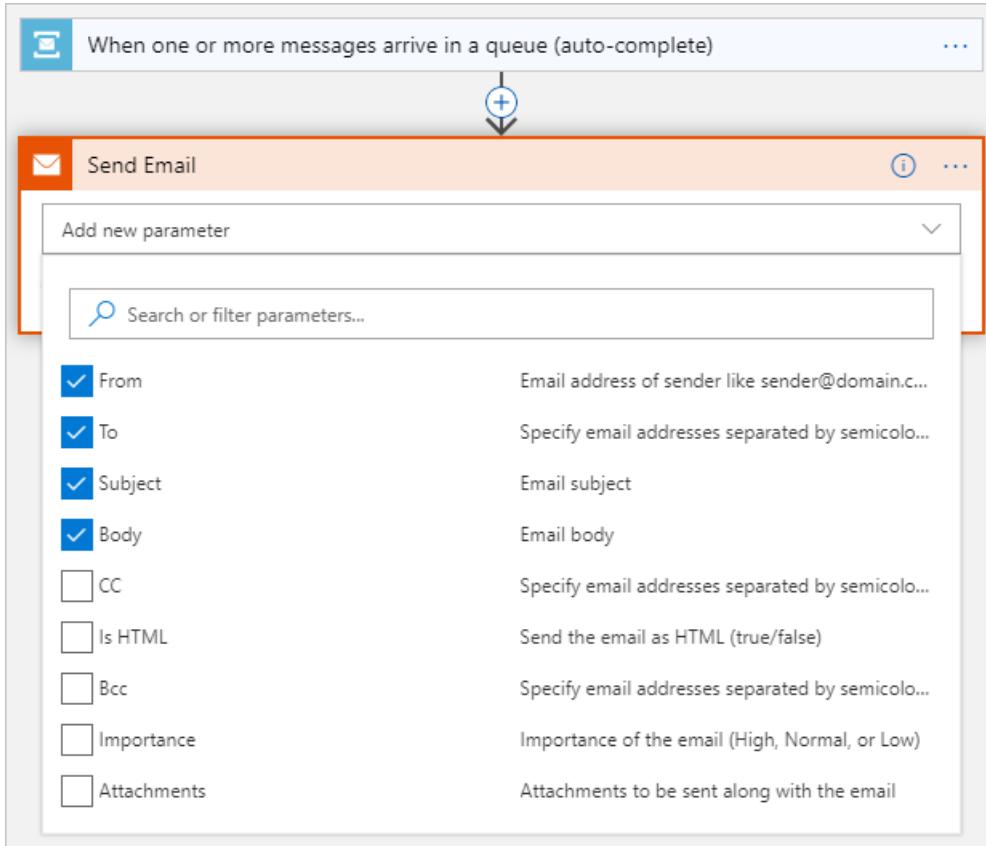
At the bottom right is a 'Create' button.

Get the SMTP information for [Hotmail/Outlook.com](#), [Gmail](#), and [Yahoo Mail](#).

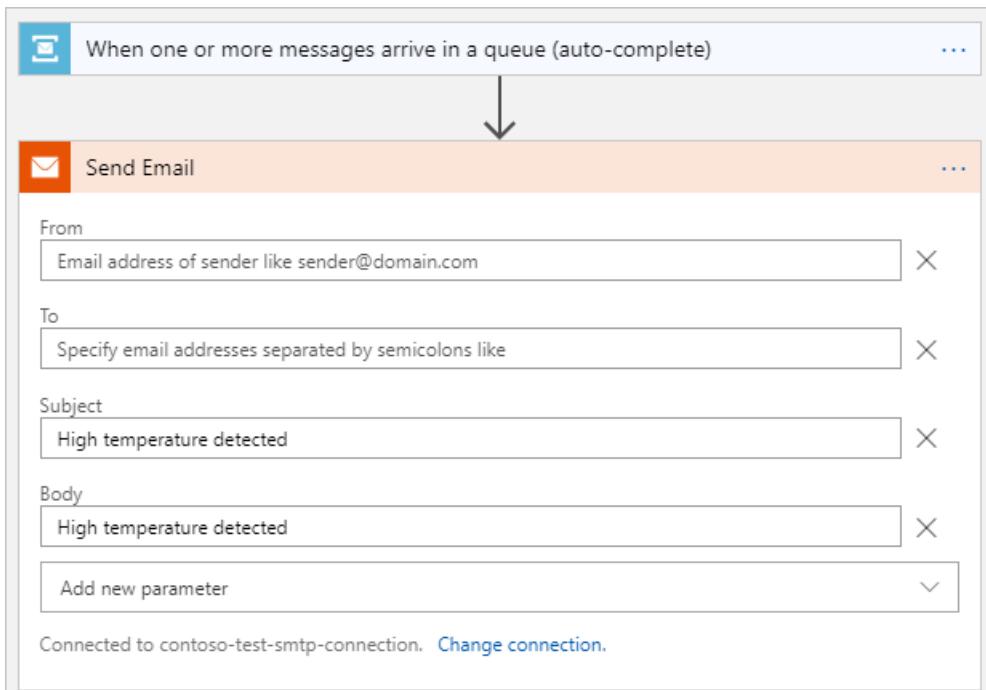
NOTE

You may need to disable TLS/SSL to establish the connection. If this is the case and you want to re-enable TLS after the connection has been established, see the optional step at the end of this section.

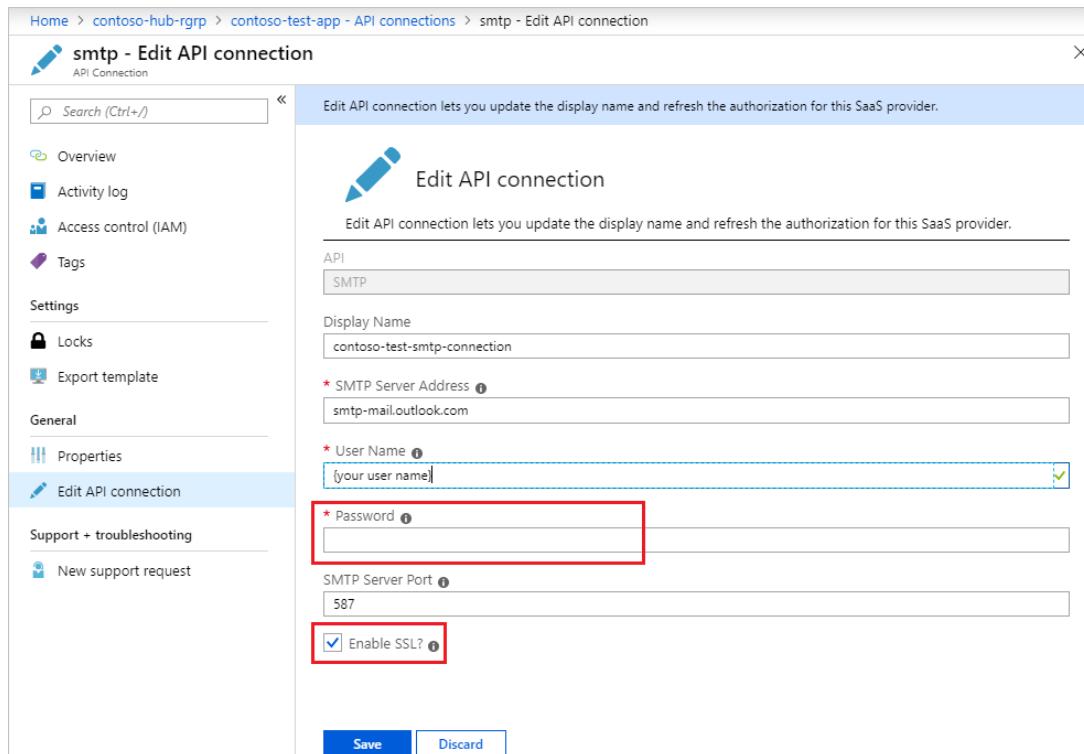
- d. From the **Add new parameter** drop-down on the **Send Email** step, select **From, To, Subject** and **Body**. Click or tap anywhere on the screen to close the selection box.



- e. Enter your email address for **From** and **To**, and **High temperature detected** for **Subject** and **Body**. If the **Add dynamic content from the apps and connectors used in this flow** dialog opens, select **Hide** to close it. You do not use dynamic content in this tutorial.



- f. Select **Save** to save the SMTP connection.
- (Optional) If you had to disable TLS to establish a connection with your email provider and want to re-enable it, follow these steps:
 - On the **Logic app** pane, under **Development Tools**, select **API connections**.
 - From the list of API connections, select the SMTP connection.
 - On the **smtp API Connection** pane, under **General**, select **Edit API connection**.
 - On the **Edit API Connection** pane, select **Enable SSL?**, re-enter the password for your email account, and select **Save**.



Your logic app is now ready to process temperature alerts from the Service Bus queue and send notifications to your email account.

Test the logic app

- Start the client application on your device.
- If you're using a physical device, carefully bring a heat source near the heat sensor until the temperature exceeds 30 degrees C. If you're using the online simulator, the client code will randomly output telemetry messages that exceed 30 C.
- You should begin receiving email notifications 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 extension for Visual Studio Code](#)
- [Manage devices with Azure IoT Hub extension for Visual Studio Code](#)
- [Set up message routing](#)
- [Use Power BI to visualize real-time sensor data from your IoT hub](#)
- [Use a web app 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](#)

Monitor, diagnose, and troubleshoot disconnects with Azure IoT Hub

7/29/2020 • 3 minutes to read • [Edit Online](#)

Connectivity issues for IoT devices can be difficult to troubleshoot because there are many possible points of failure. Application logic, physical networks, protocols, hardware, IoT Hub, and other cloud services can all cause problems. The ability to detect and pinpoint the source of an issue is critical. However, an IoT solution at scale could have thousands of devices, so it's not practical to check individual devices manually. To help you detect, diagnose, and troubleshoot these issues at scale, use the monitoring capabilities IoT Hub provides through Azure Monitor. These capabilities are limited to what IoT Hub can observe, so we also recommend that you follow monitoring best practices for your devices and other Azure services.

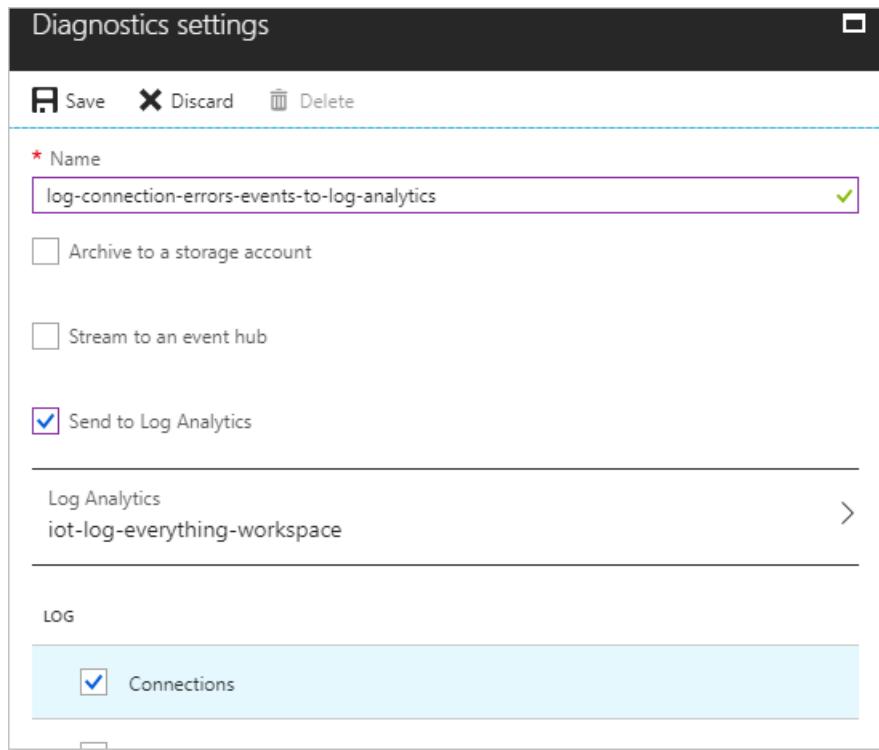
Get alerts and error logs

Use Azure Monitor to get alerts and write logs when devices disconnect.

Turn on diagnostic logs

To log device connection events and errors, turn on diagnostics for IoT Hub. We recommend turning on these logs as early as possible, because if diagnostic logs aren't enabled, when device disconnects occur, you won't have any information to troubleshoot the problem with.

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 device disconnect at scale

To get alerts when devices disconnect, configure alerts on the **Connected devices (preview)** metric.

1. Sign in to the [Azure portal](#).
2. Browse to your IoT hub.
3. Select **Alerts**.
4. Select **New alert rule**.
5. Select **Add condition**, then select "Connected devices (preview)".
6. Set up threshold and alerting by following prompts.

To learn more, see [What are alerts in Microsoft Azure?](#).

Detecting individual device disconnects

To detect *per-device* disconnects, such as when you need to know a factory just went offline, [configure device disconnect events with Event Grid](#).

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 look for common issues when you receive an alert. The steps below assume you've set up Azure Monitor logs for your diagnostic logs.

1. Sign in to the [Azure portal](#).
2. Browse to your IoT hub.
3. Select **Logs**.
4. To isolate connectivity error logs for IoT Hub, enter the following query and then select **Run**:

```

AzureDiagnostics
| where ( ResourceType == "IOTHUBS" and Category == "Connections" and Level == "Error")

```

5. 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 | LinkedMeterId | Type | MeterId | Computer | R |
| ▲ AzureDiagnostics | 7/5/2018 1:48:49.000 PM | | AzureDiagnostics | | | 4 |
| \$table | AzureDiagnostics | | | | | |
| TenantId | 747ddfe5-xxxx-xxxx-xxxx-xxxxxxxxxxxxxx | | | | | |
| SourceSystem | Azure | | | | | |
| TimeGenerated | 2018-07-05T20:48:49Z | | | | | |
| Type | AzureDiagnostics | | | | | |
| ResultType | 404104 | | | | | |
| ResultDescription | DeviceConnectionClosedRemotely | | | | | |
| ResourceId | /SUBSCRIPTIONS/838AAE4-XXXX-XXXX-XXXX-XXXXXXXXXX/RESOURCEGROUPS/US-WEST-SG/PROVIDERS/MICROSOFT.DEVICES/IOTHUBS/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-xxxxxxxxxxxxxx | | | | | |
| ResourceGroup | US-WEST-SG | | | | | |
| ResourceProvider | MICROSOFT.DEVICES | | | | | |

6. Follow the problem resolution guides for the most common errors:

- [404104 DeviceConnectionClosedRemotely](#)
- [401003 IoTHubUnauthorized](#)
- [409002 LinkCreationConflict](#)
- [500001 ServerError](#)
- [500008 GenericTimeout](#)

I tried the steps, but they didn't work

If the previous steps didn't help, 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.
- If your device uses MQTT protocol, verify that port 8883 is open. For more information, see [Connecting to IoT Hub \(MQTT\)](#).
- Get help from [Microsoft Q&A question page for Azure IoT Hub](#), [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](#).

Troubleshooting message routing

7/29/2020 • 7 minutes to read • [Edit Online](#)

This article provides monitoring and troubleshooting guidance for common issues and resolution for IoT Hub message routing.

Monitoring message routing

[IoT Hub metrics](#) lists all metrics that are enabled by default for your IoT Hub. We recommend you monitor metrics related to message routing and endpoints to give you an overview of the messages sent. Also turn on [diagnostic logs](#) in Azure Monitor diagnostic settings, to track operations for [routes](#). These diagnostic logs can be sent to Azure Monitor logs, Event Hubs, or Azure Storage for custom processing. Learn how to [set up and use metrics and diagnostic logs with an IoT Hub](#).

We also recommend enabling the [fallback route](#) if you want to maintain messages that don't match the query on any of the routes. These can be retained in the [built-in endpoint](#) for the amount of retention days configured.

Top issues

The following are the most common issues observed with message routing. To start troubleshooting, click on the issue for detailed steps.

- [Messages from my devices are not being routed as expected](#)
- [I suddenly stopped getting messages at the built-in Event Hubs endpoint](#)

Messages from my devices are not being routed as expected

To troubleshoot this issue, analyze the following.

The routing metrics for this endpoint

All the [IoT Hub metrics](#) related to routing are prefixed with *Routing*. You can combine information from multiple metrics to identify root cause for issues. For example, use metric [Routing Delivery Attempts](#) to identify the number of messages that were delivered to an endpoint or dropped when they didn't match queries on any of the routes and fallback route was disabled. Check the [Routing Latency](#) metric to observe whether latency for message delivery is steady or increasing. A growing latency can indicate a problem with a specific endpoint and we recommend checking [the health of the endpoint](#). These routing metrics also have [dimensions](#) that provide details on the metric like the endpoint type, specific endpoint name and a reason why the message was not delivered.

The diagnostic logs for any operational issues

Observe the [routes](#) [diagnostic logs](#) to get more information on the routing and endpoint [operations](#) or identify errors and relevant [error code](#) to understand the issue further. For example, the operation name [RouteEvaluationError](#) in the log indicates the route could not be evaluated because of an issue with the message format. Use the tips provided for the specific [operation names](#) to mitigate the issue. When an event is logged as an error, the log will also provide more information on why the evaluation failed. For example, if the operation name is [EndpointUnhealthy](#), an [Error codes](#) of 403004 indicates the endpoint ran out of space.

The health of the endpoint

Use the REST API [Get Endpoint Health](#) to get [health status](#) of the endpoints. The [Get Endpoint Health](#) API also provides information on the last time a message was successfully sent to the endpoint, the [last known error](#), last known error time and the last time a send attempt was made for this endpoint. Use the possible mitigation provided for the specific [last known error](#).

I suddenly stopped getting messages at the built-in endpoint

To troubleshoot this issue, analyze the following.

Was a new route created?

Once a route is created, data stops flowing to the built-in-endpoint, unless a route is created to that endpoint. To ensure messages continue to flow to the built-in-endpoint if a new route is added, configure a route to the *events* endpoint.

Was the Fallback route disabled?

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. 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.

Last known errors for IoT Hub routing endpoints

[Get Endpoint Health](#) in the REST API gives the health status of the endpoints, as well as the last known error, to identify the reason an endpoint is not healthy. The table below lists the most common errors.

| LAST KNOWN ERROR | DESCRIPTION/WHEN IT OCCURS | POSSIBLE MITIGATION |
|------------------------|---|--|
| Transient | A transient error has occurred and IoT Hub will retry the operation. | Observe routes diagnostic logs . |
| InternalError | An error occurred while delivering a message to an endpoint. | This is an internal exception but also observe the routes diagnostic logs . |
| Unauthorized | IoT Hub is not authorized to send messages to the specified endpoint. | Validate that the connection string is up to date for the endpoint. If it has changed, consider an update on your IoT Hub. If the endpoint uses managed identity, check that the IoT Hub principal has the required permissions on the target. |
| Throttled | IoT Hub is being throttled while writing messages into the endpoint. | Review the throttle limits for the affected endpoint. Modify configurations for the endpoint to scale up if needed. |
| Timeout | Operation timeout. | Retry the operation. |
| Not Found | Target resource does not exist. | Ensure that the target resource exists. |
| Container Not Found | Storage container does not exist. | Ensure the storage container exists. |
| Container disabled | Storage container is disabled. | Ensure the storage container is enabled. |
| MaxMessageSizeExceeded | Message routing has a message size limit of 256Kb. The message size being routed exceeded this limit. | Check if message size can be reduced by using fewer application properties or fewer message enrichments. |

| Last Known Error | Description/When it occurs | Possible Mitigation |
|---|--|---|
| PartitioningAndDuplicateDetectionNotSupported | Service bus may not have duplicate detection enabled. | Disable duplicate detection from Service Bus or consider using an entity without duplicate detection. |
| SessionfulEntityNotSupported | Service bus may not have sessions enabled. | Disable session from Service Bus or consider using an entity without sessions. |
| NoMatchingSubscriptionsForMessage | There is no subscription to write message on the service bus topic. | Create a subscription for IoT Hub messages to be routed to. |
| EndpointExternallyDisabled | Endpoint is not in an active state so IoT Hub can send messages to it. | Enable the endpoint to bring it back to active state. |
| DeviceMaximumQueueDepthExceeded | Service bus size limit has been reached. | Consider removing messages from the target Event Hubs to allow new messages to be ingested into the Event Hubs. |

Routes diagnostic logs

The following are the operation names and error codes logged in the [diagnostic logs](#).

Operation Names

| Operation Name | Level | Description |
|--------------------------|-------------|---|
| UndefinedRouteEvaluation | Information | The message cannot be evaluated with a giving condition. For example, if a property in the route query condition is absent in the message. Learn more about routing query syntax . |
| RouteEvaluationError | Error | There was an error evaluating the message because of an issue with the message format. For example, this error will be logged if the content encoding not specified or Content type not valid in the message. These must be set in the system properties . |
| DroppedMessage | Error | Message was dropped and not routed. This could be due to reasons like message didn't match any routing query or endpoint was dead and message could not be delivered after several retries. We recommend getting more details on the endpoint by using the REST API get endpoint health . |
| EndpointUnhealthy | Error | Endpoint has not been accepting messages from IoT Hub and IoT Hub is trying to resend the messages. We recommend observing the last known error via the REST API get endpoint health . |

| OPERATION NAME | LEVEL | DESCRIPTION |
|-----------------|-------------|---|
| EndpointDead | Error | Endpoint has not been accepting messages from IoT Hub for over an hour. We recommend observing the last known error via the REST API get endpoint health . |
| EndpointHealthy | Information | Endpoint is healthy and receiving messages from IoT Hub. This message is not logged continuously, but logged only when the endpoint becomes healthy again. This message means IoT Hub was unable to send messages to the endpoint, but the endpoint is now healthy. |
| OrphanedMessage | Information | The message does not match to any route. |
| InvalidMessage | Error | Message is invalid because of incompatibility with the endpoint. We recommend check configurations of the endpoint. |

The operations *UndefinedRouteEvaluation*, *RouteEvaluationError* and *OrphanedMessage* are throttled and logged no more than once a minute per IoT Hub.

Common error codes

| ERROR CODE | DESCRIPTION |
|------------|--|
| 401002 | IoT Hub Unauthorized Access |
| 413001 | Message too large |
| 403004 | Device maximum queue depth exceeded |
| 503008 | Receive link throttled |
| 500000 | Generic Server error |
| 401 | Unauthorized |
| 503 | Service Unavailable |
| 500001 | Server Error |
| 400103 | Invalid Content Encoding Or Content Type |
| 404001 | Device Not found |

Next steps

If you need more help, you can contact the Azure experts on [the MSDN Azure and Stack Overflow forums](#). Alternatively, you can file an Azure support incident. Go to the [Azure support site](#) and select **Get Support**.

400027

ConnectionForcefullyClosedOnNewConnection

1/31/2020 • 2 minutes to read • [Edit Online](#)

This article describes the causes and solutions for **400027 ConnectionForcefullyClosedOnNewConnection** errors.

Symptoms

Your device-to-cloud twin operation (such as read or patch reported properties) or direct method invocation fails with the error code **400027**.

Cause

Another client created a new connection to IoT Hub using the same credentials, so IoT Hub closed the previous connection. IoT Hub doesn't allow more than one client to connect using the same set of credentials.

Solution

Ensure that each client connects to IoT Hub using its own identity.

401003 IoTHubUnauthorized

4/21/2020 • 2 minutes to read • [Edit Online](#)

This article describes the causes and solutions for **401003 IoTHubUnauthorized** errors.

Symptoms

Symptom 1

In diagnostic logs, you see a pattern of devices disconnecting with **401003 IoTHubUnauthorized**, followed by **404104 DeviceConnectionClosedRemotely**, and then successfully connecting shortly after.

Symptom 2

Requests to IoT Hub fail with one of the following error messages:

- Authorization header missing
- IoTHub '*' does not contain the specified device '/*'
- Authorization rule '*' does not allow access for '/*'
- Authentication failed for this device, renew token or certificate and reconnect
- Thumbprint does not match configuration: Thumbprint: SHA1Hash=*, SHA2Hash=*; Configuration: PrimaryThumbprint=*, SecondaryThumbprint=*

Cause

Cause 1

For MQTT, some SDKs rely on IoT Hub to issue the disconnect when the SAS token expires to know when to refresh it. So,

1. The SAS token expires
2. IoT Hub notices the expiration, and disconnects the device with **401003 IoTHubUnauthorized**
3. The device completes the disconnection with **404104 DeviceConnectionClosedRemotely**
4. The IoT SDK generates a new SAS token
5. The device reconnects with IoT Hub successfully

Cause 2

IoT Hub couldn't authenticate the auth header, rule, or key.

Solution

Solution 1

No action needed if using IoT SDK for connection using the device connection string. IoT SDK regenerates the new token to reconnect on SAS token expiration.

If the volume of errors is a concern, switch to the C SDK, which renews the SAS token before expiration.

Additionally, for AMQP the SAS token can refresh without disconnection.

Solution 2

In general, the error message presented should explain how to fix the error. If for some reason you don't have access to the error message detail, make sure:

- The SAS or other security token you use isn't expired.

- The authorization credential is well formed for the protocol that you use. To learn more, see [IoT Hub access control](#).
- The authorization rule used has the permission for the operation requested.

Next steps

To make authenticating to IoT Hub easier, we recommend using [Azure IoT SDKs](#).

403002 IoTHubQuotaExceeded

1/31/2020 • 2 minutes to read • [Edit Online](#)

This article describes the causes and solutions for 403002 IoTHubQuotaExceeded errors.

Symptoms

All requests to IoT Hub fail with the error 403002 IoTHubQuotaExceeded. In Azure portal, the IoT hub device list doesn't load.

Cause

The daily message quota for the IoT hub is exceeded.

Solution

[Upgrade or increase the number of units on the IoT hub](#) or wait for the next UTC day for the daily quota to refresh.

Next steps

- To understand how operations are counted toward the quota, such as twin queries and direct methods, see [Understand IoT Hub pricing](#)
- To set up monitoring for daily quota usage, set up an alert with the metric *Total number of messages used*. For step-by-step instructions, see [Set up metrics and alerts with IoT Hub](#)

403004 DeviceMaximumQueueDepthExceeded

4/21/2020 • 2 minutes to read • [Edit Online](#)

This article describes the causes and solutions for **403004 DeviceMaximumQueueDepthExceeded** errors.

Symptoms

When trying to send a cloud-to-device message, the request fails with the error **403004** or **DeviceMaximumQueueDepthExceeded**.

Cause

The underlying cause is that the number of messages enqueued for the device exceeds the [queue limit \(50\)](#).

The most likely reason that you're running into this limit is because you're using HTTPS to receive the message, which leads to continuous polling using `ReceiveAsync`, resulting in IoT Hub throttling the request.

Solution

The supported pattern for cloud-to-device messages with HTTPS is intermittently connected devices that check for messages infrequently (less than every 25 minutes). To reduce the likelihood of running into the queue limit, switch to AMQP or MQTT for cloud-to-device messages.

Alternatively, enhance device side logic to complete, reject, or abandon queued messages quickly, shorten the time to live, or consider sending fewer messages. See [C2D message time to live](#).

Lastly, consider using the [Purge Queue API](#) to periodically clean up pending messages before the limit is reached.

403006

DeviceMaximumActiveFileUploadLimitExceeded

1/31/2020 • 2 minutes to read • [Edit Online](#)

This article describes the causes and solutions for 403006 DeviceMaximumActiveFileUploadLimitExceeded errors.

Symptoms

Your file upload request fails with the error code **403006** and a message "Number of active file upload requests cannot exceed 10".

Cause

Each device client is limited to [10 concurrent file uploads](#).

You can easily exceed the limit if your device doesn't notify IoT Hub when file uploads are completed. This problem is commonly caused by an unreliable device side network.

Solution

Ensure the device can promptly [notify IoT Hub file upload completion](#). Then, try [reducing the SAS token TTL for file upload configuration](#).

Next steps

To learn more about file uploads, see [Upload files with IoT Hub](#) and [Configure IoT Hub file uploads using the Azure portal](#).

404001 DeviceNotFound

1/31/2020 • 2 minutes to read • [Edit Online](#)

This article describes the causes and solutions for **404001 DeviceNotFound** errors.

Symptoms

During a cloud-to-device (C2D) communication, such as C2D message, twin update, or direct method, the operation fails with error **404001 DeviceNotFound**.

Cause

The operation failed because the device cannot be found by IoT Hub. The device is either not registered or disabled.

Solution

Register the device ID that you used, then try again.

404103 DeviceNotOnline

1/31/2020 • 2 minutes to read • [Edit Online](#)

This article describes the causes and solutions for **404103 DeviceNotOnline** errors.

Symptoms

A direct method to a device fails with the error **404103 DeviceNotOnline** even if the device is online.

Cause

If you know that the device is online and still get the error, it's likely because the direct method callback isn't registered on the device.

Solution

To configure your device properly for direct method callbacks, see [Handle a direct method on a device](#).

404104 DeviceConnectionClosedRemotely

4/21/2020 • 2 minutes to read • [Edit Online](#)

This article describes the causes and solutions for **404104 DeviceConnectionClosedRemotely** errors.

Symptoms

Symptom 1

Devices disconnect at a regular interval (every 65 minutes, for example) and you see **404104 DeviceConnectionClosedRemotely** in IoT Hub diagnostic logs. Sometimes, you also see **401003 IoTHubUnauthorized** and a successful device connection event less than a minute later.

Symptom 2

Devices disconnect randomly, and you see **404104 DeviceConnectionClosedRemotely** in IoT Hub diagnostic logs.

Symptom 3

Many devices disconnect at once, you see a dip in the [connected devices metric](#), and there are more **404104 DeviceConnectionClosedRemotely** and [500xxx Internal errors](#) in diagnostic logs than usual.

Causes

Cause 1

The [SAS token used to connect to IoT Hub](#) expired, which causes IoT Hub to disconnect the device. The connection is re-established when the token is refreshed by the device. For example, [the SAS token expires every hour by default for C SDK](#), which can lead to regular disconnects.

To learn more, see [401003 IoTHubUnauthorized cause](#).

Cause 2

Some possibilities include:

- The device lost underlying network connectivity longer than the [MQTT keep-alive](#), resulting in a remote idle timeout. The MQTT keep-alive setting can be different per device.
- The device sent a TCP/IP-level reset but didn't send an application-level `MQTT DISCONNECT`. Basically, the device abruptly closed the underlying socket connection. Sometimes, this issue is caused by bugs in older versions of the Azure IoT SDK.
- The device side application crashed.

Cause 3

IoT Hub might be experiencing a transient issue. See [IoT Hub internal server error cause](#).

Solutions

Solution 1

See [401003 IoTHubUnauthorized solution 1](#)

Solution 2

- Make sure the device has good connectivity to IoT Hub by [testing the connection](#). If the network is

unreliable or intermittent, we don't recommend increasing the keep-alive value because it could result in detection (via Azure Monitor alerts, for example) taking longer.

- Use the latest versions of the [IoT SDKs](#).

Solution 3

See [solutions to IoT Hub internal server errors](#).

Next steps

We recommend using Azure IoT device SDKs to manage connections reliably. To learn more, see [Manage connectivity and reliable messaging by using Azure IoT Hub device SDKs](#)

409001 DeviceAlreadyExists

1/31/2020 • 2 minutes to read • [Edit Online](#)

This article describes the causes and solutions for **409001 DeviceAlreadyExists** errors.

Symptoms

When trying to register a device in IoT Hub, the request fails with the error **409001 DeviceAlreadyExists**.

Cause

There's already a device with the same device ID in the IoT hub.

Solution

Use a different device ID and try again.

409002 LinkCreationConflict

4/21/2020 • 2 minutes to read • [Edit Online](#)

This article describes the causes and solutions for **409002 LinkCreationConflict** errors.

Symptoms

You see the error **409002 LinkCreationConflict** logged in diagnostic logs along with device disconnection or cloud-to-device message failure.

Cause

Generally, this error happens when IoT Hub detects a client has more than one connection. In fact, when a new connection request arrives for a device with an existing connection, IoT Hub closes the existing connection with this error.

Cause 1

In the most common case, a separate issue (such as [404104 DeviceConnectionClosedRemotely](#)) causes the device to disconnect. The device tries to reestablish the connection immediately, but IoT Hub still considers the device connected. IoT Hub closes the previous connection and logs this error.

Cause 2

Faulty device-side logic causes the device to establish the connection when one is already open.

Solution

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.

412002 DeviceMessageLockLost

1/31/2020 • 2 minutes to read • [Edit Online](#)

This article describes the causes and solutions for **412002 DeviceMessageLockLost** errors.

Symptoms

When trying to send a cloud-to-device message, the request fails with the error **412002 DeviceMessageLockLost**.

Cause

When a device receives a cloud-to-device message from the queue (for example, using `ReceiveAsync()`) the message is locked by IoT Hub for a lock timeout duration of one minute. If the device tries to complete the message after the lock timeout expires, IoT Hub throws this exception.

Solution

If IoT Hub doesn't get the notification within the one-minute lock timeout duration, it sets the message back to *Enqueued* state. The device can attempt to receive the message again. To prevent the error from happening in the future, implement device side logic to complete the message within one minute of receiving the message. This one-minute time-out can't be changed.

429001 ThrottlingException

1/31/2020 • 2 minutes to read • [Edit Online](#)

This article describes the causes and solutions for **429001 ThrottlingException** errors.

Symptoms

Your requests to IoT Hub fail with the error **429001 ThrottlingException**.

Cause

IoT Hub [throttling limits](#) have been exceeded for the requested operation.

Solution

Check if you're hitting the throttling limit by comparing your *Telemetry message send attempts* metric against the limits specified above. You can also check the *Number of throttling errors* metric. For more information about these and other metrics available for IoT Hub, see [IoT Hub metrics and how to use them](#).

IoT Hub returns 429 ThrottlingException only after the limit has been violated for too long a period. This is done so that your messages aren't dropped if your IoT hub gets burst traffic. In the meantime, IoT Hub processes the messages at the operation throttle rate, which might be slow if there's too much traffic in the backlog. To learn more, see [IoT Hub traffic shaping](#).

Next steps

Consider [scaling up your IoT Hub](#) if you're running into quota or throttling limits.

500xxx Internal errors

1/31/2020 • 2 minutes to read • [Edit Online](#)

This article describes the causes and solutions for **500xxx Internal errors**.

Symptoms

Your request to IoT Hub fails with an error that begins with 500 and/or some sort of "server error". Some possibilities are:

- **500001 ServerError**: IoT Hub ran into a server-side issue.
- **500008 GenericTimeout**: IoT Hub couldn't complete the connection request before timing out.
- **ServiceUnavailable (no error code)**: IoT Hub encountered an internal error.
- **InternalServerError (no error code)**: IoT Hub encountered an internal error.

Cause

There can be a number of causes for a 500xxx error response. In all cases, the issue is most likely 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.

Solution

To mitigate 500xxx errors, 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, check [Resource Health](#) and [Azure Status](#) to see if IoT Hub has a known problem. You can also use the [manual failover feature](#). If there are no known problems and the issue continues, [contact support](#) for further investigation.

503003 PartitionNotFound

1/31/2020 • 2 minutes to read • [Edit Online](#)

This article describes the causes and solutions for **503003 PartitionNotFound** errors.

Symptoms

Requests to IoT Hub fail with the error **503003 PartitionNotFound**.

Cause

This error is internal to IoT Hub and is likely transient. See [IoT Hub internal server error cause](#).

Solution

See [solutions to IoT Hub internal server errors](#).

504101 GatewayTimeout

4/21/2020 • 2 minutes to read • [Edit Online](#)

This article describes the causes and solutions for **504101 GatewayTimeout** errors.

Symptoms

When trying to invoke a direct method from IoT Hub to a device, the request fails with the error **504101 GatewayTimeout**.

Cause

Cause 1

IoT Hub encountered an error and couldn't confirm if the direct method completed before timing out.

Cause 2

When using an earlier version of the Azure IoT C# SDK (<1.19.0), the AMQP link between the device and IoT Hub can be dropped silently because of a bug.

Solution

Solution 1

Issue a retry.

Solution 2

Upgrade to the latest version of the Azure IOT C# SDK.

IoT Hub operations monitoring (deprecated)

4/21/2020 • 6 minutes to read • [Edit Online](#)

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 'getStartedWithAnIoTHub' hub. The left sidebar lists various management options 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 currently selected and has a red box around the 'Operations monitoring' link. The main pane displays the hub's configuration details: Resource group (getStartedWithAnIoTHub_rg), Status (Active), Location (West US), Subscription name (Visual Studio Enterprise), and Subscription ID (your subscription id). Below this is a summary card showing metrics: 3/24/2017 UTC, GETSTARTEDWITHANIOUTHUB, 0% TOTAL (for messages), 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 Error Verbose

Device-to-cloud communications None Error Verbose

Cloud-to-device communications None Error Verbose

Connections None Error Verbose

File uploads None Error Verbose

Message routing None Error Verbose

Monitoring settings

Partitions 2 []

Event Hub-compatible name iothub-ehub-getstarted-96435-6074af59 []

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' settings for an IoT Hub. The 'Operations monitoring' section is highlighted with a red box. Below it, the 'Event Hub-compatible name' and 'Event Hub-compatible endpoint' fields are also 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' page for an IoT Hub. The 'Shared access policies' section is highlighted with a red box. A modal window is open for the 'service' access policy, showing its permissions and shared access keys. The 'Primary key' value is 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

1/8/2020 • 3 minutes to read • [Edit Online](#)

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

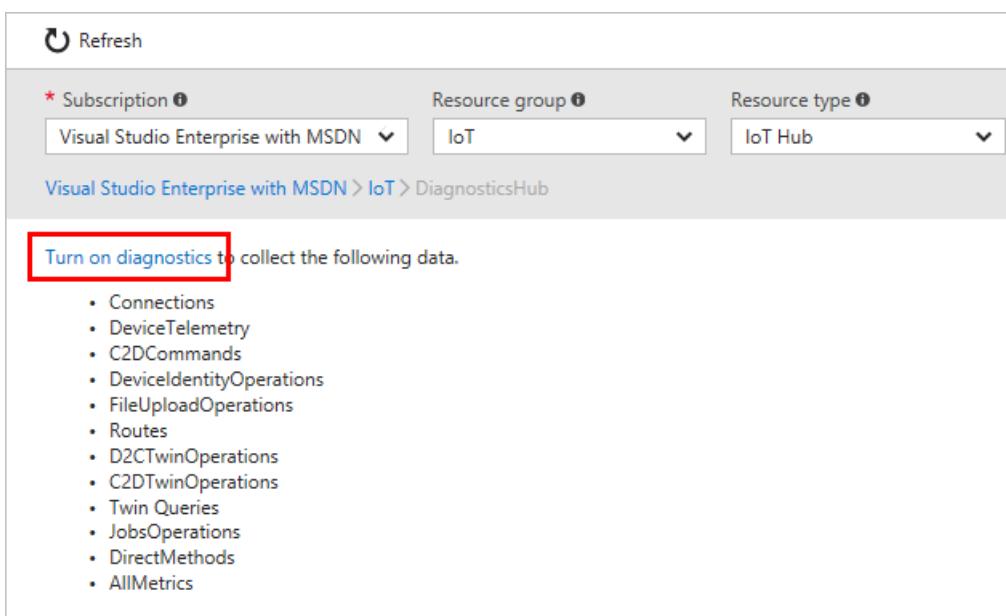
To update your IoT Hub in the Azure portal, first turn on diagnostics settings, then turn off operations monitoring.

Enable logging with diagnostics settings

NOTE

This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module installation instructions, see [Install Azure PowerShell](#).

1. Sign in to the [Azure portal](#) and navigate to your IoT hub.
2. Select **Diagnostics settings**.
3. Select **Turn on diagnostics**.



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-AzAccount  
Select-AzSubscription -SubscriptionName <subscription that includes your IoT Hub>  
Set-AzDiagnosticSetting -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](#)

Deprecation of TLS 1.0 and 1.1 in IoT Hub

7/29/2020 • 2 minutes to read • [Edit Online](#)

To provide best-in-class encryption, IoT Hub is moving to Transport Layer Security (TLS) 1.2 as the encryption mechanism of choice for IoT devices and services.

Timeline

IoT Hub will continue to support TLS 1.0/1.1 until further notice. However, we recommend that all customers migrate to TLS 1.2 as soon as possible.

Deprecating TLS 1.1 ciphers

- [TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA](#)
- [TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA](#)
- [TLS_RSA_WITH_AES_256_CBC_SHA](#)
- [TLS_RSA_WITH_AES_128_CBC_SHA](#)
- [TLS_RSA_WITH_3DES_EDE_CBC_SHA](#)

Deprecating TLS 1.0 ciphers

- [TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA](#)
- [TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA](#)
- [TLS_RSA_WITH_AES_256_CBC_SHA](#)
- [TLS_RSA_WITH_AES_128_CBC_SHA](#)
- [TLS_RSA_WITH_3DES_EDE_CBC_SHA](#)

TLS 1.2 ciphers

See [IoT Hub TLS 1.2 recommended ciphers](#).

Customer feedback

While the TLS 1.2 enforcement is an industry-wide best-in-class encryption choice and will be enabled as planned, we still would like to hear from customers regarding their specific deployments and difficulties adopting TLS 1.2. For this purpose, you can send your comments to iot_tls1_deprecation@microsoft.com.

Customer data request features for Azure IoT Hub devices

7/29/2020 • 2 minutes to read • [Edit Online](#)

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 pane 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](#).