

Contents

Azure IoT Hub Device Provisioning Service documentation

Overview

[What is IoT Hub Device Provisioning Service?](#)

Quickstarts

[Set up DPS](#)

[Azure portal](#)

[Azure CLI](#)

[Bicep](#)

[ARM template](#)

[Provision a simulated symmetric key device](#)

[Provision a simulated X.509 certificate device](#)

[Simulated TPM device attestation sample](#)

Tutorials

[1 - Provision sets of devices with enrollment groups](#)

[Provision devices with symmetric keys](#)

[Use custom allocation policies](#)

[Provision multiple X.509 devices](#)

[2 - Provision devices to multiple hubs](#)

[Link multiple hubs to DPS](#)

[Provision for multitenancy](#)

Concepts

[DPS terminology](#)

[Symmetric key attestation](#)

[X.509 certificate attestation](#)

[TPM attestation](#)

[Control access to DPS](#)

[Overview](#)

[Control access to DPS with SAS](#)

[Control access to DPS with Azure AD \(preview\)](#)

- [Roles and operations](#)
- [Virtual networks support](#)
- [Reprovisioning](#)
- [Best practices for large-scale IoT device deployments](#)
- [High availability and disaster recovery](#)
- [Understanding DPS IP addresses](#)
- [TLS support](#)
- [Security practices for device manufacturers](#)
- [How-to guides](#)
 - [Develop](#)
 - [How to send additional data from devices](#)
 - [Provision devices using custom allocation policies](#)
 - [Create an X.509 enrollment group with DPS service SDK](#)
 - [Create a TPM individual enrollment with DPS service SDK](#)
 - [Communicate with your DPS using MQTT protocol](#)
 - [Manage](#)
 - [Manage enrollments - Portal](#)
 - [Configure verified CA certificates](#)
 - [Roll device certificates](#)
 - [Reprovision devices](#)
 - [Manage disenrollment](#)
 - [Manage deprovisioning](#)
 - [Configure IP filtering](#)
 - [Managing public network access](#)
 - [Monitor](#)
 - [Monitor Device Provisioning Service](#)
 - [Monitoring data reference](#)
 - [Provision IoT Edge devices](#)
 - [Linux](#)
 - [Windows](#)
 - [Troubleshooting and FAQ](#)
 - [Troubleshooting DPS](#)

DPS FAQ

Reference

[Libraries and SDKs](#)

[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 \(Management\)](#)

[Azure IoT SDK for C API documentation](#)

[REST API](#)

[Resource Manager template](#)

Resources

[Support and help options](#)

[IoT Glossary](#)

[Azure IoT services](#)

[IoT Hub](#)

[IoT Hub Device Provisioning Service](#)

[IoT Central](#)

[IoT Edge](#)

[Azure Maps](#)

[Time Series Insights](#)

[IoT device developer](#)

[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 Explorer tool](#)

[iothub-diagnostics tool](#)

[Pricing](#)

[Pricing calculator](#)

[Service updates](#)

[Technical case studies](#)

[Videos](#)

What is Azure IoT Hub Device Provisioning Service?

8/22/2022 • 10 minutes to read • [Edit Online](#)

Microsoft Azure provides a rich set of integrated public cloud services for all your IoT solution needs. The IoT Hub Device Provisioning Service (DPS) is a helper service for IoT Hub that enables zero-touch, just-in-time provisioning to the right IoT hub without requiring human intervention. DPS enables the provisioning of millions of devices in a secure and scalable manner.

When to use Device Provisioning Service

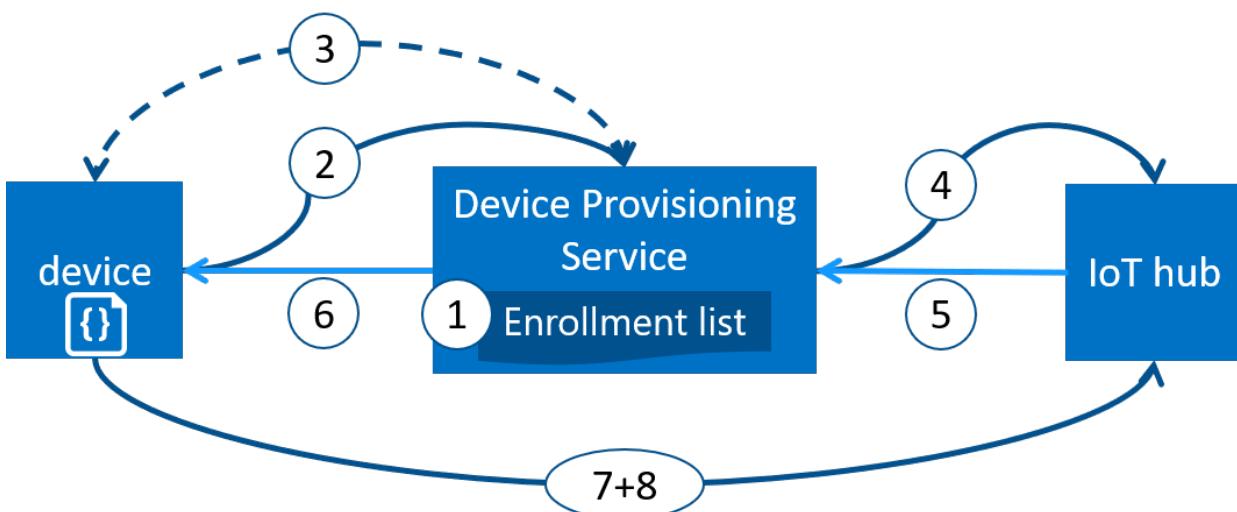
There are many provisioning scenarios in which DPS is an excellent choice for getting devices connected and configured to IoT Hub, such as:

- Zero-touch provisioning to a single IoT solution without hardcoding IoT Hub connection information at the factory (initial setup)
- Load-balancing devices across multiple hubs
- Connecting devices to their owner's IoT solution based on sales transaction data (multitenancy)
- Connecting devices to a particular IoT solution depending on use-case (solution isolation)
- Connecting a device to the IoT hub with the lowest latency (geo-sharding)
- Reprovisioning based on a change in the device
- Rolling the keys used by the device to connect to IoT Hub (when not using X.509 certificates to connect)

Provisioning of nested edge devices (parent/child hierarchies) is not currently supported by DPS.

Behind the scenes

All the scenarios listed in the previous section can be done using DPS for zero-touch provisioning with the same flow. Many of the manual steps traditionally involved in provisioning are automated with DPS to reduce the time to deploy IoT devices and lower the risk of manual error. The following section describes what goes on behind the scenes to get a device provisioned. The first step is manual, all of the following steps are automated.



1. Device manufacturer adds the device registration information to the enrollment list in the Azure portal.
2. Device contacts the DPS endpoint set at the factory. The device passes the identifying information to DPS to prove its identity.
3. DPS validates the identity of the device by validating the registration ID and key against the enrollment list

entry using either a nonce challenge ([Trusted Platform Module](#)) or standard X.509 verification (X.509).

4. DPS registers the device with an IoT hub and populates the device's [desired twin state](#).
5. The IoT hub returns device ID information to DPS.
6. DPS returns the IoT hub connection information to the device. The device can now start sending data directly to the IoT hub.
7. The device connects to IoT hub.
8. The device gets the desired state from its device twin in IoT hub.

Provisioning process

There are two distinct steps in the deployment process of a device in which DPS takes a part that can be done independently:

- The **manufacturing step** in which the device is created and prepared at the factory, and
- The **cloud setup step** in which the Device Provisioning Service is configured for automated provisioning.

Both these steps fit in seamlessly with existing manufacturing and deployment processes. DPS even simplifies some deployment processes that involve manual work to get connection information onto the device.

Manufacturing step

This step is all about what happens on the manufacturing line. The roles involved in this step include silicon designer, silicon manufacturer, integrator and/or the end manufacturer of the device. This step is concerned with creating the hardware itself.

DPS does not introduce a new step in the manufacturing process; rather, it ties into the existing step that installs the initial software and (ideally) the HSM on the device. Instead of creating a device ID in this step, the device is programmed with the provisioning service information, enabling it to call the provisioning service to get its connection info/IoT solution assignment when it is switched on.

Also in this step, the manufacturer supplies the device deployer/operator with identifying key information. Supplying that information could be as simple as confirming that all devices have an X.509 certificate generated from a signing certificate provided by the device deployer/operator, or as complicated as extracting the public portion of a TPM endorsement key from each TPM device. These services are offered by many silicon manufacturers today.

Cloud setup step

This step is about configuring the cloud for proper automatic provisioning. Generally there are two types of users involved in the cloud setup step: someone who knows how devices need to be initially set up (a device operator), and someone else who knows how devices are to be split among the IoT hubs (a solution operator).

There is a one-time initial setup of the provisioning that must occur, which is usually handled by the solution operator. Once the provisioning service is configured, it does not have to be modified unless the use case changes.

After the service has been configured for automatic provisioning, it must be prepared to enroll devices. This step is done by the device operator, who knows the desired configuration of the device(s) and is in charge of making sure the provisioning service can properly attest to the device's identity when it comes looking for its IoT hub. The device operator takes the identifying key information from the manufacturer and adds it to the enrollment list. There can be subsequent updates to the enrollment list as new entries are added or existing entries are updated with the latest information about the devices.

Registration and provisioning

Provisioning means various things depending on the industry in which the term is used. In the context of provisioning IoT devices to their cloud solution, provisioning is a two part process:

1. The first part is establishing the initial connection between the device and the IoT solution by registering the device.
2. The second part is applying the proper configuration to the device based on the specific requirements of the solution it was registered to.

Once both of those two steps have been completed, we can say that the device has been fully provisioned. Some cloud services only provide the first step of the provisioning process, registering devices to the IoT solution endpoint, but do not provide the initial configuration. DPS automates both steps to provide a seamless provisioning experience for the device.

Features of the Device Provisioning Service

DPS has many features, making it ideal for provisioning devices.

- **Secure attestation** support for both X.509 and TPM-based identities.
- **Enrollment list** containing the complete record of devices/groups of devices that may at some point register. The enrollment list contains information about the desired configuration of the device once it registers, and it can be updated at any time.
- **Multiple allocation policies** to control how DPS assigns devices to IoT hubs in support of your scenarios: Lowest latency, evenly weighted distribution (default), and static configuration via the enrollment list. Latency is determined using the same method as [Traffic Manager](#).
- **Monitoring and diagnostics logging** to make sure everything is working properly.
- **Multi-hub support** allows DPS to assign devices to more than one IoT hub. DPS can talk to hubs across multiple Azure subscriptions.
- **Cross-region support** allows DPS to assign devices to IoT hubs in other regions.
- **Encryption for data at rest** allows data in DPS to be encrypted and decrypted transparently using 256-bit AES encryption, one of the strongest block ciphers available, and is FIPS 140-2 compliant.

You can learn more about the concepts and features involved in device provisioning by reviewing the [DPS terminology](#) topic along with the other conceptual topics in the same section.

Cross-platform support

Just like all Azure IoT services, DPS works cross-platform with a variety of operating systems. Azure offers open-source SDKs in a variety of [languages](#) to facilitate connecting devices and managing the service. DPS supports the following protocols for connecting devices:

- HTTPS
- AMQP
- AMQP over web sockets
- MQTT
- MQTT over web sockets

DPS only supports HTTPS connections for service operations.

Regions

DPS is available in many regions. The list supported regions for all services is available at [Azure Regions](#). You can check availability of the Device Provisioning Service on the [Azure Status](#) page.

For resiliency and reliability, we recommend deploying to one of the regions that support [Availability Zones](#).

Data residency consideration

Device Provisioning Service doesn't store or process customer data outside of the geography where you deploy

the service instance. For more information, see [Cross-region replication in Azure](#).

However, by default, DPS uses the same [device provisioning endpoint](#) for all provisioning service instances, and performs traffic load balancing to the nearest available service endpoint. As a result, authentication secrets may be temporarily transferred outside of the region where the DPS instance was initially created. However, once the device is connected, the device data will flow directly to the original region of the DPS instance.

To ensure that your data doesn't leave the region that your DPS instance was created in, use a private endpoint. To learn how to set up private endpoints, see [Azure IoT Device Provisioning Service \(DPS\) support for virtual networks](#).

Quotas and Limits

Each Azure subscription has default quota limits in place that could impact the scope of your IoT solution. The current limit on a per-subscription basis is 10 Device Provisioning Services per subscription.

For more details on quota limits, see [Azure Subscription Service Limits](#).

NOTE

Some areas of this service have adjustable limits. This is represented in the tables below with the *Adjustable?* column.

When the limit can be adjusted, the *Adjustable?* value is *Yes*.

The actual value to which a limit can be adjusted may vary based on each customer's deployment. Multiple instances of DPS may be required for very large deployments.

If your business requires raising an adjustable limit or quota above the default limit, you can submit a request for additional resources by [opening a support ticket](#). Requesting an increase does not guarantee that it will be granted, as it needs to be reviewed on a case-by-case basis. Please contact Microsoft support as early as possible during your implementation, to be able to determine if your request could be approved and plan accordingly.

The following table lists the limits that apply to Azure IoT Hub Device Provisioning Service resources.

RESOURCE	LIMIT	ADJUSTABLE?
Maximum device provisioning services per Azure subscription	10	Yes
Maximum number of registrations	1,000,000	Yes
Maximum number of individual enrollments	1,000,000	Yes
Maximum number of enrollment groups (<i>X.509 certificate</i>)	100	Yes
Maximum number of enrollment groups (<i>symmetric key</i>)	100	No
Maximum number of CAs	25	No
Maximum number of linked IoT hubs	50	No
Maximum size of message	96 KB	No

TIP

If the hard limit on symmetric key enrollment groups is a blocking issue, it is recommended to use individual enrollments as a workaround.

The Device Provisioning Service has the following rate limits.

RATE	PER-UNIT VALUE	ADJUSTABLE?
Operations	200/min/service	Yes
Device registrations	200/min/service	Yes
Device polling operation	5/10 sec/device	No

Billable service operations and pricing

Each API call on DPS is billable as one *operation*. This includes all the service APIs and the device registration API.

The tables below show the current billable status for each DPS service API operation. To learn more about pricing for DPS, select **Pricing table** at the top of the [Azure IoT Hub pricing](#) page. Then select the **IoT Hub Device Provisioning Service** tab and the currency and region for your service.

API	OPERATION	BILLABLE?
Device API	Device Registration Status Lookup	No
Device API	Operation Status Lookup	No
Device API	Register Device	Yes
DPS Service API (registration state)	Delete	Yes
DPS Service API (registration state)	Get	Yes
DPS Service API (registration state)	Query	Yes
DPS Service API (enrollment group)	Create or Update	Yes
DPS Service API (enrollment group)	Delete	Yes
DPS Service API (enrollment group)	Get	Yes
DPS Service API (enrollment group)	Get Attestation Mechanism	Yes
DPS Service API (enrollment group)	Query	Yes
DPS Service API (enrollment group)	Run Bulk Operation	Yes
DPS Service API (individual enrollment)	Create or Update	Yes

API	OPERATION	BILLABLE?
DPS Service API (individual enrollment)	Delete	Yes
DPS Service API (individual enrollment)	Get	Yes
DPS Service API (individual enrollment)	Get Attestation Mechanism	Yes
DPS Service API (individual enrollment)	Query	Yes
DPS Service API (individual enrollment)	Run Bulk Operation	Yes
DPS Certificate API	Create or Update	No
DPS Certificate API	Delete	No
DPS Certificate API	Generate Verification Code	No
DPS Certificate API	Get	No
DPS Certificate API	List	No
DPS Certificate API	Verify Certificate	No
IoT DPS Resource API	Check Provisioning Service Name Availability	No
IoT DPS Resource API	Create or Update	No
IoT DPS Resource API	Delete	No
IoT DPS Resource API	Get	No
IoT DPS Resource API	Get Operation Result	No
IoT DPS Resource API	List By Resource Group	No
IoT DPS Resource API	List By Subscription	No
IoT DPS Resource API	List By Keys	No
IoT DPS Resource API	List Keys for Key Name	No
IoT DPS Resource API	List Valid SKUs	No
IoT DPS Resource API	Update	No

Related Azure components

DPS automates device provisioning with Azure IoT Hub. Learn more about [IoT Hub](#).

NOTE

Provisioning of nested edge devices (parent/child hierarchies) is not currently supported by DPS.

IoT Central applications use an internal DPS instance to manage device connections. To learn more, see:

- [How devices connect to IoT Central](#)
- [Tutorial: Create and connect a client application to your Azure IoT Central application](#)

Next steps

You now have an overview of provisioning IoT devices in Azure. The next step is to try out an end-to-end IoT scenario.

[Set up IoT Hub Device Provisioning Service with the Azure portal](#)

[Create and provision a simulated device](#)

Quickstart: Set up the IoT Hub Device Provisioning Service with the Azure portal

8/22/2022 • 6 minutes to read • [Edit Online](#)

The IoT Hub Device Provisioning Service enables zero-touch, just-in-time device provisioning to any IoT hub. The Device Provisioning Service enables customers to provision millions of IoT devices in a secure and scalable manner, without requiring human intervention. Azure IoT Hub Device Provisioning Service supports IoT devices with TPM, symmetric key, and X.509 certificate authentications. For more information, please refer to [IoT Hub Device Provisioning Service overview](#)

In this quickstart, you'll learn how to set up the IoT Hub Device Provisioning Service in the Azure portal.

To provision your devices, you will:

- Use the Azure portal to create an IoT Hub
- Use the Azure portal to create an IoT Hub Device Provisioning Service
- Link the IoT hub to the Device Provisioning Service

Prerequisites

You'll need an Azure subscription to begin with this article. You can create a [free account](#), if you haven't already.

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. On the Azure homepage, select the **+ Create a resource** button.
3. From the **Categories** menu, select **Internet of Things** then **IoT Hub**.
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, with a length between 3 and 50 alphanumeric characters. The name can also include the dash ('-') character.

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

Microsoft

Basics Networking Management 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 * ⓘ

Internal use

Resource group * ⓘ

[Create new](#)

Region * ⓘ

IoT hub name * ⓘ

Enter a name for your hub

[Review + create](#)

< Previous

Next: Networking >

Automation options

5. Select **Next: Networking** to continue creating your hub.

Choose the endpoints that devices can use to connect to your IoT hub. You can select the default setting, **Public access**, or choose **Private access**. Accept the default setting for this example.

Home > New >

IoT hub

Microsoft

Basics Networking Management Tags Review + create

You can connect to your IoT hub either publicly via its public hostname or privately using a private endpoint. [Learn more](#)

Connectivity configuration *

Public access

Private access (Recommended)

i You can change this or configure another connectivity method after this resource has been created. [Learn more](#)

[Review + create](#)

< Previous: Basics

Next: Management >

Automation options

6. Select **Next: Management** to continue creating your hub.

IoT hub

Microsoft

[Basics](#) [Networking](#) [Management](#) [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.

Defender for IoT

On

Turn on Defender 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		Cloud-to-device commands ⓘ	Enabled
Defender for IoT ⓘ	USD per device per month	IoT Edge ⓘ	Enabled
		Device management ⓘ	Enabled

Advanced settings

Scale

Device-to-cloud partitions ⓘ

4

[Review + create](#)

[< Previous: Networking](#)

[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, 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](#).
- **Microsoft Defender for IoT:** 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. Learn more about [security recommendations for IoT Hub in Defender for IoT](#).
- **Role-based access control:** Choose how access to the IoT hub is managed, whether shared access policies are allowed or only role-based access control is supported. For more information, see [Control access to IoT Hub by using Azure Active Directory](#).
- **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.

7. 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. In this document, you won't be adding any tags. For more information, see [Use tags to organize your Azure resources](#).

Home > New >

IoT hub

Microsoft

Basics Networking Management **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
	:	IoT Hub

Review + create < Previous: Management Next: Review + create > Automation options

8. 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.

IoT hub

Microsoft

[Basics](#) [Networking](#) [Management](#) [Tags](#) [Review + create](#)

Basics

Subscription	Internal use
Resource group	
Region	
IoT hub name	

Networking

Connectivity method	Public endpoint (all networks)
Private endpoint connections	None

Management

Pricing and scale tier	S1
Number of S1 IoT hub units	1
Messages per day	400,000
Device-to-cloud partitions	4
Cost per month	
Defender for IoT	See the Defender for IoT pricing
Minimum TLS Version	1.0

Tags

[Create](#)[< Previous: Tags](#)[Next >](#)[Automation options](#)

- Select **Create** to start the deployment of your new hub. Your deployment will be in progress a few minutes while the hub is being created. Once the deployment is complete, select **Go to resource** to open the new hub.

Create a new IoT Hub Device Provisioning Service

- In the Azure portal, select **+ Create a resource**.
- From the **Categories** menu, select **Internet of Things** then **IoT Hub Device Provisioning Service**.
- Select **Create**.
- Enter the following information:
 - Name:** Provide a unique name for your new Device Provisioning Service instance. If the name you enter is available, a green check mark appears.
 - Subscription:** Choose the subscription that you want to use to create this Device Provisioning Service instance.
 - Resource group:** This field allows you to create a new resource group, or choose an existing one to contain the new instance. Choose the same resource group that contains the IoT hub you created in the previous steps. By putting all related resources in a group together, you can manage

them together. For example, deleting the resource group deletes all resources contained in that group. For more information, see [Manage Azure Resource Manager resource groups](#).

- **Location:** Select a location that's close to your devices. For resiliency and reliability, we recommend deploying to one of the regions that support [Availability Zones](#).

The screenshot shows the 'IoT Hub device provisioning service' creation page. At the top, there are tabs for Basics, Networking, Management, Tags, and Review + create. The Basics tab is selected. Below the tabs, a description states: 'The IoT Hub device provisioning service is a helper service for IoT Hub that enables zero-touch, just-in-time provisioning to the right IoT hub without requiring human intervention, allowing customers to provision millions of devices in a secure and scalable manner.' A 'Learn more' link is provided. The 'Project details' section asks to choose a subscription and resource group. The 'Subscription' dropdown is set to 'Contoso-Subscription'. The 'Resource group' dropdown is set to 'Contoso-Resource', with a 'Create new' link below it. The 'Instance details' section asks for a name and region. The 'Name' dropdown is set to 'Contoso-DPS', which has a green checkmark. The 'Region' dropdown is set to 'East US'. At the bottom, there are buttons for 'Review + create', '< Previous', and 'Next: Networking >'.

5. Select **Review + Create** to validate your provisioning service.
6. Select **Create**.
7. After the deployment successfully completes, select **Go to resource** to view your Device Provisioning Service instance.

Link the IoT hub and your Device Provisioning Service

In this section, you'll add a configuration to the Device Provisioning Service instance. This configuration sets the IoT hub for which devices will be provisioned.

1. In the *Settings* menu, select **Linked IoT hubs**.
2. Select **+ Add**.
3. On the **Add link to IoT hub** panel, provide the following information:
 - **Subscription:** Select the subscription containing the IoT hub that you want to link with your new Device Provisioning Service instance.
 - **IoT hub:** Select the IoT hub to link with your new Device Provisioning Service instance.
 - **Access Policy:** Select **iothubowner** as the credentials for establishing the link with the IoT hub.

The screenshot shows the Azure portal interface for the Contoso-DPS Device Provisioning Service. On the left, the 'Linked IoT hubs' blade is open, with the 'Add' button highlighted by a red box. On the right, the 'Add link to IoT hub' dialog is displayed, also with its input fields highlighted by a red box. The dialog includes fields for Subscription (Contoso-Subscription), IoT hub (Contoso-IotHub-2), Access Policy (iothubowner), and basic hub properties like Hostname, Status, Pricing Tier, and Location.

4. Select **Save**.

5. Select **Refresh**. Now you should see the selected hub under the **Linked IoT hubs** blade.

Clean up resources

The rest of the Device Provisioning Service quickstarts and tutorials use the resources that you created in this quickstart. However, if you don't plan on doing any more quickstarts or tutorials, you'll want to delete those resources.

To clean up resources in the Azure portal:

1. From the left-hand menu in the Azure portal, select **All resources**.
2. Select your Device Provisioning Service.
3. At the top of the device detail pane, select **Delete**.
4. From the left-hand menu in the Azure portal, select **All resources**.
5. Select your IoT hub.
6. At the top of the hub detail pane, select **Delete**.

Next steps

Provision a simulated device with IoT hub and the Device Provisioning Service:

[Quickstart: Provision a simulated symmetric key device](#)

Quickstart: Set up the IoT Hub Device Provisioning Service with Azure CLI

8/22/2022 • 5 minutes to read • [Edit Online](#)

The Azure CLI is used to create and manage Azure resources from the command line or in scripts. This quickstart details using the Azure CLI to create an IoT hub and an IoT Hub Device Provisioning Service, and to link the two services together.

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

IMPORTANT

Both the IoT hub and the provisioning service you create in this quickstart will be publicly discoverable as DNS endpoints. Make sure to avoid any sensitive information if you decide to change the names used for these resources.

Prerequisites

- Use the Bash environment in [Azure Cloud Shell](#). For more information, see [Azure Cloud Shell Quickstart - Bash](#).



[Launch Cloud Shell](#)

- If you prefer to run CLI reference commands locally, [install](#) the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see [How to run the Azure CLI in a Docker container](#).
 - If you're using a local installation, sign in to the Azure CLI by using the [az login](#) command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see [Sign in with the Azure CLI](#).
 - When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see [Use extensions with the Azure CLI](#).
 - Run [az version](#) to find the version and dependent libraries that are installed. To upgrade to the latest version, run [az upgrade](#).

Create a resource group

Create a resource group with the [az group create](#) command. An Azure resource group is a logical container into which Azure resources are deployed and managed.

The following example creates a resource group named *my-sample-resource-group* in the *westus* location.

```
az group create --name my-sample-resource-group --location westus
```

TIP

The example creates the resource group in the West US location. You can view a list of available locations by running the command `az account list-locations -o table`.

Create an IoT hub

Create an IoT hub with the [az iot hub create](#) command.

The following example creates an IoT hub named *my-sample-hub* in the *westus* location. An IoT hub name must be globally unique in Azure, so you may want to add a unique prefix or suffix to the example name, or choose a new name altogether. Make sure your name follows proper naming conventions for an IoT hub: it should be 3-50 characters in length, and can contain only upper or lower case alphanumeric characters or hyphens ('-').

```
az iot hub create --name my-sample-hub --resource-group my-sample-resource-group --location westus
```

Create a Device Provisioning Service

Create a Device Provisioning Service with the [az iot dps create](#) command.

The following example creates a provisioning service named *my-sample-dps* in the *westus* location. You will also need to choose a globally unique name for your own provisioning service. Make sure it follows proper naming conventions for an IoT Hub Device Provisioning Service: it should be 3-64 characters in length and can contain only upper or lower case alphanumeric characters or hyphens ('-').

```
az iot dps create --name my-sample-dps --resource-group my-sample-resource-group --location westus
```

TIP

The example creates the provisioning service in the West US location. You can view a list of available locations by running the command

```
az provider show --namespace Microsoft.Devices --query "resourceTypes[?resourceType=='ProvisioningServices'].locations | [0]" --out table
```

or by going to the [Azure Status](#) page and searching for "Device Provisioning Service". In commands, locations can be specified either in one word or multi-word format; for example: *westus*, *West US*, *WEST US*, etc. The value is not case sensitive. If you use multi-word format to specify location, enclose the value in quotes; for example,

```
--location "West US".
```

For resiliency and reliability, we recommend deploying to one of the regions that support [Availability Zones](#).

Get the connection string for the IoT hub

You need your IoT hub's connection string to link it with the Device Provisioning Service. Use the [az iot hub show-connection-string](#) command to get the connection string and use its output to set a variable that you will use when you link the two resources.

The following example sets the *hubConnectionString* variable to the value of the connection string for the primary key of the hub's *iothubowner* policy (the `--policy-name` parameter can be used to specify a different policy). Trade out *my-sample-hub* for the unique IoT hub name you chose earlier. The command uses the Azure CLI [query](#) and [output](#) options to extract the connection string from the command output.

```
hubConnectionString=$(az iot hub show-connection-string --name my-sample-hub --key primary --query connectionString -o tsv)
```

You can use the `echo` command to see the connection string.

```
echo $hubConnectionString
```

NOTE

These two commands are valid for a host running under Bash.

If you're using a local Windows/CMD shell or a PowerShell host, modify the commands to use the correct syntax for that environment.

If you're using Azure Cloud Shell, check that the environment drop-down on the left side of the shell window says **Bash**.

Link the IoT hub and the provisioning service

Link the IoT hub and your provisioning service with the [az iot dps linked-hub create](#) command.

The following example links an IoT hub named *my-sample-hub* in the *westus* location and a Device Provisioning Service named *my-sample-dps*. Trade out these names for the unique IoT hub and Device Provisioning Service names you chose earlier. The command uses the connection string for your IoT hub that was stored in the *hubConnectionString* variable in the previous step.

```
az iot dps linked-hub create --dps-name my-sample-dps --resource-group my-sample-resource-group --connection-string $hubConnectionString --location westus
```

The command may take a few minutes to complete.

Verify the provisioning service

Get the details of your provisioning service with the [az iot dps show](#) command.

The following example gets the details of a provisioning service named *my-sample-dps*. Trade out this name for your own Device Provisioning Service name.

```
az iot dps show --name my-sample-dps
```

The linked IoT hub is shown in the *properties.iotHubs* collection.

```
user@Azure:~$ az iot dps show --name my-sample-dps
{
    "etag": "AAAAAAAGPDJY=",
    "id": "/subscriptions/<subscription ID>/resourceGroups/my-sample-resource-group/providers/Microsoft.Devices/provisioningServices/my-sample-dps",
    "location": "westus",
    "name": "my-sample-dps",
    "properties": {
        "allocationPolicy": "Hashed",
        "authorizationPolicies": null,
        "deviceProvisioningHostName": "global.azure-devices-provisioning.net",
        "idScope": "0ne000956C5",
        "iotHubs": [
            {
                "allocationWeight": null,
                "applyAllocationPolicy": null,
                "connectionString": "HostName=my-sample-hub.azure-devices.net;SharedAccessKeyName=iothubowner;SharedAccessKey=*****",
                "location": "westus",
                "name": "my-sample-hub.azure-devices.net"
            }
        ],
        "provisioningState": null,
        "serviceOperationsHostName": "my-sample-dps.azure-devices-provisioning.net",
        "state": "Active"
    },
    "resourcegroup": "my-sample-resource-group",
    "sku": {
        "capacity": 1,
        "name": "S1",
        "tier": "Standard"
    },
    "subscriptionid": "<subscription ID>",
    "tags": {},
    "type": "Microsoft.Devices/provisioningServices"
}
```

Clean up resources

Other quickstarts in this collection build upon this quickstart. If you plan to continue on to work with subsequent quickstarts or with the tutorials, do not clean up the resources created in this quickstart. If you do not plan to continue, you can use the following commands to delete the provisioning service, the IoT hub or the resource group and all of its resources. Replace the names of the resources written below with the names of your own resources.

To delete the provisioning service, run the [az iot dps delete](#) command:

```
az iot dps delete --name my-sample-dps --resource-group my-sample-resource-group
```

To delete the IoT hub, run the [az iot hub delete](#) command:

```
az iot hub delete --name my-sample-hub --resource-group my-sample-resource-group
```

To delete a resource group and all its resources, run the [az group delete](#) command:

```
az group delete --name my-sample-resource-group
```

Next steps

In this quickstart, you've deployed an IoT hub and a Device Provisioning Service instance, and linked the two

resources. To learn how to use this setup to provision a simulated device, continue to the quickstart for creating a simulated device.

[Quickstart to create a simulated device](#)

Quickstart: Set up the IoT Hub Device Provisioning Service (DPS) with Bicep

8/22/2022 • 5 minutes to read • [Edit Online](#)

You can use a [Bicep](#) file to programmatically set up the Azure cloud resources necessary for provisioning your devices. These steps show how to create an IoT hub and a new IoT Hub Device Provisioning Service instance with a Bicep file. The IoT Hub is also linked to the DPS resource using the Bicep file. This linking allows the DPS resource to assign devices to the hub based on allocation policies you configure.

[Bicep](#) is a domain-specific language (DSL) that uses declarative syntax to deploy Azure resources. It provides concise syntax, reliable type safety, and support for code reuse. Bicep offers the best authoring experience for your infrastructure-as-code solutions in Azure.

This quickstart uses [Azure PowerShell](#), and the [Azure CLI](#) to perform the programmatic steps necessary to create a resource group and deploy the Bicep file, but you can easily use .NET, Ruby, or other programming languages to perform these steps and deploy your Bicep file.

Prerequisites

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

- Use the Bash environment in [Azure Cloud Shell](#). For more information, see [Azure Cloud Shell Quickstart - Bash](#).

 [Launch Cloud Shell](#)

- If you prefer to run CLI reference commands locally, [install](#) the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see [How to run the Azure CLI in a Docker container](#).
 - If you're using a local installation, sign in to the Azure CLI by using the `az login` command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see [Sign in with the Azure CLI](#).
 - When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see [Use extensions with the Azure CLI](#).
 - Run `az version` to find the version and dependent libraries that are installed. To upgrade to the latest version, run `az upgrade`.
- If you choose to use Azure PowerShell locally:
 - [Install the Az PowerShell module](#).
 - Connect to your Azure account using the `Connect-AzAccount` cmdlet.
- If you choose to use Azure Cloud Shell:
 - See [Overview of Azure Cloud Shell](#) for more information.

Review the Bicep file

The Bicep file used in this quickstart is from [Azure Quickstart Templates](#).

NOTE

Currently there is no Bicep file support for creating enrollments with new DPS resources. This is a common and understood request that is being considered for implementation.

```
@description('Specify the name of the IoT hub.')
param iotHubName string

@description('Specify the name of the provisioning service.')
param provisioningServiceName string

@description('Specify the location of the resources.')
param location string = resourceGroup().location

@description('The SKU to use for the IoT Hub.')
param skuName string = 'S1'

@description('The number of IoT Hub units.')
param skuUnits int = 1

var iotHubKey = 'iothubowner'

resource iotHub 'Microsoft.Devices/IotHubs@2021-07-02' = {
    name: iotHubName
    location: location
    sku: {
        name: skuName
        capacity: skuUnits
    }
    properties: {}
}

resource provisioningService 'Microsoft.Devices/provisioningServices@2022-02-05' = {
    name: provisioningServiceName
    location: location
    sku: {
        name: skuName
        capacity: skuUnits
    }
    properties: {
        iotHubs: [
            {
                connectionString:
                    'HostName=${iotHub.properties.hostName};SharedAccessKeyName=${iotHubKey};SharedAccessKey=${iotHub.listkeys().value}'
                location: location
            }
        ]
    }
}
```

Two Azure resources are defined in the Bicep file above:

- **Microsoft.Devices/iothubs**: Creates a new Azure IoT Hub.
- **Microsoft.Devices/provisioningservices**: Creates a new Azure IoT Hub Device Provisioning Service with the new IoT Hub already linked to it.

Save a copy of the Bicep file locally as **main.bicep**.

Deploy the Bicep file

Sign in to your Azure account and select your subscription.

1. To log in Azure at the command prompt:

- [CLI](#)
- [PowerShell](#)

```
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.

- [CLI](#)
- [PowerShell](#)

```
az account list -o table
```

Use the following command to select the subscription that you want to use to run the commands to create your IoT hub and DPS resources. You can use either the subscription name or ID from the output of the previous command:

- [CLI](#)
- [PowerShell](#)

```
az account set --subscription {your subscription name or id}
```

3. Deploy the Bicep file with the following commands.

TIP

The commands will prompt for a resource group location. You can view a list of available locations by first running the command:

- [CLI](#)
- [PowerShell](#)

```
az account list-locations -o table
```

- [CLI](#)
- [PowerShell](#)

```
az group create --name exampleRG --location eastus
az deployment group create --resource-group exampleRG --template-file main.bicep --parameters
  iotHubName={IoT-Hub-name} provisioningServiceName={DPS-name}
```

Replace **{IoT-Hub-name}** with a globally unique IoT Hub name, replace **{DPS-name}** with a globally unique Device Provisioning Service (DPS) resource name.

It takes a few moments to create the resources.

Review deployed resources

1. To verify the deployment, run the following command and look for the new provisioning service and IoT hub in the output:

- [CLI](#)
- [PowerShell](#)

```
az resource list -g exampleRg
```

2. To verify that the hub is already linked to the DPS resource, run the following command.

- [CLI](#)
- [PowerShell](#)

```
az iot dps show --name <Your provisioningServiceName>
```

Clean up resources

Other quickstarts in this collection build upon this quickstart. If you plan to continue on to work with subsequent quickstarts or with the tutorials, do not clean up the resources created in this quickstart. If you do not plan to continue, you can use the Azure PowerShell or Azure CLI to delete the resource group and all of its resources.

To delete a resource group and all its resources from the Azure portal, just open the resource group and click **Delete resource group** and the top.

To delete the resource group deployed:

- [CLI](#)
- [PowerShell](#)

```
az group delete --name exampleRG
```

You can also delete resource groups and individual resources using the Azure portal, PowerShell, or REST APIs, as well as with supported platform SDKs published for Azure Resource Manager or IoT Hub Device Provisioning Service.

Next steps

In this quickstart, you've deployed an IoT hub and a Device Provisioning Service instance, and linked the two resources. To learn how to use this setup to provision a device, continue to the quickstart for creating a device.

[Quickstart: Provision a simulated symmetric key device](#)

Quickstart: Set up the IoT Hub Device Provisioning Service (DPS) with an ARM template

8/22/2022 • 7 minutes to read • [Edit Online](#)

You can use an [Azure Resource Manager](#) template (ARM template) to programmatically set up the Azure cloud resources necessary for provisioning your devices. These steps show how to create an IoT hub and a new IoT Hub Device Provisioning Service with an ARM template. The IoT Hub is also linked to the DPS resource using the template. This linking allows the DPS resource to assign devices to the hub based on allocation policies you configure.

An [ARM template](#) is a JavaScript Object Notation (JSON) file that defines the infrastructure and configuration for your project. The template uses declarative syntax. In declarative syntax, you describe your intended deployment without writing the sequence of programming commands to create the deployment.

This quickstart uses [Azure portal](#), and the [Azure CLI](#) to perform the programmatic steps necessary to create a resource group and deploy the template, but you can easily use the [PowerShell](#), .NET, Ruby, or other programming languages to perform these steps and deploy your template.

If your environment meets the prerequisites, and you're already familiar with using ARM templates, selecting the **Deploy to Azure** button below will open the template for deployment in the Azure portal.

 [Deploy to Azure](#)

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

Prerequisites

- Use the Bash environment in [Azure Cloud Shell](#). For more information, see [Azure Cloud Shell Quickstart - Bash](#).

 [Launch Cloud Shell](#)

- If you prefer to run CLI reference commands locally, [install](#) the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see [How to run the Azure CLI in a Docker container](#).
 - If you're using a local installation, sign in to the Azure CLI by using the [az login](#) command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see [Sign in with the Azure CLI](#).
 - When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see [Use extensions with the Azure CLI](#).
 - Run [az version](#) to find the version and dependent libraries that are installed. To upgrade to the latest version, run [az upgrade](#).

Review the template

The template used in this quickstart is from [Azure Quickstart Templates](#).

NOTE

Currently there is no ARM template support for creating enrollments with new DPS resources. This is a common and understood request that is being considered for implementation.

```
{  
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",  
    "contentVersion": "1.0.0.0",  
    "metadata": {  
        "_generator": {  
            "name": "bicep",  
            "version": "0.9.1.41621",  
            "templateHash": "17325722629306591414"  
        }  
    },  
    "parameters": {  
        "iotHubName": {  
            "type": "string",  
            "metadata": {  
                "description": "Specify the name of the IoT hub."  
            }  
        },  
        "provisioningServiceName": {  
            "type": "string",  
            "metadata": {  
                "description": "Specify the name of the provisioning service."  
            }  
        },  
        "location": {  
            "type": "string",  
            "defaultValue": "[resourceGroup().location]",  
            "metadata": {  
                "description": "Specify the location of the resources."  
            }  
        },  
        "skuName": {  
            "type": "string",  
            "defaultValue": "S1",  
            "metadata": {  
                "description": "The SKU to use for the IoT Hub."  
            }  
        },  
        "skuUnits": {  
            "type": "int",  
            "defaultValue": 1,  
            "metadata": {  
                "description": "The number of IoT Hub units."  
            }  
        }  
    },  
    "variables": {  
        "iotHubKey": "iothubowner"  
    },  
    "resources": [  
        {  
            "type": "Microsoft.Devices/IotHubs",  
            "apiVersion": "2021-07-02",  
            "name": "[parameters('iotHubName')]",  
            "location": "[parameters('location')]",  
            "sku": {  
                "name": "[parameters('skuName')]",  
                "capacity": "[parameters('skuUnits')]"  
            },  
            "properties": {}  
        },  
    ]  
}
```

```

    "type": "Microsoft.Devices/provisioningServices",
    "apiVersion": "2022-02-05",
    "name": "[parameters('provisioningServiceName')]",
    "location": "[parameters('location')]",
    "sku": {
        "name": "[parameters('skuName')]",
        "capacity": "[parameters('skuUnits')]"
    },
    "properties": {
        "iotHubs": [
            {
                "connectionString": "[format('HostName={0};SharedAccessKeyName={1};SharedAccessKey={2}', reference(resourceId('Microsoft.Devices/IotHubs', parameters('iotHubName'))).hostName, variables('iotHubKey'), listkeys(resourceId('Microsoft.Devices/IotHubs', parameters('iotHubName')), '2021-07-02').value)]",
                "location": "[parameters('location')]"
            }
        ],
        "dependsOn": [
            "[resourceId('Microsoft.Devices/IotHubs', parameters('iotHubName'))]"
        ]
    }
}
]
}

```

Two Azure resources are defined in the template above:

- **Microsoft.Devices/iothubs**: Creates a new Azure IoT Hub.
- **Microsoft.Devices/provisioningservices**: Creates a new Azure IoT Hub Device Provisioning Service with the new IoT Hub already linked to it.

Deploy the template

Deploy with the Portal

1. Select the following image to sign in to Azure and open the template for deployment. The template creates a new IoT Hub and DPS resource. The hub will be linked in the DPS resource.



[Deploy to Azure](#)

2. Select or enter the following values and click **Review + Create**.

Create an IoT Hub Device Provisioning Service

Azure quickstart template

Basics Review + create

Template

101-iothub-device-provisioning [2 resources] Edit template Edit parameters

Deployment scope

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

Subscription * <Select Your Subscription> (New) YourResourceGroupName Create new

Parameters

Region * East US

IoT Hub Name * EnterYourHubName

Provisioning Service Name * EnterYourDPSName

Location [resourceGroup().location]

Sku Name S1

Sku Units 1

Review + create

< Previous Next : Review + create >

The screenshot shows the Azure portal interface for creating an IoT Hub Device Provisioning Service. It's a 'Quickstart template'. The 'Subscription' section is highlighted with a red box, showing dropdown menus for selecting a subscription and creating a new resource group named 'YourResourceGroupName'. The 'Parameters' section is also highlighted with a red box, showing fields for Region (East US), IoT Hub Name ('EnterYourHubName'), Provisioning Service Name ('EnterYourDPSName'), Location (auto-filled as [resourceGroup().location]), Sku Name (S1), and Sku Units (1). The 'Review + create' button at the bottom left is also highlighted with a red box.

Unless it's specified below, use the default value to create the IoT Hub and DPS resource.

FIELD	DESCRIPTION
Subscription	Select your Azure subscription.
Resource group	Click Create new , and enter a unique name for the resource group, and then click OK .
Region	Select a region for your resources. For example, East US . For resiliency and reliability, we recommend deploying to one of the regions that support Availability Zones .
IoT Hub Name	Enter a name for the IoT Hub that must be globally unique within the <code>.azure-devices.net</code> namespace. You need the hub name in the next section when you validate the deployment.

FIELD	DESCRIPTION
Provisioning Service Name	Enter a name for the new Device Provisioning Service (DPS) resource. The name must be globally unique within the <i>.azure-devices-provisioning.net</i> namespace. You need the DPS name in the next section when you validate the deployment.

3. On the next screen, read the terms. If you agree to all terms, click **Create**.

The deployment will take a few moments to complete.

In addition to the Azure portal, you can also use the Azure PowerShell, Azure CLI, and REST API. To learn other deployment methods, see [Deploy templates](#).

Deploy with the Azure CLI

Using the Azure CLI requires version 2.6 or later. If you are running the Azure CLI locally, verify your version by running:

```
az --version
```

Sign in to your Azure account and select your subscription.

1. If you are running the Azure CLI locally instead of running it in the portal, you will need to log in. To log in 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 -o table
```

Use the following command to select subscription that you want to use to run the commands to create your IoT hub and DPS resources. 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}
```

3. Copy and paste the following commands into your CLI prompt. Then execute the commands by pressing **ENTER**.

TIP

The commands will prompt for a resource group location. You can view a list of available locations by first running the command:

```
az account list-locations -o table
```

```

read -p "Enter a project name that is used for generating resource names:" projectName &&
read -p "Enter the location (i.e. centralus):" location &&
templateUri="https://raw.githubusercontent.com/Azure/azure-quickstart-templates/master/quickstarts/microsoft.devices/iothub-device-provisioning/azuredploy.json" &&
resourceGroupName="${projectName}rg" &&
az group create --name $resourceGroupName --location "$location" &&
az deployment group create --resource-group $resourceGroupName --template-uri $templateUri &&
echo "Press [ENTER] to continue ..." &&
read

```

- The commands will prompt you for the following information. Provide each value and press **ENTER**.

PARAMETER	DESCRIPTION
Project name	The value of this parameter will be used to create a resource group to hold all resources. The string <code>rg</code> will be added to the end of the value for your resource group name.
location	This value is the region where all resources will reside.
iotHubName	Enter a name for the IoT Hub that must be globally unique within the <code>.azure-devices.net</code> namespace. You need the hub name in the next section when you validate the deployment.
provisioningServiceName	Enter a name for the new Device Provisioning Service (DPS) resource. The name must be globally unique within the <code>.azure-devices-provisioning.net</code> namespace. You need the DPS name in the next section when you validate the deployment.

The AzureCLI is used to deploy the template. In addition to the Azure CLI, you can also use the Azure PowerShell, Azure portal, and REST API. To learn other deployment methods, see [Deploy templates](#).

Review deployed resources

- To verify the deployment, run the following [command to list resources](#) and look for the new provisioning service and IoT hub in the output:

```
az resource list -g "${projectName}rg"
```

- To verify that the hub is already linked to the DPS resource, run the following [DPS extension show command](#).

```
az iot dps show --name <Your provisioningServiceName>
```

Notice the hubs that are linked on the `iotHubs` member.

Clean up resources

Other quickstarts in this collection build upon this quickstart. If you plan to continue on to work with subsequent quickstarts or with the tutorials, do not clean up the resources created in this quickstart. If you do not plan to continue, you can use the Azure portal or Azure CLI to delete the resource group and all of its resources.

To delete a resource group and all its resources from the Azure portal, just open the resource group and click **Delete resource group** and the top.

To delete the resource group deployed using the Azure CLI:

```
az group delete --name "${projectName}rg"
```

You can also delete resource groups and individual resources using the Azure portal, PowerShell, or REST APIs, as well as with supported platform SDKs published for Azure Resource Manager or IoT Hub Device Provisioning Service.

Next steps

In this quickstart, you've deployed an IoT hub and a Device Provisioning Service instance, and linked the two resources. To learn how to use this setup to provision a device, continue to the quickstart for creating a device.

[Quickstart: Provision a simulated symmetric key device](#)

Quickstart: Provision a simulated symmetric key device

8/22/2022 • 16 minutes to read • [Edit Online](#)

In this quickstart, you'll create a simulated device on your Windows machine. The simulated device will be configured to use the [symmetric key attestation](#) mechanism for authentication. After you've configured your device, you'll then provision it to your IoT hub using the Azure IoT Hub Device Provisioning Service.

If you're unfamiliar with the process of provisioning, review the [provisioning](#) overview.

This quickstart demonstrates a solution for a Windows-based workstation. However, you can also perform the procedures on Linux. For a Linux example, see [How to provision for multitenancy](#).

Prerequisites

- If you don't have an Azure subscription, create a [free account](#) before you begin.
- Complete the steps in [Set up IoT Hub Device Provisioning Service with the Azure portal](#).
- If you're using a Windows development environment, install [Visual Studio 2019](#) with the '[Desktop development with C++](#)' workload enabled. Visual Studio 2015 and Visual Studio 2017 are also supported. For Linux or macOS, see the appropriate section in [Prepare your development environment](#) in the SDK documentation.
- Install [.NET SDK 6.0](#) or later on your Windows-based machine. You can use the following command to check your version.

```
dotnet --info
```

- Install [Node.js v4.0+](#).
- Install [Python 3.7](#) or later installed on your Windows-based machine. You can check your version of Python by running `python --version`.
- Install [Java SE Development Kit 8](#) or later installed on your machine.
- Download and install [Maven](#).
- Install the latest version of [Git](#). Make sure that Git is added to the environment variables accessible to the command window. See [Software Freedom Conservancy's Git client tools](#) for the latest version of `git` tools to install, which includes *Git Bash*, the command-line app that you can use to interact with your local Git repository.

Prepare your development environment

In this section, you'll prepare a development environment that's used to build the [Azure IoT C SDK](#). The sample code attempts to provision the device, during the device's boot sequence.

1. Download the latest [CMake build system](#).

IMPORTANT

Confirm 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. Also, be aware that older versions of the CMake build system fail to generate the solution file used in this article. Make sure to use the latest version of CMake.

2. Open a web browser, and go to the [Release page of the Azure IoT C SDK](#).
3. Select the **Tags** tab at the top of the page.
4. Copy the tag name for the latest release of the Azure IoT C SDK.
5. 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(replace <release-tag> with the tag you copied in the previous step).

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

This operation could take several minutes to complete.

6. When the operation is complete, run the following commands from the `azure-iot-sdk-c` directory:

```
mkdir cmake
cd cmake
```

7. The code sample uses a symmetric key to provide attestation. Run the following command to build a version of the SDK specific to your development client platform that includes the device provisioning client:

```
cmake -Dhsm_type_symm_key:BOOL=ON -Duse_prov_client:BOOL=ON ..
```

TIP

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

8. When the build completes successfully, the last few output lines will look similar to the following output:

```
$ cmake -Dhsm_type_symm_key:BOOL=ON -Duse_prov_client:BOOL=ON ..
-- Building for: Visual Studio 16 2019
-- Selecting Windows SDK version 10.0.19041.0 to target Windows 10.0.19042.
-- The C compiler identification is MSVC 19.29.30040.0
-- The CXX compiler identification is MSVC 19.29.30040.0

...
-- Configuring done
-- Generating done
-- Build files have been written to: E:/IoT Testing/azure-iot-sdk-c/cmake
```

1. Open a Git CMD or Git Bash command-line environment.

2. Clone the [Azure IoT Samples for C# GitHub](#) repository using the following command:

```
git clone https://github.com/Azure-Samples/azure-iot-samples-csharp.git
```

1. Open a Git CMD or Git Bash command-line environment.
2. Clone the [Azure IoT SDK for Node.js GitHub](#) repository using the following command:

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

1. Open a Git CMD or Git Bash command-line environment.
2. Clone the [Azure IoT SDK for Python GitHub](#) repository using the following command:

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

1. Open a Git CMD or Git Bash command-line environment.
2. Clone the [Azure IoT SDK for Java GitHub](#) repository using the following command:

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

3. Go to the root `azure-iot-sdk-java` directory and build the project to download all needed packages. This step can take several minutes to complete.

```
cd azure-iot-sdk-java  
mvn install -DskipTests=true
```

Create a device enrollment

The Azure IoT Device Provisioning Service supports two types of enrollments:

- [Enrollment groups](#): Used to enroll multiple related devices.
- [Individual enrollments](#): Used to enroll a single device.

This article demonstrates an individual enrollment for a single device to be provisioned with an IoT hub.

1. Sign in to the [Azure portal](#).
2. On the left-hand menu or on the portal page, select **All resources**.
3. Select your Device Provisioning Service.
4. In the **Settings** menu, select **Manage enrollments**.
5. At the top of the page, select **+ Add individual enrollment**.
6. In the **Add Enrollment** page, enter the following information.
 - **Mechanism**: Select *Symmetric Key* as the identity attestation Mechanism.
 - **Auto-generate keys**: Check this box.
 - **Registration ID**: Enter a registration ID to identify the enrollment. The registration ID is a case-insensitive string (up to 128 characters long) of alphanumeric characters plus the special

characters: `'-'`, `'.'`, `'_'`, `':'`. The last character must be alphanumeric or dash (`'-'`). For example, `symm-key-device-007`.

- **IoT Hub Device ID:** Enter a device identifier. The device ID must comply with the [Device ID string requirements](#).

Add Enrollment

Mechanism * ⓘ

Symmetric Key

Auto-generate keys ⓘ

Primary Key ⓘ

Enter your primary key

Secondary Key ⓘ

Enter your secondary key

Registration ID *

symm-key-device-007

IoT Hub Device ID ⓘ

device-007

IoT Edge device ⓘ

True False

Select how you want to assign devices to hubs ⓘ

Evenly weighted distribution

Select the IoT hubs this device can be assigned to: ⓘ

Contoso-lotHub-2.azure-devices.net

[Link a new IoT hub](#)

[Save](#)

Home > test-dps-docs | Manage enrollments >

test-dps-docs | Manage enrollments

+ Add enrollment group + Add individual enrollment Refresh Delete

You can add or remove individual device enrollments and/or enrollment groups from this page.

Enrollment Groups Individual Enrollments

Search group enrollment by group name (name has to be exact match)

GROUP NAME

No results

Manage enrollments

Quick Start Shared access policies Linked IoT hubs Certificates Manage allocation policy IP Filter Properties Locks

Add Enrollment

Mechanism * ⓘ

Symmetric Key

Auto-generate keys ⓘ

Primary Key ⓘ

Enter your primary key

Secondary Key ⓘ

Enter your secondary key

Registration ID *

symm-key-csharp-device-01

IoT Hub Device ID ⓘ

csharp-device-01

IoT Edge device ⓘ

True False

Select how you want to assign devices to hubs ⓘ

Evenly weighted distribution

test-dps-docs | Manage enrollments

Device Provisioning Service

Search (Ctrl+ /)

+ Add enrollment group + Add individual enrollment Refresh Delete

Overview Activity log Access control (IAM) Tags Diagnose and solve problems Settings Quick Start Shared access policies Linked IoT hubs Certificates **Manage enrollments** Manage allocation policy IP Filter Properties Locks

You can add or remove individual device enrollments and/or enrollment groups from this page

Enrollment Groups Individual Enrollments

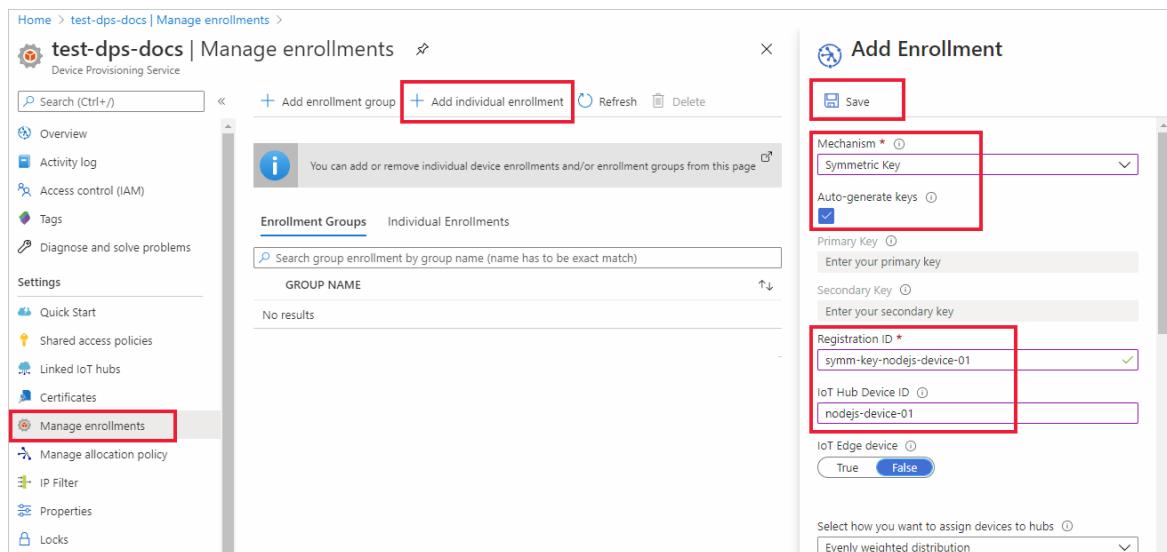
Search group enrollment by group name (name has to be exact match)

GROUP NAME

No results

Add Enrollment

Save Mechanism * Symmetric Key Auto-generate keys Primary Key Secondary Key Registration ID * symm-key-nodejs-device-01 IoT Hub Device ID nodejs-device-01 IoT Edge device True False Select how you want to assign devices to hubs Evenly weighted distribution



test-dps-docs | Manage enrollments

Device Provisioning Service

Search (Ctrl+ /)

+ Add enrollment group + Add individual enrollment Refresh Delete

Overview Activity log Access control (IAM) Tags Diagnose and solve problems Settings Quick Start Shared access policies Linked IoT hubs Certificates **Manage enrollments** Manage allocation policy IP Filter Properties Locks

You can add or remove individual device enrollments and/or enrollment groups from this page

Enrollment Groups Individual Enrollments

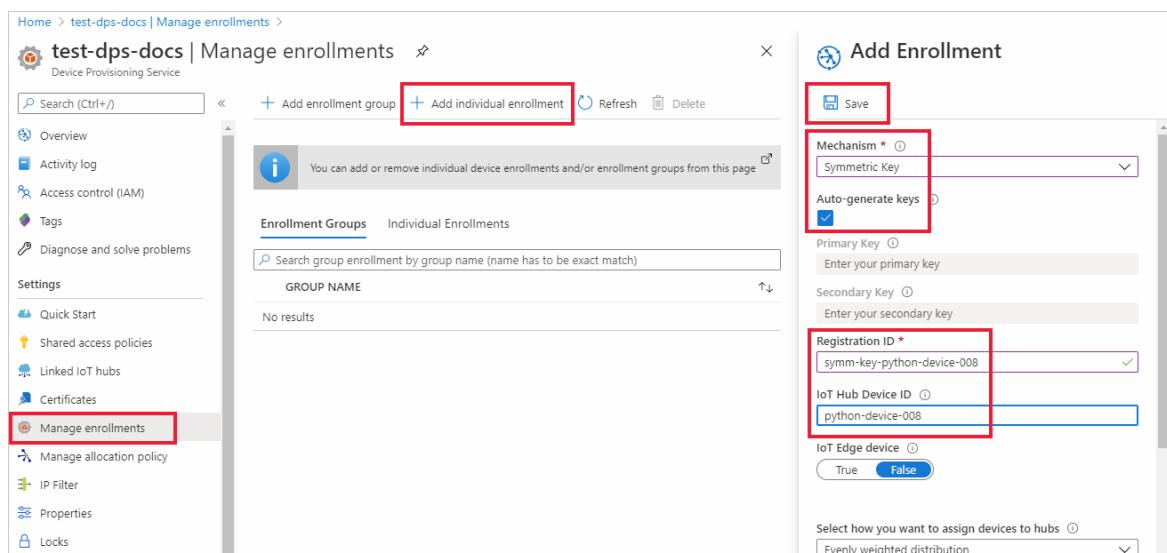
Search group enrollment by group name (name has to be exact match)

GROUP NAME

No results

Add Enrollment

Save Mechanism * Symmetric Key Auto-generate keys Primary Key Secondary Key Registration ID * symm-key-python-device-008 IoT Hub Device ID python-device-008 IoT Edge device True False Select how you want to assign devices to hubs Evenly weighted distribution

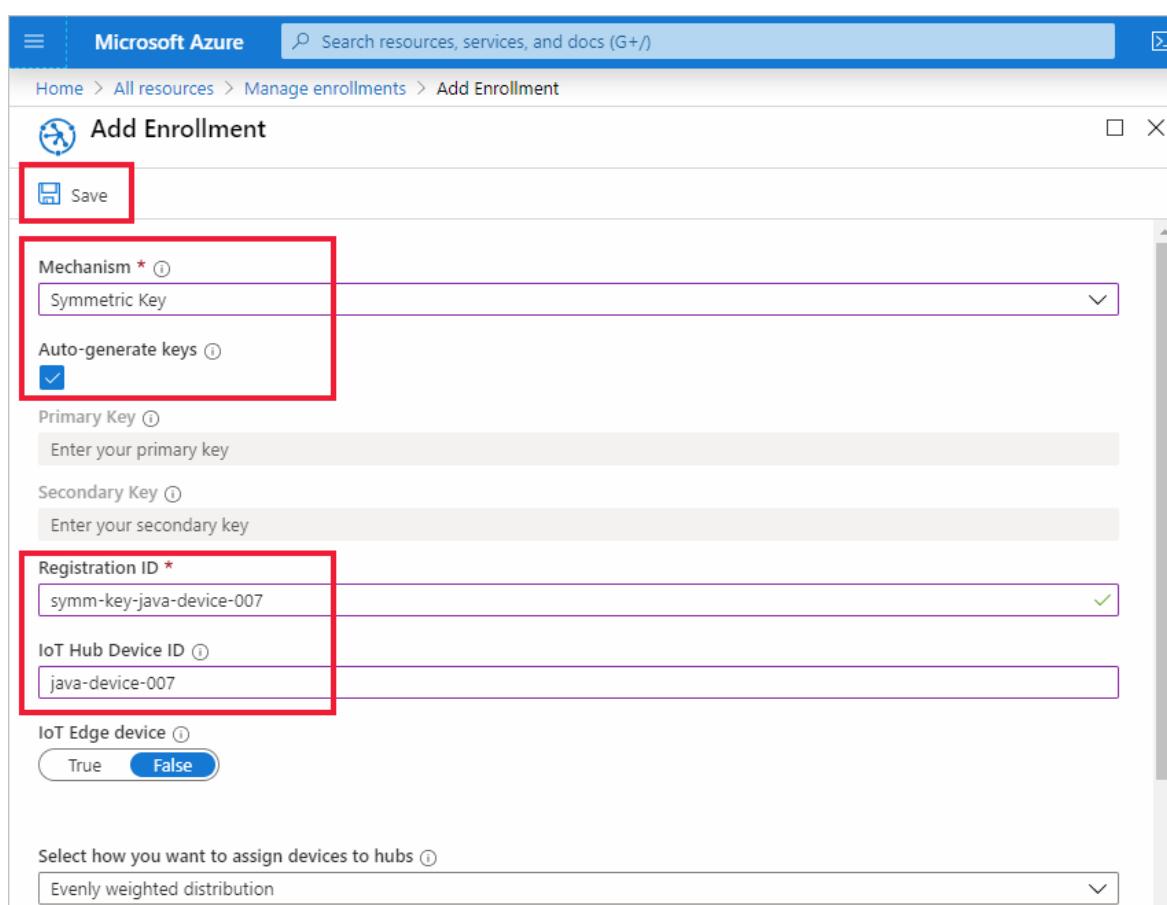


Microsoft Azure Search resources, services, and docs (G+/)

Home > All resources > Manage enrollments > Add Enrollment

Add Enrollment

Save Mechanism * Symmetric Key Auto-generate keys Primary Key Secondary Key Registration ID * symm-key-java-device-007 IoT Hub Device ID java-device-007 IoT Edge device True False Select how you want to assign devices to hubs Evenly weighted distribution



7. Select **Save**. A **Primary Key** and **Secondary Key** are generated and added to the enrollment entry, and you are taken back to the **Manage enrollments** page.
8. To view your simulated symmetric key device enrollment, select the **Individual Enrollments** tab.
9. Select your device (*symm-key-device-007*).
10. Copy the value of the generated **Primary Key**.

The screenshot shows the 'Enrollment Details' page for the device 'symm-key-device-007'. At the top, there's a save button, a refresh button, and a 'Regenerate keys' button. Below that is a note: 'You can view and update the enrollment details for an individual enrollment or remove the registration record for a previously provisioned device'. The 'Registration Status' section shows 'Status: unassigned', 'Assigned hub: -', 'Device ID: -', and 'Last assigned: -'. The 'Authentication Type' section shows 'Mechanism: Symmetric Key'. Under 'Primary Key', there is a long string of asterisks followed by a copy icon. Under 'Secondary Key', there is another long string of asterisks followed by a copy icon. At the bottom, there's an 'IoT Edge device' setting with a 'False' button highlighted in blue.

Prepare and run the device provisioning code

In this section, you'll update the device sample code to send the device's boot sequence to your Device Provisioning Service instance. This boot sequence will cause the device to be recognized, authenticated, and assigned to an IoT hub linked to the Device Provisioning Service instance.

The sample provisioning code accomplishes the following tasks, in order:

1. Authenticates your device with your Device Provisioning resource using the following three parameters:
 - The ID Scope of your Device Provisioning Service
 - The registration ID for your device enrollment.
 - The primary symmetric key for your device enrollment.
2. Assigns the device to the IoT hub already linked to your Device Provisioning Service instance.

To update and run the provisioning sample with your device information:

1. In the main menu of your Device Provisioning Service, select **Overview**.

2. Copy the **ID Scope** value.

The screenshot shows the Azure Device Provisioning Service overview page for a resource group named 'Contoso-Resource'. The 'ID Scope' field is highlighted with a red box and contains the value '0ne00364DC0'. Other visible details include the service endpoint 'Contoso-DPS.azure-devices-provisioning.net', global device endpoint 'global.azure-devices-provisioning.net', and pricing tier 'S1'.

3. In Visual Studio, open the *azure_iot_sdks.sln* solution file that was generated by running CMake. The solution file should be in the following location:

```
\azure-iot-sdk-c\cmake\azure_iot_sdks.sln
```

TIP

If the file was not generated in your cmake directory, make sure you used a recent version of the CMake build system.

4. In Visual Studio's *Solution Explorer* window, go to the **Provision_Samples** folder. Expand the sample project named **prov_dev_client_sample**. Expand **Source Files**, and open **prov_dev_client_sample.c**.

5. Find the `id_scope` constant, and replace the value with the **ID Scope** value that you copied in step 2.

```
static const char* id_scope = "0ne00002193";
```

6. Find the definition for the `main()` function in the same file. Make sure the `hsm_type` variable is set to `SECURE_DEVICE_TYPE_SYMMETRIC_KEY` as shown below:

```
SECURE_DEVICE_TYPE hsm_type;
//hsm_type = SECURE_DEVICE_TYPE TPM;
//hsm_type = SECURE_DEVICE_TYPE X509;
hsm_type = SECURE_DEVICE_TYPE SYMMETRIC KEY;
```

7. Find the call to `prov_dev_set_symmetric_key_info()` in `prov_dev_client_sample.c` that is commented out.

```
// Set the symmetric key if using they auth type  
//prov_dev_set_symmetric_key_info("<symm_registration_id>", "<symmetric_Key>");
```

Uncomment the function call, and replace the placeholder values (including the angle brackets) with your registration ID and the primary key value you copied earlier.

```
// Set the symmetric key if using they auth type  
prov_dev_set_symmetric_key_info("symm-key-device-007", "your primary key here");
```

8. Save the file.
9. Right-click the `prov_dev_client_sample` project and select **Set as Startup Project**.
10. On the Visual Studio menu, select **Debug > Start without debugging** to run the solution. In the rebuild the project prompt, select **Yes** to rebuild the project before running.

The following output is an example of the device successfully connecting to the provisioning Service instance to be assigned to an IoT hub:

```
Provisioning API Version: 1.2.8  
  
Registering Device  
  
Provisioning Status: PROV_DEVICE_REG_STATUS_CONNECTED  
Provisioning Status: PROV_DEVICE_REG_STATUS_ASSIGNING  
  
Registration Information received from service:  
test-docs-hub.azure-devices.net, deviceId: device-007  
Press enter key to exit:
```

The sample provisioning code accomplishes the following tasks:

1. Authenticates your device with your Device Provisioning resource using the following three parameters:
 - The ID Scope of your Device Provisioning Service
 - The registration ID for your device enrollment.
 - The primary symmetric key for your device enrollment.
2. Assigns the device to the IoT hub already linked to your Device Provisioning Service instance.
3. Sends a test telemetry message to the IoT hub.

To update and run the provisioning sample with your device information:

1. In the main menu of your Device Provisioning Service, select **Overview**.
2. Copy the **ID Scope** value.

The screenshot shows the Azure portal interface for the 'Contoso-DPS' Device Provisioning Service. On the left, there's a navigation menu with sections like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings (Quick Start, Shared access policies, Linked IoT hubs, Certificates, Manage enrollments, Manage allocation policy, Networking, Properties, Locks), Monitoring (Alerts, Metrics, Diagnostic settings, Logs), and Quick Links (Azure IoT Hub Device Provisioning Service Documentation, Learn more about IoT Hub Device Provisioning Service, Device Provisioning concepts, Pricing and scale details). The main content area displays service details: Resource group (Contoso-Resource), Status (Active), Location (East US), Subscription (Internal use - Aquent vendors), Subscription ID (xxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx), Tags (Click here to add tags), and Pricing and scale tier (S1). The 'ID Scope' field, which contains 'One00364DC0', is highlighted with a red box.

3. Open a command prompt and go to the *SymmetricKeySample* in the cloned samples repository:

```
cd azure-iot-samples-csharp\provisioning\Samples\device\SymmetricKeySample
```

4. In the *SymmetricKeySample* folder, open *Parameters.cs* in a text editor. This file shows the parameters that are supported by the sample. Only the first three required parameters will be used in this article when running the sample. Review the code in this file. No changes are needed.

PARAMETER	REQUIRED	DESCRIPTION
--s or --IdScope	True	The ID Scope of the DPS instance
--i or --Id	True	The registration ID when using individual enrollment, or the desired device ID when using group enrollment. The registration ID is a case-insensitive string (up to 128 characters long) of alphanumeric characters plus the special characters: '-' , '.' , '_' , ':' . The last character must be alphanumeric or dash ('-'). The device ID must comply with the Device ID string requirements .
--p or --PrimaryKey	True	The primary key of the individual or group enrollment.
--e or --EnrollmentType	False	The type of enrollment: Individual or Group . Defaults to Individual

PARAMETER	REQUIRED	DESCRIPTION
--g or --GlobalDeviceEndpoint	False	The global endpoint for devices to connect to. Defaults to global.azure-devices-provisioning.net
--t or --TransportType	False	The transport to use to communicate with the device provisioning instance. Defaults to Mqtt. Possible values include Mqtt, Mqtt_WebSocket_Only, Mqtt_Tcp_Only, Amqp, Amqp_WebSocket_Only, Amqp_Tcp_only, and Http1.

5. In the *SymmetricKeySample* folder, open *ProvisioningDeviceClientSample.cs* in a text editor. This file shows how the [SecurityProviderSymmetricKey](#) class is used along with the [ProvisioningDeviceClient](#) class to provision your simulated symmetric key device. Review the code in this file. No changes are needed.

6. Build and run the sample code using the following command:

- Replace <id-scope> with the ID Scope that you copied in step 2.
- Replace <registration-id> with the Registration ID that you copied from the device enrollment.
- Replace <primarykey> with the Primary Key that you copied from the device enrollment.

```
dotnet run --s <id-scope> --i <registration-id> --p <primarykey>
```

7. You should now see something similar to the following output. A "TestMessage" string is sent to the hub as a test message.

```
D:\azure-iot-samples-csharp\provisioning\Samples\device\SymmetricKeySample>dotnet run --s 0ne00000A0A
-i symm-key-csharp-device-01 --p
sbDDeEzRuEuGKag+kQKV+T1QGakRtHpsERLP0yPjwR93TrpEgEh/Y07CXstfha6dhIPWvdD1nRxK5T0KGKA+nQ==

Initializing the device provisioning client...
Initialized for registration Id symm-key-csharp-device-01.
Registering with the device provisioning service...
Registration status: Assigned.
Device csharp-device-01 registered to ExampleIoTHub.azure-devices.net.
Creating symmetric key authentication for IoT Hub...
Testing the provisioned device with IoT Hub...
Sending a telemetry message...
Finished.
Enter any key to exit.
```

The sample provisioning code accomplishes the following tasks, in order:

1. Authenticates your device with your Device Provisioning resource using the following four parameters:

- PROVISIONING_HOST
- PROVISIONING_IDSCOPE
- PROVISIONING_REGISTRATION_ID
- PROVISIONING_SYMMETRIC_KEY

2. Assigns the device to the IoT hub already linked to your Device Provisioning Service instance.

3. Sends a test telemetry message to the IoT hub.

To update and run the provisioning sample with your device information:

1. In the main menu of your Device Provisioning Service, select **Overview**.

2. Copy the **ID Scope** and **Global device endpoint** values.

The screenshot shows the Azure portal interface for a Device Provisioning Service named 'Contoso-DPS'. The left sidebar contains navigation links like Overview, Activity log, Access control (IAM), Tags, and Diagnose and solve problems. The main content area is titled 'Essentials' and includes fields for Resource group (Contoso-Resource), Status (Active), Location (East US), Subscription (Internal use - Aquent vendors), and Tags. On the right, there are sections for Service endpoint (Contoso-DPS.azure-devices-provisioning.net), Global device endpoint (global.azure-devices-provisioning.net), and ID Scope (One00364DC0). The 'Global device endpoint' and 'ID Scope' fields are both highlighted with red boxes.

3. Open a command prompt for executing Node.js commands, and go to the following directory:

```
cd azure-iot-sdk-node/provisioning/device/samples
```

4. In the *provisioning/device/samples* folder, open *register_symkey.js* and review the code. Notice that the sample code sets a custom payload:

```
provisioningClient.setProvisioningPayload({a: 'b'});
```

You may comment out this code, as it is not needed with for this quick start. A custom payload would be required you wanted to use a custom allocation function to assign your device to an IoT Hub. For more information, see [Tutorial: Use custom allocation policies](#).

The `provisioningClient.register()` method attempts the registration of your device.

No further changes are needed.

5. In the command prompt, run the following commands to set environment variables used by the sample:

- Replace `<provisioning-global-endpoint>` with the **Global device endpoint** that you copied in step 2.
- Replace `<id-scope>` with the **ID Scope** that you copied in step 2.
- Replace `<registration-id>` with the **Registration ID** that you copied from the device enrollment.
- Replace `<primarykey>` with the **Primary Key** that you copied from the device enrollment.

```
set PROVISIONING_HOST=<provisioning-global-endpoint>
```

```
set PROVISIONING_IDSCOPE=<id-scope>
```

```
set PROVISIONING_REGISTRATION_ID=<registration-id>
```

```
set PROVISIONING_SYMMETRIC_KEY=<primarykey>
```

6. Build and run the sample code using the following commands:

```
npm install
```

```
node register_symkey.js
```

7. You should now see something similar to the following output. A "Hello World" string is sent to the hub as a test message.

```
D:\azure-iot-samples-csharp\provisioning\Samples\device\SymmetricKeySample>dotnet run --s One00000A0A
--i symm-key-csharp-device-01 --p
sbDDeEzRuEuGKag+kQKV+T1QGakRtHpsERLP0yPjwR93TrpEgEh/Y07CXstfha6dhIPWvdD1nRxK5T0KGKA+nQ==

Initializing the device provisioning client...
Initialized for registration Id symm-key-csharp-device-01.
Registering with the device provisioning service...
Registration status: Assigned.
Device csharp-device-01 registered to ExampleIoTHub.azure-devices.net.
Creating symmetric key authentication for IoT Hub...
Testing the provisioned device with IoT Hub...
Sending a telemetry message...
Finished.

Enter any key to exit.
```

The sample provisioning code accomplishes the following tasks, in order:

1. Authenticates your device with your Device Provisioning resource using the following four parameters:

- PROVISIONING_HOST
- PROVISIONING_IDSCOPE
- PROVISIONING_REGISTRATION_ID
- PROVISIONING_SYMMETRIC_KEY

2. Assigns the device to the IoT hub already linked to your Device Provisioning Service instance.

3. Sends a test telemetry message to the IoT hub.

To update and run the provisioning sample with your device information:

1. In the main menu of your Device Provisioning Service, select **Overview**.
2. Copy the **ID Scope** and **Global device endpoint** values.

Service endpoint
Contoso-DPS.azure-devices-provisioning.net

Global device endpoint
global.azure-devices-provisioning.net

ID Scope
One00364DC0

3. Open a command prompt and go to the directory where the sample file, *provision_symmetric_key.py*, is located.

```
cd azure-iot-sdk-python\azure-iot-device\samples\async-hub-scenarios
```

4. In the command prompt, run the following commands to set environment variables used by the sample:

- Replace <provisioning-global-endpoint> with the **Global device endpoint** that you copied in step 2.
- Replace <id-scope> with the **ID Scope** that you copied in step 2.
- Replace <registration-id> with the **Registration ID** that you copied from the device enrollment.
- Replace <primarykey> with the **Primary Key** that you copied from the device enrollment.

```
set PROVISIONING_HOST=<provisioning-global-endpoint>
```

```
set PROVISIONING_IDSCOPE=<id-scope>
```

```
set PROVISIONING_REGISTRATION_ID=<registration-id>
```

```
set PROVISIONING_SYMMETRIC_KEY=<primarykey>
```

5. Install the *azure-iot-device* library by running the following command.

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

6. Run the Python sample code in *provision_symmetric_key.py*.

```
python provision_symmetric_key.py
```

7. You should now see something similar to the following output. Some example wind speed telemetry

messages are also sent to the hub as a test.

```
D:\azure-iot-sdk-python\azure-iot-device\samples\async-hub-scenarios>python
provision_symmetric_key.py
RegistrationStage(RequestAndResponseOperation): Op will transition into polling after interval 2.
Setting timer.
The complete registration result is
python-device-008
docs-test-iot-hub.azure-devices.net
initialAssignment
null
Will send telemetry from the provisioned device
sending message #8
sending message #9
sending message #3
sending message #10
sending message #4
sending message #2
sending message #6
sending message #7
sending message #1
sending message #5
done sending message #8
done sending message #9
done sending message #3
done sending message #10
done sending message #4
done sending message #2
done sending message #6
done sending message #7
done sending message #1
done sending message #5
```

The sample provisioning code accomplishes the following tasks, in order:

1. Authenticates your device with your Device Provisioning resource using the following four parameters:

- GLOBAL_ENDPOINT
- SCOPE_ID
- REGISTRATION_ID
- SYMMETRIC_KEY

2. Assigns the device to the IoT hub already linked to your Device Provisioning Service instance.

3. Sends a test telemetry message to the IoT hub.

To update and run the provisioning sample with your device information:

1. In the main menu of your Device Provisioning Service, select **Overview**.
2. Copy the **ID Scope** and **Global device endpoint** values. These are your SCOPE_ID and GLOBAL_ENDPOINT respectively.

The screenshot shows the Azure portal interface for the 'Contoso-DPS' Device Provisioning Service. On the left, there's a navigation sidebar with sections like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings, Quick Start, Shared access policies, Linked IoT hubs, Certificates, Manage enrollments, Manage allocation policy, Networking, Properties, Locks, Monitoring, and Alerts. The main content area has a header with 'Move', 'Delete', and 'Refresh' buttons, and links for 'View Cost' and 'JSON View'. Under the 'Essentials' section, it shows the Resource group (Contoso-Resource), Status (Active), Location (East US), Subscription (Internal use - Aquent vendors), Subscription ID (xxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx), and Tags (Click here to add tags). It also lists 'Service endpoint' (Contoso-DPS.azure-devices-provisioning.net) and 'Global device endpoint' (global.azure-devices-provisioning.net). The 'ID Scope' field (One00364DC0) is highlighted with a red box. Below this, it shows 'Pricing and scale tier' (S1). On the right, there's a 'Quick Links' section with three items: 'Azure IoT Hub Device Provisioning Service Documentation', 'Learn more about IoT Hub Device Provisioning Service', and 'Device Provisioning concepts', each with an 'Open' button.

3. Open the Java device sample code for editing. The full path to the device sample code is:

```
azure-iot-sdk-java/provisioning/provisioning-samples/provisioning-symmetrickey-individual-sample/src/main/java/samples/com/microsoft/azure/sdk/iot/ProvisioningSymmetricKeyIndividualEnrollmentSample.java
```

4. Set the value of the following variables for your DPS and device enrollment:

- Replace <id-scope> with the **ID Scope** that you copied in step 2.
- Replace <provisioning-global-endpoint> with the **Global device endpoint** that you copied in step 2.
- Replace <registration-id> with the **Registration ID** that you copied from the device enrollment.
- Replace <primarykey> with the **Primary Key** that you copied from the device enrollment.

```
private static final String SCOPE_ID = "<id-scope>";  
private static final String GLOBAL_ENDPOINT = "<provisioning-global-endpoint>";  
private static final String SYMMETRIC_KEY = "<primarykey>";  
private static final String REGISTRATION_ID = "<registration-id>";
```

5. Open a command prompt for building. Go to the provisioning sample project folder of the Java SDK repository.

```
cd azure-iot-sdk-java\provisioning\provisioning-samples\provisioning-symmetrickey-individual-sample
```

6. Build the sample.

```
mvn clean install
```

7. Go to the target folder and execute the created .jar file. In the java command, replace the {version} placeholder with the version in the .jar filename on your machine.

```
cd target  
java -jar ./provisioning-symmetrickey-individual-sample-{version}-with-deps.jar
```

8. You should now see something similar to the following output.

```
Starting...
Beginning setup.
Initialized a ProvisioningDeviceClient instance using SDK version 1.11.0
Starting provisioning thread...
Waiting for Provisioning Service to register
Opening the connection to device provisioning service...
Connection to device provisioning service opened successfully, sending initial device registration message
Authenticating with device provisioning service using symmetric key
Waiting for device provisioning service to provision this device...
Current provisioning status: ASSIGNING
Device provisioning service assigned the device successfully
IotHub Uri : <Your IoT hub name>.azure-devices.net
Device ID : java-device-007
Sending message from device to IoT Hub...
Press any key to exit...
Message received! Response status: OK_EMPTY
```

Confirm your device provisioning registration

1. Go to the [Azure portal](#).
2. On the left-hand menu or on the portal page, select **All resources**.
3. Select the IoT hub to which your device was assigned.
4. In the **Explorers** menu, select **IoT Devices**.
5. If your device was provisioned successfully, the device ID should appear in the list, with **Status** set as *enabled*. If you don't see your device, select **Refresh** at the top of the page.

The screenshot shows the Azure IoT Hub management interface for 'Contoso-IotHub-2'. The left sidebar includes links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Events, Settings (with Shared access policies, Identity, Pricing and scale, Networking, Certificates), Failover, Properties, Locks, Explorers (with Query explorer selected), and IoT devices (which is highlighted with a red box). The main content area displays a table of IoT devices:

Device ID	Status	Last Status Update	Authen...	Cloud ...
device-007	Enabled	--	Sas	0

Home > test-dps-docs > docs-test-iot-hub.azure-devices.net > docs-test-iot-hub

docs-test-iot-hub | IoT devices

IoT Hub

Search (Ctrl+ /) <> + New Refresh Delete

Settings

- Shared access policies
- Identity
- Pricing and scale
- Networking
- Certificates
- Built-in endpoints
- Failover
- Properties
- Locks

Explorers

- Query explorer
- IoT devices**

Automatic Device Management

IoT Edge

View, create, delete, and update devices in your IoT Hub.

Field Operator Value

+ X select or enter a property name = specify constraint value

+ Add a new clause

Query devices </> Switch to query editor

DEVICE ID	STATUS	LAST STATUS UPDATE (UTC)	AUTHENTICAT...	CLOUD ...
csharp-device-01	Enabled	--	Sas	0

Home > test-dps-docs > docs-test-iot-hub.azure-devices.net > docs-test-iot-hub

docs-test-iot-hub | IoT devices

IoT Hub

Search (Ctrl+ /) <> + New Refresh Delete

Settings

- Shared access policies
- Identity
- Pricing and scale
- Networking
- Certificates
- Built-in endpoints
- Failover
- Properties
- Locks

Explorers

- Query explorer
- IoT devices**

Automatic Device Management

IoT Edge

View, create, delete, and update devices in your IoT Hub.

Field Operator Value

+ X select or enter a property name = specify constraint value

+ Add a new clause

Query devices </> Switch to query editor

DEVICE ID	STATUS	LAST STATUS UPDATE (UTC)	AUTHENTICAT...	CLOUD ...
nodejs-device-01	Enabled	--	Sas	0

Home > test-dps-docs | Linked IoT hubs > docs-test-iot-hub.azure-devices.net >

docs-test-iot-hub | IoT devices

IoT Hub

Search (Ctrl+ /) < + New Refresh Delete

Pricing and scale Networking Certificates Built-in endpoints Failover Properties Locks Export template

Explorers Query explorer IoT devices

Automatic Device Management IoT Edge

View, create, delete, and update devices in your IoT Hub.

Field Operator Value
+ select or enter a property name = specify constraint value
+ Add a new clause

Query devices </> Switch to query editor

DEVICE ID	STATUS	LAST ST...	AUTHE...	CLOUD ...
python-device-008	Enabled	--	Sas	0

test-docs-dps - IoT devices

IoT Hub

Search (Ctrl+ /) < + New Refresh Delete

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 Export template

Explorers Query explorer IoT devices

View, create, delete, and update devices in your IoT Hub.

Field Operator Value
+ select or enter a property name = specify constraint
+ Add a new clause

Query devices

DEVICE ID	STATUS	LAST STATUS UPDATE (UTC)	AUTHENTICATION TYPE
java-device-007	Enabled	--	Sas

NOTE

If you changed the *initial device twin state* from the default value in the enrollment entry for your device, it can pull the desired twin state from the hub and act accordingly. For more information, see [Understand and use device twins in IoT Hub](#).

Clean up resources

If you plan to continue working on and exploring the device client sample, don't clean up the resources created in this quickstart. If you don't plan to continue, use the following steps to delete all resources created by this

quickstart.

Delete your device enrollment

1. Close the device client sample output window on your machine.
2. From the left-hand menu in the Azure portal, select **All resources**.
3. Select your Device Provisioning Service.
4. In the **Settings** menu, select **Manage enrollments**.
5. Select the **Individual Enrollments** tab.
6. Select the check box next to the *REGISTRATION ID* of the device you enrolled in this quickstart.
7. At the top of the page, select **Delete**.

Delete your device registration from IoT Hub

1. From the left-hand menu in the Azure portal, select **All resources**.
2. Select your IoT hub.
3. In the **Explorers** menu, select **IoT devices**.
4. Select the check box next to the *DEVICE ID* of the device you registered in this quickstart.
5. At the top of the page, select **Delete**.

Next steps

Provision an X.509 certificate device:

[Quickstart - Provision an X.509 device using the Azure IoT C SDK](#)

Quickstart: Provision an X.509 certificate simulated device

8/22/2022 • 25 minutes to read • [Edit Online](#)

In this quickstart, you'll create a simulated device on your Windows machine. The simulated device will be configured to use the [X.509 certificate attestation](#) mechanism for authentication. After you've configured your device, you'll then provision it to your IoT hub using the Azure IoT Hub Device Provisioning Service.

If you're unfamiliar with the process of provisioning, review the [provisioning overview](#). Also make sure you've completed the steps in [Set up IoT Hub Device Provisioning Service with the Azure portal](#) before continuing.

This quickstart demonstrates a solution for a Windows-based workstation. However, you can also perform the procedures on Linux. For a Linux example, see [How to provision for multitenancy](#).

Prerequisites

- If you don't have an Azure subscription, create a [free account](#) before you begin.
- Complete the steps in [Set up IoT Hub Device Provisioning Service with the Azure portal](#).

The following prerequisites are for a Windows development environment. For Linux or macOS, see the appropriate section in [Prepare your development environment](#) in the SDK documentation.

- Install [Visual Studio 2022](#) with the 'Desktop development with C++' workload enabled. Visual Studio 2015, Visual Studio 2017, and Visual Studio 19 are also supported. For Linux or macOS, see the appropriate section in [Prepare your development environment](#) in the SDK documentation.
- Install the latest [CMake build system](#). Make sure you check the option that adds the CMake executable to your path.

IMPORTANT

Confirm 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. Also, be aware that older versions of the CMake build system fail to generate the solution file used in this article. Make sure to use the latest version of CMake.

The following prerequisites are for a Windows development environment. For Linux or macOS, see the appropriate section in [Prepare your development environment](#) in the SDK documentation.

- Install [.NET SDK 6.0](#) or later on your Windows-based machine. You can use the following command to check your version.

```
dotnet --info
```

The following prerequisites are for a Windows development environment. For Linux or macOS, see the appropriate section in [Prepare your development environment](#) in the SDK documentation.

- Install [Node.js v4.0 or above](#) on your machine.

The following prerequisites are for a Windows development environment.

- Python 3.6 or later on your machine.

The following prerequisites are for a Windows development environment. For Linux or macOS, see the appropriate section in [Prepare your development environment](#) in the SDK documentation.

- Install the [Java SE Development Kit 8](#) or later on your machine.
- Download and install [Maven](#).
- Install the latest version of [Git](#). Make sure that Git is added to the environment variables accessible to the command window. See [Software Freedom Conservancy's Git client tools](#) for the latest version of `git` tools to install, which includes *Git Bash*, the command-line app that you can use to interact with your local Git repository.
- Make sure [OpenSSL](#) is installed on your machine. On Windows, your installation of Git includes an installation of OpenSSL. You can access OpenSSL from the Git Bash prompt. To verify that OpenSSL is installed, open a Git Bash prompt and enter `openssl version`.

NOTE

Unless you're familiar with OpenSSL and already have it installed on your Windows machine, we recommend using OpenSSL from the Git Bash prompt. Alternatively, you can choose to download the source code and build OpenSSL. To learn more, see the [OpenSSL Downloads](#) page. Or, you can download OpenSSL pre-built from a third-party. To learn more, see the [OpenSSL wiki](#). Microsoft makes no guarantees about the validity of packages downloaded from third-parties. If you do choose to build or download OpenSSL make sure that the OpenSSL binary is accessible in your path and that the `OPENSSL_CNF` environment variable is set to the path of your *openssl.cnf* file.

- Open both a Windows command prompt and a Git Bash prompt.

The steps in this quickstart assume that you're using a Windows machine and the OpenSSL installation that is installed as part of Git. You'll use the Git Bash prompt to issue OpenSSL commands and the Windows command prompt for everything else. If you're using Linux, you can issue all commands from a Bash shell.

Prepare your development environment

In this section, you'll prepare a development environment that's used to build the [Azure IoT C SDK](#). The sample code attempts to provision the device, during the device's boot sequence.

1. Open a web browser, and go to the [Release page of the Azure IoT C SDK](#).
2. Select the **Tags** tab at the top of the page.
3. Copy the tag name for the latest release of the Azure IoT C SDK.
4. In your Windows command prompt, run the following commands to clone the latest release of the [Azure IoT C SDK](#) GitHub repository. (replace `<release-tag>` with the tag you copied in the previous step).

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

This operation could take several minutes to complete.

5. When the operation is complete, run the following commands from the `azure-iot-sdk-c` directory:

```
mkdir cmake  
cd cmake
```

6. The code sample uses an X.509 certificate to provide attestation via X.509 authentication. Run the following command to build a version of the SDK specific to your development platform that includes the device provisioning client. A Visual Studio solution for the simulated device is generated in the `cmake` directory.

When specifying the path used with `-Dhsm_custom_lib` in the command below, make sure to use the absolute path to the library in the `cmake` directory you previously created. The path shown below assumes that you cloned the C SDK in the root directory of the C drive. If you used another directory, adjust the path accordingly.

```
cmake -Duse_prov_client:BOOL=ON -Dhsm_custom_lib=c:/azure-iot-sdk-  
c/cmake/provisioning_client/samples/custom_hsm_example/Debug/custom_hsm_example.lib ..
```

TIP

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

7. When the build succeeds, the last few output lines look similar to the following output:

```
cmake -Duse_prov_client:BOOL=ON -Dhsm_custom_lib=c:/azure-iot-sdk-  
c/cmake/provisioning_client/samples/custom_hsm_example/Debug/custom_hsm_example.lib ..  
-- Building for: Visual Studio 17 2022  
-- Selecting Windows SDK version 10.0.19041.0 to target Windows 10.0.22000.  
-- The C compiler identification is MSVC 19.32.31329.0  
-- The CXX compiler identification is MSVC 19.32.31329.0  
  
...  
  
-- Configuring done  
-- Generating done  
-- Build files have been written to: C:/azure-iot-sdk-c/cmake
```

1. In your Windows command prompt, clone the [Azure IoT Samples for C#](#) GitHub repository using the following command:

```
git clone https://github.com/Azure-Samples/azure-iot-samples-csharp.git
```

1. In your Windows command prompt, clone the [Azure IoT Samples for Node.js](#) GitHub repository using the following command:

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

1. In your Windows command prompt, clone the [Azure IoT Samples for Python](#) GitHub repository using the following command:

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

1. In your Windows command prompt, clone the [Azure IoT Samples for Java GitHub repository](#) using the following command:

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

2. Go to the root `azure-iot-sdk-java` directory and build the project to download all needed packages.

```
cd azure-iot-sdk-java  
mvn install -DskipTests=true
```

Create a self-signed X.509 device certificate

In this section, you'll use OpenSSL to create a self-signed X.509 certificate and a private key. This certificate will be uploaded to your provisioning service instance and verified by the service.

Caution

Use certificates created with OpenSSL in this quickstart for development testing only. Do not use these certificates in production. These certificates expire after 30 days and may contain hard-coded passwords, such as `1234`. To learn about obtaining certificates suitable for use in production, see [How to get an X.509 CA certificate](#) in the Azure IoT Hub documentation.

Perform the steps in this section in your Git Bash prompt.

1. In your Git Bash prompt, navigate to a directory where you'd like to create your certificates.
2. Run the following command:

- [Windows](#)
- [Linux](#)

```
winpty openssl req -outform PEM -x509 -sha256 -newkey rsa:4096 -keyout device-key.pem -out device-cert.pem -days 30 -extensions usr_cert -addext extendedKeyUsage=clientAuth -subj "//CN=my-x509-device"
```

IMPORTANT

The extra forward slash given for the subject name (`//CN=my-x509-device`) is only required to escape the string with Git on Windows platforms.

3. When asked to **Enter PEM pass phrase:**, use the pass phrase `1234`.
4. When asked **Verifying - Enter PEM pass phrase:**, use the pass phrase `1234` again.

A public key certificate file (`device-cert.pem`) and private key file (`device-key.pem`) should now be generated in the directory where you ran the `openssl` command.

The certificate file has its subject common name (CN) set to `my-x509-device`. For X.509-based enrollments, the [Registration ID](#) is set to the common name. The registration ID is a case-insensitive string of alphanumeric characters plus the special characters: `'-'`, `'.'`, `'_'`, `:`. The last character must be alphanumeric or dash (`'-'`). The common name must adhere to this format. DPS supports registration IDs up to 128 characters long; however, the maximum length of the subject common name in an X.509 certificate is 64 characters. The registration ID, therefore, is limited to 64 characters when using X.509 certificates.

5. The certificate file is Base64 encoded. To view the subject common name (CN) and other properties of the certificate file, enter the following command:

- [Windows](#)
- [Linux](#)

```
winpty openssl x509 -in device-cert.pem -text -noout
```

```
Certificate:  
Data:  
    Version: 3 (0x2)  
    Serial Number:  
        77:3e:1d:e4:7e:c8:40:14:08:c6:09:75:50:9c:1a:35:6e:19:52:e2  
    Signature Algorithm: sha256WithRSAEncryption  
    Issuer: CN = my-x509-device  
    Validity  
        Not Before: May  5 21:41:42 2022 GMT  
        Not After : Jun  4 21:41:42 2022 GMT  
    Subject: CN = my-x509-device  
    Subject Public Key Info:  
        Public Key Algorithm: rsaEncryption  
            RSA Public-Key: (4096 bit)  
                Modulus:  
                    00:d2:94:37:d6:1b:f7:43:b4:21:c6:08:1a:d6:d7:  
                    e6:40:44:4e:4d:24:41:6c:3e:8c:b2:2c:b0:23:29:  
                    ...  
                    23:6e:58:76:45:18:03:dc:2e:9d:3f:ac:a3:5c:1f:  
                    9f:66:b0:05:d5:1c:fe:69:de:a9:09:13:28:c6:85:  
                    0e:cd:53  
                Exponent: 65537 (0x10001)  
    X509v3 extensions:  
        X509v3 Basic Constraints:  
            CA:FALSE  
        Netscape Comment:  
            OpenSSL Generated Certificate  
        X509v3 Subject Key Identifier:  
            63:C0:B5:93:BF:29:F8:57:F8:F9:26:44:70:6F:9B:A4:C7:E3:75:18  
        X509v3 Authority Key Identifier:  
            keyid:63:C0:B5:93:BF:29:F8:57:F8:F9:26:44:70:6F:9B:A4:C7:E3:75:18  
  
    X509v3 Extended Key Usage:  
        TLS Web Client Authentication  
Signature Algorithm: sha256WithRSAEncryption  
82:8a:98:f8:47:00:85:be:21:15:64:b9:22:b0:13:cc:9e:9a:  
ed:f5:93:b9:4b:57:0f:79:85:9d:89:47:69:95:65:5e:b3:b1:  
...  
cc:b2:20:9a:b7:f2:5e:6b:81:a1:04:93:e9:2b:92:62:e0:1c:  
ac:d2:49:b9:36:d2:b0:21
```

6. The sample code requires a private key that isn't encrypted. Run the following command to create an unencrypted private key:

- [Windows](#)
- [Linux](#)

```
winpty openssl rsa -in device-key.pem -out unencrypted-device-key.pem
```

7. When asked to **Enter pass phrase for device-key.pem:**, use the same pass phrase you did previously,

1234 .

Keep the Git Bash prompt open. You'll need it later in this quickstart.

The C# sample code is set up to use X.509 certificates that are stored in a password-protected PKCS12 formatted file (`certificate.pfx`). You'll still need the PEM formatted public key certificate file (`device-cert.pem`) that you just created to create an individual enrollment entry later in this quickstart.

1. To generate the PKCS12 formatted file expected by the sample, enter the following command:

- [Windows](#)
- [Linux](#)

```
winpty openssl pkcs12 -inkey device-key.pem -in device-cert.pem -export -out certificate.pfx
```

2. When asked to **Enter pass phrase for device-key.pem:**, use the same pass phrase you did previously,

`1234`.

3. When asked to **Enter Export Password:**, use the password `1234`.

4. When asked **Verifying - Enter Export Password:**, use the password `1234` again.

A PKCS12 formatted certificate file (`certificate.pfx`) should now be generated in the directory where you ran the `openssl` command.

5. Copy the PKCS12 formatted certificate file to the project directory for the X.509 device provisioning sample. The path given is relative to the location where you downloaded the sample repo.

```
cp certificate.pfx ./azure-iot-samples-csharp/provisioning/Samples/device/X509Sample
```

You won't need the Git Bash prompt for the rest of this quickstart. However, you may want to keep it open to check your certificate if you have problems in later steps.

6. Copy the device certificate and private key to the project directory for the X.509 device provisioning sample. The path given is relative to the location where you downloaded the SDK.

```
cp device-cert.pem ./azure-iot-sdk-node/provisioning/device/samples
cp device-key.pem ./azure-iot-sdk-node/provisioning/device/samples
```

You won't need the Git Bash prompt for the rest of this quickstart. However, you may want to keep it open to check your certificate if you have problems in later steps.

6. Copy the device certificate and private key to the project directory for the X.509 device provisioning sample. The path given is relative to the location where you downloaded the SDK.

```
cp device-cert.pem ./azure-iot-sdk-python/azure-iot-device/samples/async-hub-scenarios
cp device-key.pem ./azure-iot-sdk-python/azure-iot-device/samples/async-hub-scenarios
```

You won't need the Git Bash prompt for the rest of this quickstart. However, you may want to keep it open to check your certificate if you have problems in later steps.

6. The Java sample code requires a private key that isn't encrypted. Run the following command to create an unencrypted private key:

- [Windows](#)
- [Linux](#)

```
winpty openssl pkey -in device-key.pem -out unencrypted-device-key.pem
```

7. When asked to **Enter pass phrase for device-key.pem:**, use the same pass phrase you did previously,

1234.

Keep the Git Bash prompt open. You'll need it later in this quickstart.

Create a device enrollment

The Azure IoT Device Provisioning Service supports two types of enrollments:

- **Enrollment groups:** Used to enroll multiple related devices.
- **Individual enrollments:** Used to enroll a single device.

This article demonstrates an individual enrollment for a single device to be provisioned with an IoT hub.

1. Sign in to the [Azure portal](#).
2. On the left-hand menu or on the portal page, select **All resources**.
3. Select your Device Provisioning Service.
4. In the **Settings** menu, select **Manage enrollments**.
5. At the top of the page, select **+ Add individual enrollment**.
6. In the **Add Enrollment** page, enter the following information.
 - **Mechanism:** Select X.509 as the identity attestation *Mechanism*.
 - **Primary certificate .pem or .cer file:** Choose **Select a file** and navigate to and select the certificate file, *device-cert.pem*, that you created in the previous section.
 - Leave **IoT Hub Device ID:** blank. Your device will be provisioned with its device ID set to the common name (CN) in the X.509 certificate, *my-x509-device*. This common name will also be the name used for the registration ID for the individual enrollment entry.
 - Optionally, you can provide the following information:
 - Select an IoT hub linked with your provisioning service.
 - Update the **Initial device twin state** with the desired initial configuration for the device.

 Save

Mechanism * ⓘ
X.509

Primary Certificate .pem or .cer file ⓘ
"device-cert.pem"
Clear Selection

Secondary Certificate .pem or .cer file ⓘ
Select a file
Clear Selection

IoT Hub Device ID ⓘ
Device ID

IoT Edge device ⓘ
True False

Select how you want to assign devices to hubs ⓘ
Evenly weighted distribution

Select the IoT hubs this device can be assigned to: ⓘ
contoso-iot-hub-2.azure-devices.net

[Link a new IoT hub](#)

7. Select **Save**. You'll be returned to **Manage enrollments**.
8. Select **Individual Enrollments**. Your X.509 enrollment entry, *my-x509-device*, should appear in the list.

Prepare and run the device provisioning code

In this section, you'll update the sample code to send the device's boot sequence to your Device Provisioning Service instance. This boot sequence will cause the device to be recognized and assigned to an IoT hub linked to the DPS instance.

In this section, you'll use your Git Bash prompt and the Visual Studio IDE.

Configure the provisioning device code

In this section, you update the sample code with your Device Provisioning Service instance information.

1. In the Azure portal, select the **Overview** tab for your Device Provisioning Service.
2. Copy the **ID Scope** value.

The screenshot shows the Azure IoT Hub Device Provisioning Service (DPS) Overview page. On the left, there's a navigation sidebar with sections like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings, Quick Start, Shared access policies, Linked IoT hubs, Certificates, Manage enrollments, Manage allocation policy, Networking, Properties, Locks, Monitoring, Alerts, Metrics, Diagnostic settings, and Logs. The main content area has tabs for Essentials, Resource group (move), Status, Location, Subscription (move), Tags (edit), and Quick Links. The 'ID Scope' field under the Essentials tab is highlighted with a red box and contains the value 'One005ED5E1'. Other details shown include Service endpoint, Global device endpoint, Pricing and scale tier (S1), and Automatic failover enabled (Yes).

3. Launch Visual Studio and open the new solution file that was created in the `cmake` directory you created in the root of the azure-iot-sdk-c git repository. The solution file is named `azure_iot_sdks.sln`.
4. In Solution Explorer for Visual Studio, navigate to **Provision_Samples > prov_dev_client_sample > Source Files** and open `prov_dev_client_sample.c`.
5. Find the `id_scope` constant, and replace the value with your **ID Scope** value that you copied in step 2.

```
static const char* id_scope = "One00000A0A";
```

6. Find the definition for the `main()` function in the same file. Make sure the `hsm_type` variable is set to `SECURE_DEVICE_TYPE_X509` as shown below.

```
SECURE_DEVICE_TYPE hsm_type;
//hsm_type = SECURE_DEVICE_TYPE TPM;
hsm_type = SECURE_DEVICE_TYPE_X509;
//hsm_type = SECURE_DEVICE_TYPE_SYMMETRIC_KEY;
```

7. Save your changes.
8. Right-click the `prov_dev_client_sample` project and select **Set as Startup Project**.

Configure the custom HSM stub code

The specifics of interacting with actual secure hardware-based storage vary depending on the hardware. As a result, the certificate and private key used by the simulated device in this quickstart will be hardcoded in the custom Hardware Security Module (HSM) stub code.

To update the custom HSM stub code to simulate the identity of the device with ID `my-x509-device`:

1. In Solution Explorer for Visual Studio, navigate to **Provision_Samples > custom_hsm_example > Source Files** and open `custom_hsm_example.c`.

2. Update the string value of the `COMMON_NAME` string constant using the common name you used when generating the device certificate, `my-x509-device`.

```
static const char* const COMMON_NAME = "my-x509-device";
```

3. Update the string value of the `CERTIFICATE` constant string using the device certificate, `device-cert.pem`, that you generated previously.

The syntax of certificate text in the sample must follow the pattern below with no extra spaces or parsing done by Visual Studio.

```
static const char* const CERTIFICATE = "-----BEGIN CERTIFICATE-----\n"
"MIIF0jCCAYKgAwIBAgIJAPzMa6s7mj7+MA0GCSqGSIb3DQEBCwUAMCoxKDAmBgNV\n"
...
"MDMwWhcNMjAxMTIyMjEzMDMwWjAqMSgwJgYDVQQDB9BenVyZSBjb1QgSHViIENB\n"
"-----END CERTIFICATE-----";
```

Updating this string value manually can be prone to error. To generate the proper syntax, you can copy and paste the following command into your **Git Bash prompt**, and press **ENTER**. This command will generate the syntax for the `CERTIFICATE` string constant value and write it to the output.

```
sed -e 's/^"/;$ !s/""/\n"/;$ s///' device-cert.pem
```

Copy and paste the output certificate text for the constant value.

4. Update the string value of the `PRIVATE_KEY` constant with the unencrypted private key for your device certificate, `unencrypted-device-key.pem`.

The syntax of the private key text must follow the pattern below with no extra spaces or parsing done by Visual Studio.

```
static const char* const PRIVATE_KEY = "-----BEGIN RSA PRIVATE KEY-----\n"
"MIJJJwIBAAKCAgEAtjkQjIhp0EE1PoADL1rFF/W6v4vlAz0SifKSQsaPeebqg8U\n"
...
"X7fi9OZ26QpnkS5QjjPTYI/wnn0J9YAwNfKS1NeXTJDfJ+KpjXBcvaLxeBQbQhij\n"
"-----END RSA PRIVATE KEY-----";
```

Updating this string value manually can be prone to error. To generate the proper syntax, you can copy and paste the following command into your **Git Bash prompt**, and press **ENTER**. This command will generate the syntax for the `PRIVATE_KEY` string constant value and write it to the output.

```
sed -e 's/^"/;$ !s/""/\n"/;$ s///' unencrypted-device-key.pem
```

Copy and paste the output private key text for the constant value.

5. Save your changes.
6. Right-click the `custom_hsm_-_example` project and select **Build**.

IMPORTANT

You must build the `custom_hsm_example` project before you build the rest of the solution in the next section.

Run the sample

1. On the Visual Studio menu, select **Debug > Start without debugging** to run the solution. If you're prompted to rebuild the project, select **Yes** to rebuild the project before running.

The following output is an example of the simulated device `my-x509-device` successfully booting up, and connecting to the provisioning service. The device is assigned to an IoT hub and registered:

```
Provisioning API Version: 1.8.0

Registering Device

Provisioning Status: PROV_DEVICE_REG_STATUS_CONNECTED
Provisioning Status: PROV_DEVICE_REG_STATUS_ASSIGNING

Registration Information received from service: contoso-iot-hub-2.azure-devices.net, deviceId: my-x509-device
Press enter key to exit:
```

In this section, you'll use your Windows command prompt.

1. In the Azure portal, select the **Overview** tab for your Device Provisioning Service.
2. Copy the **ID Scope** value.

The screenshot shows the Azure IoT Hub Device Provisioning Service (DPS) Overview page. The left sidebar lists various tabs: Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings, Quick Start, Shared access policies, Linked IoT hubs, Certificates, Manage enrollments, Manage allocation policy, Networking, Properties, Locks, Monitoring, Alerts, Metrics, Diagnostic settings, and Logs. The main content area has a header with 'Search (Ctrl+ /)', 'Move', 'Delete', and 'Refresh' buttons, and a 'JSON View' link. Under the 'Essentials' section, it shows the Resource group ([move](#)) as 'contoso-rg', Status as 'Active', Location as 'Central US', Subscription ([move](#)) as 'IoT Documentation', and Subscription ID. The 'ID Scope' field is highlighted with a red box and contains the value '0ne005ED5E1'. Other fields include Service endpoint ('contoso-dps-2.azure-devices-provisioning.net'), Global device endpoint ('global.azure-devices-provisioning.net'), Pricing and scale tier ('S1'), and Automatic failover enabled ('Yes'). Below this, there's a 'Tags' section with a link to 'Click here to add tags' and a 'Quick Links' section with links to 'Azure IoT Hub Device Provisioning Service Documentation', 'Learn more about Azure IoT Hub Device Provisioning Service', 'Device Provisioning concepts', and 'Pricing and scale details'.

3. In your Windows command prompt, change to the X509Sample directory. This is located in the `.\azure-iot-samples-csharp\provisioning\Samples\device\X509Sample` directory off the directory where you cloned the samples on your computer.
4. Enter the following command to build and run the X.509 device provisioning sample (replace the `<IDSscope>` value with the ID Scope that you copied in the previous section. The certificate file will default to `./certificate.pfx` and prompt for the .pfx password).

```
dotnet run -- -s <IDSscope>
```

If you want to pass the certificate and password as a parameter, you can use the following format.

NOTE

Additional parameters can be passed along while running the application to change the `TransportType (-t)` and the `GlobalDeviceEndpoint (-g)`.

```
dotnet run -- -s 0ne00000A0A -c certificate.pfx -p 1234
```

5. The device will connect to DPS and be assigned to an IoT hub. Then, the device will send a telemetry message to the IoT hub.

```
Loading the certificate...
Enter the PFX password for certificate.pfx:
*****
Found certificate: A33DB11B8883DEE5B1690ACFEAAB69E8E928080B CN=my-x509-device; PrivateKey: True
Using certificate A33DB11B8883DEE5B1690ACFEAAB69E8E928080B CN=my-x509-device
Initializing the device provisioning client...
Initialized for registration Id my-x509-device.
Registering with the device provisioning service...
Registration status: Assigned.
Device my-x509-device registered to MyExampleHub.azure-devices.net.
Creating X509 authentication for IoT Hub...
Testing the provisioned device with IoT Hub...
Sending a telemetry message...
Finished.
```

In this section, you'll use your Windows command prompt.

1. In the Azure portal, select the **Overview** tab for your Device Provisioning Service.
2. Copy the **ID Scope** and **Global device endpoint** values.

The screenshot shows the Azure IoT Hub Device Provisioning Service (DPS) Overview page. On the left, there's a navigation sidebar with sections like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings, Quick Start, Shared access policies, Linked IoT hubs, Certificates, Manage enrollments, Manage allocation policy, Networking, Properties, Locks, Monitoring, Alerts, Metrics, Diagnostic settings, and Logs. The main content area has tabs for Essentials, Resource group (move), Status, Location, Subscription (move), Subscription ID, Tags (edit), and Click here to add tags. Under the Essentials tab, it shows Service endpoint: contoso-dps-2.azure-devices-provisioning.net, Global device endpoint: global.azure-devices-provisioning.net, and ID Scope: One005ED5E1. These three fields are highlighted with red boxes. Other details shown include Status: Active, Location: Central US, Subscription tier: S1, and Automatic failover enabled: Yes. Below the main content is a section titled 'Quick Links' with links to Documentation, Learn more about DPS, Device Provisioning concepts, and Pricing and scale details.

3. In your Windows command prompt, go to the sample directory, and install the packages needed by the sample. The path shown is relative to the location where you cloned the SDK.

```
cd ./azure-iot-sdk-node/provisioning/device/samples  
npm install
```

4. Edit the `register_x509.js` file and make the following changes:

- Replace `provisioning host` with the **Global Device Endpoint** noted in **Step 1** above.
- Replace `id scope` with the **ID Scope** noted in **Step 1** above.
- Replace `registration id` with the **Registration ID** noted in the previous section.
- Replace `cert filename` and `key filename` with the files you generated previously, `device-cert.pem` and `device-key.pem`.

5. Save the file.

6. Run the sample and verify that the device was provisioned successfully.

```
node register_x509.js
```

TIP

The Azure IoT Hub Node.js Device SDK provides an easy way to simulate a device. For more information, see [Device concepts](#).

In this section, you'll use your Windows command prompt.

1. In the Azure portal, select the **Overview** tab for your Device Provisioning Service.
2. Copy the **ID Scope** and **Global device endpoint** values.

- In your Windows command prompt, go to the directory of the [provision_x509.py](#) sample. The path shown is relative to the location where you cloned the SDK.

```
cd ./azure-iot-sdk-python/azure-iot-device/samples/async-hub-scenarios
```

This sample uses six environment variables to authenticate and provision an IoT device using DPS. These environment variables are:

VARIABLE NAME	DESCRIPTION
PROVISIONING_HOST	This value is the global endpoint used for connecting to your DPS resource
PROVISIONING_IDSCOPE	This value is the ID Scope for your DPS resource
DPS_X509_REGISTRATION_ID	This value is the ID for your device. It must also match the subject name on the device certificate
X509_CERT_FILE	Your device certificate filename
X509_KEY_FILE	The private key filename for your device certificate
PASS_PHRASE	The pass phrase you used to encrypt the certificate and private key file (1234).

- Add the environment variables for the global device endpoint and ID Scope.

```
set PROVISIONING_HOST=global.azure-devices-provisioning.net
set PROVISIONING_IDSCOPE=<ID scope for your DPS resource>
```

5. The registration ID for the IoT device must match subject name on its device certificate. If you generated a self-signed test certificate, `my-x509-device` is both the subject name and the registration ID for the device.
6. Set the environment variable for the registration ID as follows:

```
set DPS_X509_REGISTRATION_ID=my-x509-device
```

7. Set the environment variables for the certificate file, private key file, and pass phrase.

```
set X509_CERT_FILE=../device-cert.pem
set X509_KEY_FILE=../device-key.pem
set PASS_PHRASE=1234
```

8. Review the code for [provision_x509.py](#). If you're not using **Python version 3.7** or later, make the [code change mentioned here](#) to replace `asyncio.run(main())`.
9. Save your changes.
10. Run the sample. The sample will connect to DPS, which will provision the device to an IoT hub. After the device is provisioned, the sample will send some test messages to the IoT hub.

```
$ python azure-iot-sdk-python/azure-iot-device/samples/async-hub-scenarios/provision_x509.py
RegistrationStage(RequestAndResponseOperation): Op will transition into polling after interval 2.
Setting timer.
The complete registration result is
my-x509-device
TestHub12345.azure-devices.net
initialAssignment
null
Will send telemetry from the provisioned device
sending message #4
sending message #7
sending message #2
sending message #8
sending message #5
sending message #9
sending message #1
sending message #6
sending message #10
sending message #3
done sending message #4
done sending message #7
done sending message #2
done sending message #8
done sending message #5
done sending message #9
done sending message #1
done sending message #6
done sending message #10
done sending message #3
```

In this section, you'll use both your Windows command prompt and your Git Bash prompt.

1. In the Azure portal, select the **Overview** tab for your Device Provisioning Service.
2. Copy the **ID Scope** and **Global device endpoint** values.

The screenshot shows the Azure IoT Hub Device Provisioning Service (DPS) blade. In the 'Essentials' section, several key parameters are listed:

- Resource group ([move](#)) **contoso-rg**
- Status **Active**
- Location **Central US**
- Subscription ([move](#)) **IoT Documentation**
- Subscription ID
- Tags ([edit](#)) [Click here to add tags](#)
- Service endpoint **contoso-dps-2.azure-devices-provisioning.net**
- Global device endpoint **global.azure-devices-provisioning.net**
- ID Scope **One005ED5E1**
- Pricing and scale tier **S1**
- Automatic failover enabled **Yes**

3. In your Windows command prompt, navigate to the sample project folder. The path shown is relative to the location where you cloned the SDK

```
cd ..\azure-iot-sdk-java\provisioning\provisioning-samples\provisioning-X509-sample
```

4. Enter the provisioning service and X.509 identity information in the sample code. This is used during provisioning, for attestation of the simulated device, prior to device registration.
 - a. Open the file `..\src\main\java\samples\com\microsoft\azure\ sdk\iot\ ProvisioningX509Sample.java` in your favorite editor.
 - b. Update the following values with the **ID Scope** and **Provisioning Service Global Endpoint** that you copied previously.

```
private static final String idScope = "[Your ID scope here]";
private static final String globalEndpoint = "[Your Provisioning Service Global Endpoint here]";
private static final ProvisioningDeviceClientTransportProtocol
PROVISIONING_DEVICE_CLIENT_TRANSPORT_PROTOCOL =
ProvisioningDeviceClientTransportProtocol.HTTPS;
```

- c. Update the value of the `leafPublicPem` constant string with the value of your certificate, `device-cert.pem`.

The syntax of certificate text must follow the pattern below with no extra spaces or characters.

```
private static final String leafPublicPem = "-----BEGIN CERTIFICATE-----\n" +  
"MIIF0jCCAyKgAwIBAgIJAPzMa6s7mj7+MA0GCSqGSIb3DQEBCwUAMCoxKDAmBgNV\n" +  
"..."  
"MDMwWhcNMjAxMTIyMjEzMFMwWjAqMSgwJgYDVQQDDB9BenVyZSBJb1QgSHViIENB\n" +  
"-----END CERTIFICATE-----";
```

Updating this string value manually can be prone to error. To generate the proper syntax, you can copy and paste the following command into your **Git Bash prompt**, and press **ENTER**. This command will generate the syntax for the `leafPublicPem` string constant value and write it to the output.

```
sed 's/^"/;$ !s/$/\n" +;$ s$/"/' device-cert.pem
```

Copy and paste the output certificate text for the constant value.

- d. Update the string value of the `leafPrivateKey` constant with the unencrypted private key for your device certificate, *unencrypted-device-key.pem*.

The syntax of the private key text must follow the pattern below with no extra spaces or characters.

```
private static final String leafPrivateKey = "-----BEGIN PRIVATE KEY-----\n" +  
"MIJJwIBAAKCAgEAtjvKQjIhp0EE1PoADL1rfF/W6v4vlAzOSifKSQsaPeebqg8U\n" +  
"..."  
"X7fi9OZ26QpnkS5QjjPTYI/wn0J9YAwNfKS1NeXTJDFJ+KpjXBcvaxLxeBQbQhij\n" +  
"-----END PRIVATE KEY-----";
```

Updating this string value manually can be prone to error. To generate the proper syntax, you can copy and paste the following command into your **Git Bash prompt**, and press **ENTER**. This command will generate the syntax for the `leafPrivateKey` string constant value and write it to the output.

```
sed 's/^"/;$ !s/$/\n" +;$ s$/"/' unencrypted-device-key.pem
```

Copy and paste the output private key text for the constant value.

- e. Save your changes.
5. Build the sample, and then go to the `target` folder.

```
mvn clean install  
cd target
```

6. The build outputs jar file in the `target` folder with the following file format:

```
provisioning-x509-sample-{version}-with-deps.jar ; for example:  
provisioning-x509-sample-1.8.1-with-deps.jar . Execute the jar file. You may need to replace the version  
in the command below.
```

```
java -jar ./provisioning-x509-sample-1.8.1-with-deps.jar
```

The sample will connect to DPS, which will provision the device to an IoT hub. After the device is provisioned, the sample will send some test messages to the IoT hub.

```
Starting...
```

```
Beginning setup.
WARNING: sun.reflect.Reflection.getCallerClass is not supported. This will impact performance.
2022-05-11 09:42:05,025 DEBUG (main)
[com.microsoft.azure.sdk.iot.provisioning.device.ProvisioningDeviceClient] - Initialized a ProvisioningDeviceClient instance using SDK version 2.0.0
2022-05-11 09:42:05,027 DEBUG (main)
[com.microsoft.azure.sdk.iot.provisioning.device.ProvisioningDeviceClient] - Starting provisioning thread...
Waiting for Provisioning Service to register
2022-05-11 09:42:05,030 INFO (global.azure-devices-provisioning.net-6255a8ba-CxnPendingConnectionId-azure-iot-sdk-ProvisioningTask)
[com.microsoft.azure.sdk.iot.provisioning.device.internal.task.ProvisioningTask] - Opening the connection to device provisioning service...
2022-05-11 09:42:05,252 INFO (global.azure-devices-provisioning.net-6255a8ba-Cxn6255a8ba-azure-iot-sdk-ProvisioningTask)
[com.microsoft.azure.sdk.iot.provisioning.device.internal.task.ProvisioningTask] - Connection to device provisioning service opened successfully, sending initial device registration message
2022-05-11 09:42:05,286 INFO (global.azure-devices-provisioning.net-6255a8ba-Cxn6255a8ba-azure-iot-sdk-RegisterTask) [com.microsoft.azure.sdk.iot.provisioning.device.internal.task.RegisterTask] - Authenticating with device provisioning service using x509 certificates
2022-05-11 09:42:06,083 INFO (global.azure-devices-provisioning.net-6255a8ba-Cxn6255a8ba-azure-iot-sdk-ProvisioningTask)
[com.microsoft.azure.sdk.iot.provisioning.device.internal.task.ProvisioningTask] - Waiting for device provisioning service to provision this device...
2022-05-11 09:42:06,083 INFO (global.azure-devices-provisioning.net-6255a8ba-Cxn6255a8ba-azure-iot-sdk-ProvisioningTask)
[com.microsoft.azure.sdk.iot.provisioning.device.internal.task.ProvisioningTask] - Current provisioning status: ASSIGNING
Waiting for Provisioning Service to register
2022-05-11 09:42:15,685 INFO (global.azure-devices-provisioning.net-6255a8ba-Cxn6255a8ba-azure-iot-sdk-ProvisioningTask)
[com.microsoft.azure.sdk.iot.provisioning.device.internal.task.ProvisioningTask] - Device provisioning service assigned the device successfully
IoTHub Uri : MyExampleHub.azure-devices.net
Device ID : java-device-01
2022-05-11 09:42:25,057 INFO (main)
[com.microsoft.azure.sdk.iot.device.transport.ExponentialBackoffWithJitter] - NOTE: A new instance of ExponentialBackoffWithJitter has been created with the following properties. Retry Count: 2147483647, Min Backoff Interval: 100, Max Backoff Interval: 10000, Max Time Between Retries: 100, Fast Retry Enabled: true
2022-05-11 09:42:25,080 INFO (main)
[com.microsoft.azure.sdk.iot.device.transport.ExponentialBackoffWithJitter] - NOTE: A new instance of ExponentialBackoffWithJitter has been created with the following properties. Retry Count: 2147483647, Min Backoff Interval: 100, Max Backoff Interval: 10000, Max Time Between Retries: 100, Fast Retry Enabled: true
2022-05-11 09:42:25,087 DEBUG (main) [com.microsoft.azure.sdk.iot.device.DeviceClient] - Initialized a DeviceClient instance using SDK version 2.0.3
2022-05-11 09:42:25,129 DEBUG (main)
[com.microsoft.azure.sdk.iot.device.transport.mqtt.MqttIotHubConnection] - Opening MQTT connection...
2022-05-11 09:42:25,150 DEBUG (main) [com.microsoft.azure.sdk.iot.device.transport.mqtt.Mqtt] - Sending MQTT CONNECT packet...
2022-05-11 09:42:25,982 DEBUG (main) [com.microsoft.azure.sdk.iot.device.transport.mqtt.Mqtt] - Sent MQTT CONNECT packet was acknowledged
2022-05-11 09:42:25,983 DEBUG (main) [com.microsoft.azure.sdk.iot.device.transport.mqtt.Mqtt] - Sending MQTT SUBSCRIBE packet for topic devices/java-device-01/messages/devicebound/#
2022-05-11 09:42:26,068 DEBUG (main) [com.microsoft.azure.sdk.iot.device.transport.mqtt.Mqtt] - Sent MQTT SUBSCRIBE packet for topic devices/java-device-01/messages/devicebound/# was acknowledged
2022-05-11 09:42:26,068 DEBUG (main)
[com.microsoft.azure.sdk.iot.device.transport.mqtt.MqttIotHubConnection] - MQTT connection opened successfully
2022-05-11 09:42:26,070 DEBUG (main) [com.microsoft.azure.sdk.iot.device.transport.IotHubTransport] - The connection to the IoT Hub has been established
2022-05-11 09:42:26,071 DEBUG (main) [com.microsoft.azure.sdk.iot.device.transport.IotHubTransport] - Updating transport status to new status CONNECTED with reason CONNECTION_OK
2022-05-11 09:42:26,071 DEBUG (main) [com.microsoft.azure.sdk.iot.device.DeviceIO] - Starting worker threads
2022-05-11 09:42:26,073 DEBUG (main) [com.microsoft.azure.sdk.iot.device.transport.IotHubTransport] - Invoking connection status callbacks with new status details
2022-05-11 09:42:26,074 DEBUG (main) [com.microsoft.azure.sdk.iot.device.transport.IotHubTransport] -
```

```
Client connection opened successfully
2022-05-11 09:42:26,075 INFO (main) [com.microsoft.azure.sdk.iot.device.DeviceClient] - Device client
opened successfully
Sending message from device to IoT Hub...
2022-05-11 09:42:26,077 DEBUG (main) [com.microsoft.azure.sdk.iot.device.transport.IotHubTransport] -
Message was queued to be sent later ( Message details: Correlation Id [54d9c6b5-3da9-49fe-9343-
caa6864f9a02] Message Id [28069a3d-f6be-4274-a48b-1ee539524eef] )
Press any key to exit...
2022-05-11 09:42:26,079 DEBUG (MyExampleHub.azure-devices.net-java-device-01-ee6c362d-Cxn7a1fb819-
e46d-4658-9b03-ca50c88c0440-azure-iot-sdk-IotHubSendTask)
[com.microsoft.azure.sdk.iot.device.transport.IotHubTransport] - Sending message ( Message details:
Correlation Id [54d9c6b5-3da9-49fe-9343-caa6864f9a02] Message Id [28069a3d-f6be-4274-a48b-
1ee539524eef] )
2022-05-11 09:42:26,422 DEBUG (MQTT Call: java-device-01)
[com.microsoft.azure.sdk.iot.device.transport.IotHubTransport] - IoTHub message was acknowledged.
Checking if there is record of sending this message ( Message details: Correlation Id [54d9c6b5-3da9-
49fe-9343-caa6864f9a02] Message Id [28069a3d-f6be-4274-a48b-1ee539524eef] )
2022-05-11 09:42:26,425 DEBUG (MyExampleHub.azure-devices.net-java-device-01-ee6c362d-Cxn7a1fb819-
e46d-4658-9b03-ca50c88c0440-azure-iot-sdk-IotHubSendTask)
[com.microsoft.azure.sdk.iot.device.transport.IotHubTransport] - Invoking the callback function for
sent message, IoT Hub responded to message ( Message details: Correlation Id [54d9c6b5-3da9-49fe-
9343-caa6864f9a02] Message Id [28069a3d-f6be-4274-a48b-1ee539524eef] ) with status OK
Message sent!
```

Confirm your device provisioning registration

To see which IoT hub your device was provisioned to, examine the registration details of the individual enrollment you created previously:

1. In Azure portal, go to your Device Provisioning Service.
2. In the **Settings** menu, select **Manage enrollments**.
3. Select **Individual Enrollments**. The X.509 enrollment entry that you created previously, *my-x509-device*, should appear in the list.
4. Select the enrollment entry. The IoT hub that your device was assigned to and its device ID appears under **Registration Status**.

my-x509-device ...

Enrollment Details

Save Refresh

Registration Status

Status: assigned

Assigned hub: contoso-iot-hub-2.azure-devices.net

Device ID: my-x509-device

Last assigned: Mon May 16 2022 15:32:32 GMT-0700 (Pacific Daylight Time)

Delete Registration

Authentication Type

Mechanism: X.509

Primary Certificate

Delete current certificate ⓘ

Primary Certificate Thumbprint
C9161FB0B860821E15F905BAFF9CDD7C5561CF2

Primary Certificate .pem or .cer file ⓘ

Select a file

Clear Selection

To verify the device on your IoT hub:

1. In Azure portal, go to the IoT hub that your device was assigned to.
2. In the **Device management** menu, select **Devices**.
3. If your device was provisioned successfully, its device ID, *my-x509-device*, should appear in the list, with **Status** set as *enabled*. If you don't see your device, select **Refresh**.

contoso-iot-hub-2 | Devices ⌂ ...

IoT Hub

Search (Ctrl+ /) <> View, create, delete, and update devices in your IoT Hub.

Overview Activity log Access control (IAM) Tags Diagnose and solve problems Events Pricing and scale

Device management Devices IoT Edge Configurations Updates Queries Hub settings Built-in endpoints

Device name enter device ID Find devices Find using a query

Add Device Refresh Delete

Device ID	Status	Last Status Update	Authentication ...	Cloud ...
my-x509-device	Enabled	--	SelfSigned	0

IMPORTANT

If you changed the *initial device twin state* from the default value in the enrollment entry for your device, it can pull the desired twin state from the hub and act accordingly. For more information, see [Understand and use device twins in IoT Hub](#)

Clean up resources

If you plan to continue working on and exploring the device client sample, don't clean up the resources created in this quickstart. If you don't plan to continue, use the following steps to delete all resources created by this quickstart.

Delete your device enrollment

1. Close the device client sample output window on your machine.
2. From the left-hand menu in the Azure portal, select **All resources**.
3. Select your Device Provisioning Service.
4. In the **Settings** menu, select **Manage enrollments**.
5. Select the **Individual Enrollments** tab.
6. Select the check box next to the *REGISTRATION ID* of the device you enrolled in this quickstart.
7. At the top of the page, select **Delete**.

Delete your device registration from IoT Hub

1. From the left-hand menu in the Azure portal, select **All resources**.
2. Select your IoT hub.
3. In the **Explorers** menu, select **IoT devices**.
4. Select the check box next to the *DEVICE ID* of the device you registered in this quickstart.
5. At the top of the page, select **Delete**.

Next steps

To learn how to enroll your X.509 device programmatically:

[Azure quickstart - Enroll X.509 devices to Azure IoT Hub Device Provisioning Service](#)

Quickstart: Provision a simulated TPM device

8/22/2022 • 16 minutes to read • [Edit Online](#)

In this quickstart, you'll create a simulated device on your Windows machine. The simulated device will be configured to use a [Trusted Platform Module \(TPM\) attestation](#) mechanism for authentication. After you've configured your device, you'll provision it to your IoT hub using the Azure IoT Hub Device Provisioning Service. Sample code will then be used to help enroll the device with a Device Provisioning Service instance.

If you're unfamiliar with the process of provisioning, review the [provisioning overview](#). Also make sure you've completed the steps in [Set up IoT Hub Device Provisioning Service with the Azure portal](#) before continuing.

The Azure IoT Device Provisioning Service supports two types of enrollments:

- [Enrollment groups](#) that are used to enroll multiple related devices.
- [Individual Enrollments](#) that are used to enroll a single device.

This article demonstrates individual enrollments.

Trusted Platform Module (TPM) attestation isn't supported in the Python SDK. With Python, you can provision a device using [symmetric keys](#) or [X.509 certificates](#).

Prerequisites

- If you don't have an Azure subscription, create a [free account](#) before you begin.
- Complete the steps in [Set up IoT Hub Device Provisioning Service with the Azure portal](#).

The following prerequisites are for a Windows development environment. For Linux or macOS, see the appropriate section in [Prepare your development environment](#) in the SDK documentation.

- [Visual Studio 2019](#) with the '[Desktop development with C++](#)' workload enabled. Visual Studio 2015 and Visual Studio 2017 are also supported.
- A TPM 2.0 hardware security module on your Windows-based machine.
- Install [.NET Core SDK 6.0](#) or later on your Windows-based machine. You can use the following command to check your version.

```
dotnet --info
```

- Install [Node.js v4.0+](#).
- Install [Java SE Development Kit 8](#) or later installed on your machine.
- Download and install [Maven](#).
- Install the latest version of [Git](#). Make sure that Git is added to the environment variables accessible to the command window. See [Software Freedom Conservancy's Git client tools](#) for the latest version of `git` tools to install, which includes *Git Bash*, the command-line app that you can use to interact with your local Git repository.

Prepare your development environment

In this section, you'll prepare a development environment used to build the [Azure IoT C SDK](#) and the [TPM device simulator](#) sample.

1. Download the latest [CMake build system](#).

IMPORTANT

Confirm 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. Also, be aware that older versions of the CMake build system fail to generate the solution file used in this article. Make sure to use the latest version of CMake.

2. Open a web browser, and go to the [Release page of the Azure IoT C SDK](#).
3. Select the **Tags** tab at the top of the page.
4. Copy the tag name for the latest release of the Azure IoT C SDK.
5. 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. (replace `<release-tag>` with the tag you copied in the previous step).

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

This operation could take several minutes to complete.

6. When the operation is complete, run the following commands from the `azure-iot-sdk-c` directory:

```
mkdir cmake
cd cmake
```

1. Open a Git CMD or Git Bash command-line environment.
2. Clone the [Azure IoT Samples for C#](#) GitHub repository using the following command:

```
git clone https://github.com/Azure-Samples/azure-iot-samples-csharp.git
```

1. Open a Git CMD or Git Bash command-line environment.
2. Clone the `azure-utpm-c` GitHub repository using the following command:

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

1. Open a Git CMD or Git Bash command-line environment.
2. Clone the [Java](#) GitHub repository using the following command:

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

Build and run the TPM device simulator

In this section, you'll build and run the TPM simulator. This simulator listens over a socket on ports 2321 and 2322. Do not close the command window. You'll need to keep this simulator running until the end of this quickstart.

1. Run the following command to build Azure IoT C SDK that includes the TPM device simulator sample code. A Visual Studio solution for the simulated device is generated in the `cmake` directory. This sample provides a TPM [attestation mechanism](#) via Shared Access Signature (SAS) Token authentication.

```
cmake -Duse_prov_client:BOOL=ON -Duse_tpm_simulator:BOOL=ON ..
```

TIP

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

2. When the build succeeds, the last few output lines look similar to the following output:

```
$ cmake -Duse_prov_client:BOOL=ON ..  
-- Building for: Visual Studio 16 2019  
-- The C compiler identification is MSVC 19.23.28107.0  
-- The CXX compiler identification is MSVC 19.23.28107.0  
  
...  
  
-- Configuring done  
-- Generating done  
-- Build files have been written to: C:/code/azure-iot-sdk-c/cmake
```

3. Go to the root folder of the git repository you cloned.

4. Run the [TPM](#) simulator using the path shown below.

```
cd ..  
.\\provisioning_client\\deps\\utpm\\tools\\tpm_simulator\\Simulator.exe
```

The simulator doesn't display any output. Let it continue to run as it simulates a TPM device.

1. Go to the GitHub root folder.
2. Run the [TPM](#) simulator to be the [HSM](#) for the simulated device.

```
.\\azure-utpm-c\\tools\\tpm_simulator\\Simulator.exe
```

3. Create a new empty folder called **registerdevice**. In the **registerdevice** folder, create a `package.json` file using the following command at your command prompt(make sure to answer all questions asked by `npm` or accept the defaults if they suit you):

```
npm init
```

4. Install the following precursor packages:

```
npm install node-gyp -g  
npm install ffi -g
```

NOTE

There are some known issues to installing the above packages. To resolve these issues, run

```
npm install --global --production windows-build-tools
```

 using a command prompt in Run as administrator mode, run

```
SET VCTargetsPath=C:\Program Files (x86)\MSBuild\Microsoft.Cpp\v4.0\V140
```

 after replacing the path with your installed version, and then rerun the above installation commands.

5. Install all required packages running the following command at your command prompt in the **registerdevice** folder:

```
npm install --save azure-iot-device azure-iot-device-mqtt azure-iot-security-tpm azure-iot-provisioning-device-http azure-iot-provisioning-device
```

The command installs the following packages:

- A security client that works with TPM: `azure-iot-security-tpm`
- A transport for the device to connect to the Device Provisioning Service: either `azure-iot-provisioning-device-http` or `azure-iot-provisioning-device-amqp`
- A client to use the transport and security client: `azure-iot-provisioning-device`
- The device client: `azure-iot-device`
- A transport: any of `azure-iot-device-amqp`, `azure-iot-device-mqtt`, Or `azure-iot-device-http`
- The security client that you already installed: `azure-iot-security-tpm`

NOTE

The samples in this quickstart use the `azure-iot-provisioning-device-http` and `azure-iot-device-mqtt` transports.

6. Open a text editor of your choices.
7. In the **registerdevice** folder, create a new *ExtractDevice.js* file.
8. Add the following `require` statements at the start of the *ExtractDevice.js* file:

```
'use strict';

var tpmSecurity = require('azure-iot-security-tpm');
var tssJs = require("tss.js");

var myTpm = new tpmSecurity.TpmSecurityClient(undefined, new tssJs.Tpm(true));
```

9. Add the following function to implement the method:

```
myTpm.getEndorsementKey(function(err, endorsementKey) {
  if (err) {
    console.log('The error returned from get key is: ' + err);
  } else {
    console.log('the endorsement key is: ' + endorsementKey.toString('base64'));
    myTpm.getRegistrationId((getRegistrationIdError, registrationId) => {
      if (getRegistrationIdError) {
        console.log('The error returned from get registration id is: ' + getRegistrationIdError);
      } else {
        console.log('The Registration Id is: ' + registrationId);
        process.exit();
      }
    });
  }
});
```

10. Save and close the *ExtractDevice.js* file.

```
node ExtractDevice.js
```

11. Run the sample.

12. The output window displays the *Endorsement key* and the *Registration ID* needed for device enrollment. Copy these values.

1. Run the [TPM](#) simulator to be the [HSM](#) for the simulated device.
2. Select **Allow Access**. The simulator listens over a socket on ports 2321 and 2322. Do not close this command window; you will need to keep this simulator running until the end of this quickstart guide.

```
.\azure-iot-sdk-java\provisioning\provisioning-tools\tpm-simulator\Simulator.exe
```



3. Open a second command prompt.
4. In the second command prompt, navigate to the root folder and build the sample dependencies.

```
cd azure-iot-sdk-java
mvn install -DskipTests=true
```

5. Navigate to the sample folder.

```
cd provisioning/provisioning-samples/provisioning-tpm-sample
```

Read cryptographic keys from the TPM device

In this section, you'll build and execute a sample that reads the endorsement key and registration ID from the TPM simulator you left running, and is still listening over ports 2321 and 2322. These values will be used for device enrollment with your Device Provisioning Service instance.

1. Launch Visual Studio.
2. Open the solution generated in the `cmake` folder named `azure_iot_sdks.sln`.
3. On the Visual Studio menu, select **Build > Build Solution** to build all projects in the solution.
4. In Visual Studio's *Solution Explorer* window, navigate to the **Provision_Tools** folder. Right-click the **tpm_device_provision** project and select **Set as Startup Project**.
5. On the Visual Studio menu, select **Debug > Start without debugging** to run the solution. The app reads and displays a **Registration ID** and an **Endorsement key**. Note or copy these values. These will be used in the next section for device enrollment.

1. Sign in to the Azure portal, select the **All resources** button on the left-hand menu and open your Device Provisioning Service. Note your **ID Scope** and **Provisioning Service Global Endpoint**.

The screenshot shows the Azure portal interface for a Device Provisioning Service named 'test-docs-dps'. The left sidebar has a 'Properties' section selected. The main content area shows the following details:

Resource group (change) test-rg-dps	Service endpoint test-docs-dps.azure-devices-provisioning.net
Status Active	Global device endpoint global.azure-devices-provisioning.net
Location East US	ID Scope One00000A0A
Subscription (change) Microsoft Azure Internal Consumption	Pricing and scale tier S1
Subscription ID ****	

Below this, there is a 'Quick Links' section with a link to 'Azure IoT Hub Device Provisioning Service Documentation'.

2. Edit `src/main/java/samples/com/microsoft/azure/sdk/iot/ProvisioningTpmSample.java` to include your **ID Scope** and **Provisioning Service Global Endpoint** as noted before.

```
private static final String idScope = "[Your ID scope here]";  
private static final String globalEndpoint = "[Your Provisioning Service Global Endpoint here]";  
private static final ProvisioningDeviceClientTransportProtocol  
PROVISIONING_DEVICE_CLIENT_TRANSPORT_PROTOCOL = ProvisioningDeviceClientTransportProtocol.HTTPS;
```

3. Save the file.
4. Use the following commands to build the project, navigate to the target folder, and execute the created `.jar` file (replace `{version}` with your version of Java):

```
mvn clean install  
cd target  
java -jar ./provisioning-tpm-sample-{version}-with-deps.jar
```

5. When the program begins running, it will display the **Endorsement key** and **Registration ID**. Copy these values for the next section. Make sure to leave the program running.

In this section, you'll build and execute a sample that reads the endorsement key from your TPM 2.0 hardware security module. This value will be used for device enrollment with your Device Provisioning Service instance.

1. In a command prompt, change directories to the project directory for the TPM device provisioning sample.

```
cd .\azure-iot-samples-csharp\provisioning\Samples\device\TpmSample
```

2. Type the following command to build and run the TPM device provisioning sample. Copy the endorsement key returned from your TPM 2.0 hardware security module to use later when enrolling your device.

```
dotnet run -- -e
```

Create a device enrollment entry

1. Sign in to the [Azure portal](#).
2. On the left-hand menu or on the portal page, select **All resources**.
3. Select your Device Provisioning Service.
4. In the **Settings** menu, select **Manage enrollments**.
5. At the top of the page, select **+ Add individual enrollment**.
6. In the **Add Enrollment** panel, enter the following information:
 - Select **TPM** as the identity attestation *Mechanism*.
 - Enter the *Registration ID* and *Endorsement key* for your TPM device from the values you noted previously.
 - Select an IoT hub linked with your provisioning service.
 - Optionally, you may provide the following information:
 - Enter a unique *Device ID* (you can use the suggested **test-docs-device** or provide your own). Make sure to avoid sensitive data while naming your device. If you choose not to provide one, the registration ID will be used to identify the device instead.
 - Update the **Initial device twin state** with the desired initial configuration for the device.
 - Once complete, press the **Save** button.

The screenshot shows two windows side-by-side. On the left is the 'test-docs-dps - Manage enrollments' page, which has a sidebar with various service management options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings, Quick Start, Shared access policies, Linked IoT hubs, Certificates, and Manage enrollments (which is highlighted). The main area shows tabs for 'Enrollment Groups' and 'Individual Enrollments', with a message indicating you can add or remove individual device enrollments and/or enrollment groups. On the right is the 'Add Enrollment' dialog, which has a 'Save' button at the top. It contains fields for 'Mechanism' (set to TPM), 'Endorsement key' (a dropdown menu), 'Registration ID' (a dropdown menu), 'IoT Hub Device ID' (a dropdown menu), and 'Device ID' (a dropdown menu). Below these are sections for assigning devices to hubs ('Select how you want to assign devices to hubs' set to 'Evenly weighted distribution') and handling re-provisioning ('Select how you want device data to be handled on re-provisioning' set to 'Re-provision and migrate data'). At the bottom is a 'Device Twin is only supported for standard tier IoT hubs. Learn more about standard tier.' note and a 'Initial Device Twin State' text input field containing a JSON object with a 'tags' key.

7. Select Save.

1. Sign in to the [Azure portal](#).
2. On the left-hand menu or on the portal page, select All resources.
3. Select your Device Provisioning Service.
4. In the Settings menu, select **Manage enrollments**.
5. At the top of the page, select + **Add individual enrollment**.
6. In the **Add Enrollment** panel, enter the following information:
 - Select **TPM** as the identity attestation *Mechanism*.
 - Enter the *Endorsement key* you retrieved earlier from your HSM.
 - Enter a unique *Registration ID* for your device. You will also use this registration ID when registering your device, so make a note of it for later.
 - Select an IoT hub linked with your provisioning service.
 - Optionally, you may provide the following information:
 - Enter a unique *Device ID* (you can use the suggested **test-docs-device** or provide your own). Make sure to avoid sensitive data while naming your device. If you choose not to provide one, the registration ID will be used to identify the device instead.
 - Update the **Initial device twin state** with the desired initial configuration for the device.
 - Once complete, press the **Save** button.

The screenshot shows two windows side-by-side. On the left is the 'Device Provisioning Service' blade under 'All resources'. It has a sidebar with 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', 'Settings' (selected), 'Quick Start', 'Shared access policies', 'Linked IoT hubs', 'Certificates', 'Manage enrollments' (selected), 'Manage allocation policy', 'Properties', 'Locks', 'Export template', 'Monitoring' (Alerts, Metrics, Diagnostic settings, Logs), 'Support + troubleshooting' (New support request), and 'Help & feedback'. The main area shows 'Enrollment Groups' and 'Individual Enrollments'. A red box highlights the 'Individual Enrollments' tab. On the right is the 'Add Enrollment' dialog. It has fields for 'Mechanism' (set to 'TPM'), 'Endorsement key', 'Registration ID' (set to 'Individual enrollment registration id'), 'IoT Hub Device ID' (set to 'Device ID'), 'IoT Edge device' (set to 'False'), 'Select how you want to assign devices to hubs' (set to 'Evenly weighted distribution'), 'Select the IoT hubs this device can be assigned to' (set to 'test-docs-dps.azure-devices.net'), and 'Link a new IoT hub'. Below these are sections for 'Select how you want device data to be handled on re-provisioning' (set to 'Re-provision and migrate data') and 'Initial Device Twin State' (with a JSON editor showing an empty object). A red box highlights the 'Save' button at the top.

7. Select **Save**.

Register the device

In this section, you'll configure sample code to use the [Advanced Message Queuing Protocol \(AMQP\)](#) to send the device's boot sequence to your Device Provisioning Service instance. This boot sequence causes the device to be registered to an IoT hub linked to the Device Provisioning Service instance.

1. In the Azure portal, select the **Overview** tab for your Device Provisioning Service.
2. Copy the **ID Scope** value.

The screenshot shows the 'test-docs-dps' Device Provisioning Service blade in the Azure portal. The left sidebar includes 'Overview' (selected), 'Activity log', 'Access control (IAM)', 'Properties', 'Locks', 'Automation script', 'Quick Start', 'Shared access policies', 'Linked IoT hubs', 'Certificates', and 'Manage enrollments'. The main area displays service details: Resource group ('test-rg-dps'), Status ('Active'), Location ('East US'), Subscription ('Microsoft Azure Internal Consumption'), and Subscription ID ('****'). It also shows the 'Service endpoint' ('test-docs-dps.azure-devices-provisioning.net'), 'Global device endpoint' ('global.azure-devices-provisioning.net'), and 'ID Scope' ('One00000A0A'). Below this is a 'Quick Links' section with links to 'Azure IoT Hub Device Provisioning Service Documentation', 'Learn more about IoT Hub Device Provisioning Service', and 'Device Provisioning concepts'.

3. In Visual Studio's *Solution Explorer* window, navigate to the **Provision_Samples** folder. Expand the sample project named **prov_dev_client_sample**. Expand **Source Files**, and open **prov_dev_client_sample.c**.
4. Near the top of the file, find the `#define` statements for each device protocol as shown below. Make sure only `SAMPLE_AMQP` is uncommented.

Currently, the [MQTT](#) protocol is not supported for TPM Individual Enrollment.

```
//  
// 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
```

5. Find the `id_scope` constant, and replace the value with your **ID Scope** value that you copied earlier.

```
static const char* id_scope = "0ne00002193";
```

6. Find the definition for the `main()` function in the same file. Make sure the `hsm_type` variable is set to `SECURE_DEVICE_TYPE_TPM` as shown below.

```
SECURE_DEVICE_TYPE hsm_type;  
hsm_type = SECURE_DEVICE_TYPE_TPM;  
//hsm_type = SECURE_DEVICE_TYPE_X509;  
//hsm_type = SECURE_DEVICE_TYPE_SYMMETRIC_KEY;
```

7. Right-click the `prov_dev_client_sample` project and select **Set as Startup Project**.

8. On the Visual Studio menu, select **Debug > Start without debugging** to run the solution. In the prompt to rebuild the project, select **Yes**, to rebuild the project before running.

The following output is an example of the provisioning device client sample successfully booting up, and connecting to a Device Provisioning Service instance to get IoT hub information and registering:

```
Provisioning API Version: 1.2.7  
  
Registering... Press enter key to interrupt.  
  
Provisioning Status: PROV_DEVICE_REG_STATUS_CONNECTED  
Provisioning Status: PROV_DEVICE_REG_STATUS_ASSIGNING  
Provisioning Status: PROV_DEVICE_REG_STATUS_ASSIGNING  
  
Registration Information received from service:  
test-docs-hub.azure-devices.net, deviceId: test-docs-cert-device
```

1. In the Azure portal, select the **Overview** tab for your Device Provisioning Service.
2. Copy the **ID Scope** value.

The screenshot shows the Azure portal interface for a Device Provisioning Service named "test-docs-dps". The "Overview" tab is selected. Key details shown include:

- Resource group: test-rg-dps
- Status: Active
- Location: East US
- Subscription: Microsoft Azure Internal Consumption
- Pricing and scale tier: S1
- ID Scope: One00000A0A

On the right, there's a "Quick Links" section with three items:

- Azure IoT Hub Device Provisioning Service Documentation
- Learn more about IoT Hub Device Provisioning Service
- Device Provisioning concepts

3. In a command prompt, change directories to the project directory for the TPM device provisioning sample.

```
cd .\azure-iot-samples-csharp\provisioning\Samples\device\TpmSample
```

4. Run the following command to register your device. Replace <IdScope> with the value for the DPS you just copied and <RegistrationId> with the value you used when creating the device enrollment.

```
dotnet run -- -s <IdScope> -r <RegistrationId>
```

If the device registration was successful, you'll see the following messages:

```
Initializing security using the local TPM...
Initializing the device provisioning client...
Initialized for registration Id <RegistrationId>.
Registering with the device provisioning service...
Registration status: Assigned.
Device <RegistrationId> registered to <HubName>.azure-devices.net.
Creating TPM authentication for IoT Hub...
Testing the provisioned device with IoT Hub...
Sending a telemetry message...
Finished.
```

1. In the Azure portal, select the **Overview** tab for your Device Provisioning Service.
2. Copy the **ID Scope** value.

3. Open a text editor of your choice.
4. In the `registerdevice` folder, create a new `RegisterDevice.js` file.
5. Add the following `require` statements at the start of the `RegisterDevice.js` file:

```
'use strict';

var ProvisioningTransport = require('azure-iot-provisioning-device-http').Http;
var iotHubTransport = require('azure-iot-device-mqtt').Mqtt;
var Client = require('azure-iot-device').Client;
var Message = require('azure-iot-device').Message;
var tpmSecurity = require('azure-iot-security-tpm');
var ProvisioningDeviceClient = require('azure-iot-provisioning-device').ProvisioningDeviceClient;
```

NOTE

The Azure IoT SDK for Node.js supports additional protocols like *AMQP*, *AMQP WS*, and *MQTT WS*. For more examples, see [Device Provisioning Service SDK for Nodejs samples](#).

6. Add `globalDeviceEndpoint` and `idScope` variables and use them to create a `ProvisioningDeviceClient` instance. Replace `{globalDeviceEndpoint}` and `{idScope}` with the *Global Device Endpoint* and *ID Scope* values from Step 1:

```
var provisioningHost = '{globalDeviceEndpoint}';
var idScope = '{idScope}';

var tssJs = require("tss.js");
var securityClient = new tpmSecurity.TpmSecurityClient('', new tssJs.Tpm(true));
// if using non-simulated device, replace the above line with following:
//var securityClient = new tpmSecurity.TpmSecurityClient();

var provisioningClient = ProvisioningDeviceClient.create(provisioningHost, idScope, new
ProvisioningTransport(), securityClient);
```

7. Add the following function to implement the method on the device:

```

provisioningClient.register(function(err, result) {
  if (err) {
    console.log("error registering device: " + err);
  } else {
    console.log('registration succeeded');
    console.log('assigned hub=' + result.registrationState.assignedHub);
    console.log('deviceId=' + result.registrationState.deviceId);
    var tpmAuthenticationProvider =
      tpmSecurity.TpmAuthenticationProvider.fromTpmSecurityClient(result.registrationState.deviceId,
      result.registrationState.assignedHub, securityClient);
    var hubClient = Client.fromAuthenticationProvider(tpmAuthenticationProvider, iotHubTransport);

    var connectCallback = function (err) {
      if (err) {
        console.error('Could not connect: ' + err.message);
      } else {
        console.log('Client connected');
        var message = new Message('Hello world');
        hubClient.sendEvent(message, printResultFor('send'));
      }
    };
    hubClient.open(connectCallback);

    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);
        process.exit(1);
      };
    }
  });
});

```

8. Save and close the *RegisterDevice.js* file.

9. Run the following command:

```
node RegisterDevice.js
```

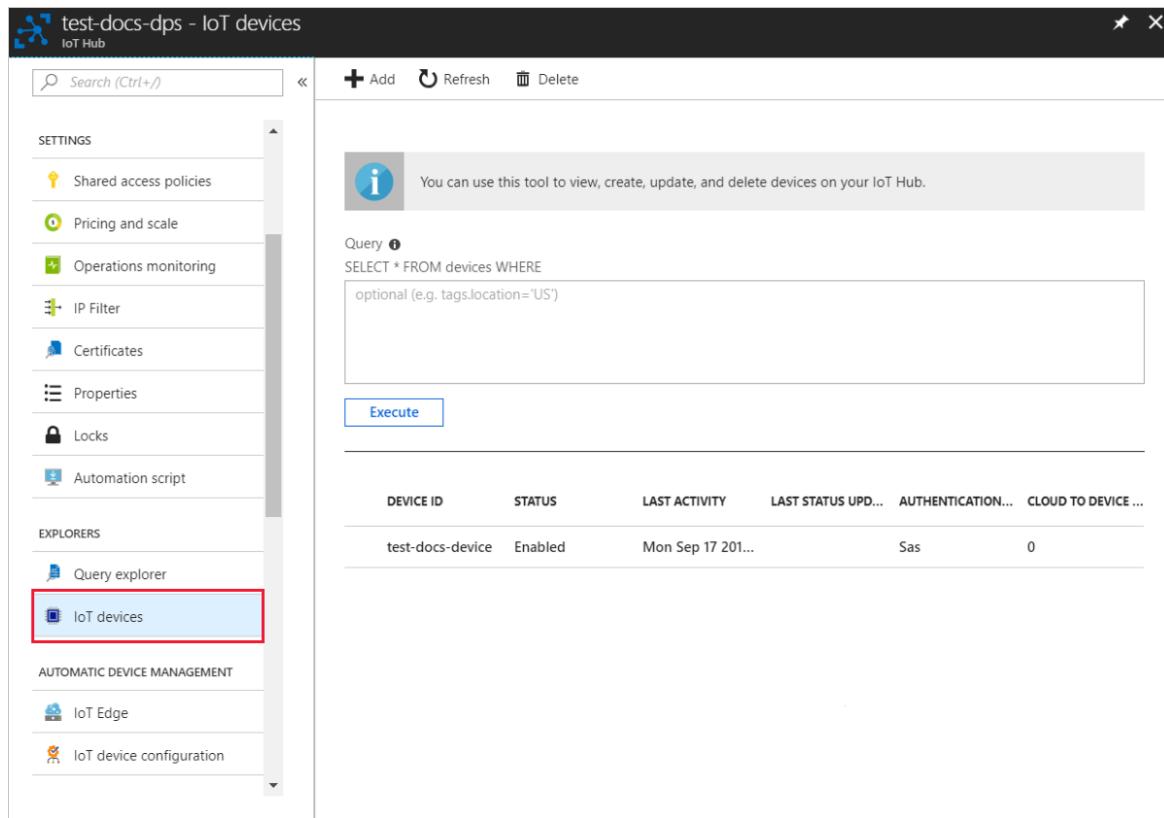
10. Notice the messages that simulate the device booting and connecting to the Device Provisioning Service to get your IoT hub information.

1. In the command window running the Java sample code on your machine, press *Enter* to continue running the application. Notice the messages that simulate the device booting and connecting to the Device Provisioning Service to get your IoT hub information.

Confirm your device provisioning registration

1. Go to the [Azure portal](#).

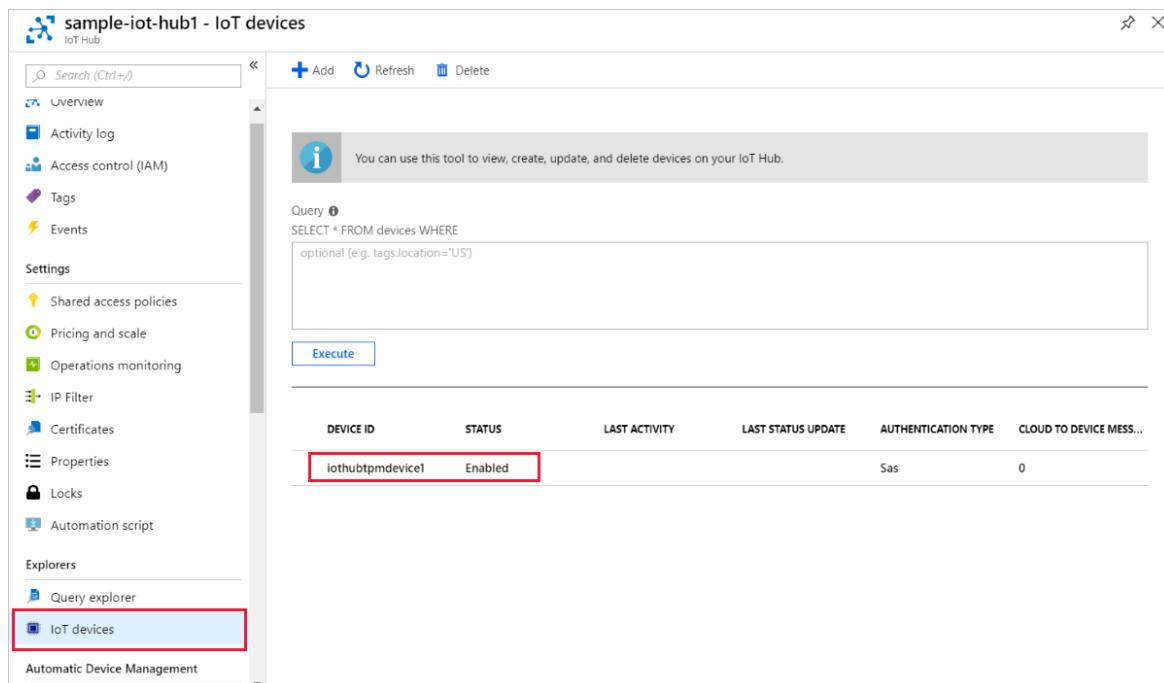
2. On the left-hand menu or on the portal page, select **All resources**.
3. Select the IoT hub to which your device was assigned.
4. In the **Explorers** menu, select **IoT Devices**.
5. If your device was provisioned successfully, the device ID should appear in the list, with **Status** set as *enabled*. If you don't see your device, select **Refresh** at the top of the page.



The screenshot shows the 'IoT devices' blade in the Azure IoT Hub. The left sidebar includes sections for SETTINGS (Shared access policies, Pricing and scale, Operations monitoring, IP Filter, Certificates, Properties, Locks, Automation script), EXPLORERS (Query explorer, IoT devices, highlighted with a red box), and AUTOMATIC DEVICE MANAGEMENT (IoT Edge, IoT device configuration). The main area has a search bar, buttons for Add, Refresh, and Delete, and a query editor with a placeholder 'optional (e.g. tags.location='US')'. Below is a table with columns: DEVICE ID, STATUS, LAST ACTIVITY, LAST STATUS UPD..., AUTHENTICATION..., CLOUD TO DEVICE A single row is shown: test-docs-device, Enabled, Mon Sep 17 201..., Sas, 0.

DEVICE ID	STATUS	LAST ACTIVITY	LAST STATUS UPD...	AUTHENTICATION...	CLOUD TO DEVICE ...
test-docs-device	Enabled	Mon Sep 17 201...	Sas	0	

5. If your device was provisioned successfully, the device ID should appear in the list, with **Status** set as *enabled*. If you don't see your device, select **Refresh** at the top of the page.



The screenshot shows the 'IoT devices' blade in the Azure IoT Hub. The left sidebar includes sections for overview, Activity log, Access control (IAM), Tags, Events, Settings (Shared access policies, Pricing and scale, Operations monitoring, IP Filter, Certificates, Properties, Locks, Automation script), Explorers (Query explorer, IoT devices, highlighted with a red box), and Automatic Device Management. The main area has a search bar, buttons for Add, Refresh, and Delete, and a query editor with a placeholder 'optional (e.g. tags.location='US')'. Below is a table with columns: DEVICE ID, STATUS, LAST ACTIVITY, LAST STATUS UPDATE, AUTHENTICATION TYPE, CLOUD TO DEVICE MESS... . A single row is shown: iothubtpmdevice1, Enabled, highlighted with a red box.

DEVICE ID	STATUS	LAST ACTIVITY	LAST STATUS UPDATE	AUTHENTICATION TYPE	CLOUD TO DEVICE MESS...
iothubtpmdevice1	Enabled		Sas	0	

5. If your device was provisioned successfully, the device ID should appear in the list, with **Status** set as *enabled*. If you don't see your device, select **Refresh** at the top of the page.

The screenshot shows the Azure IoT Hub Device Management interface. On the left, a sidebar lists various management options: SETTINGS (Shared access policies, Pricing and scale, Operations monitoring, IP Filter, Certificates, Properties, Locks, Automation script), EXPLORERS (Query explorer, IoT devices), and AUTOMATIC DEVICE MANAGEMENT (IoT Edge, IoT device configuration). The 'IoT devices' item is highlighted with a red box. The main pane displays a query editor with the following content:

```
Query ⓘ
SELECT * FROM devices WHERE
optional (e.g. tags.location='US')
```

Below the query editor is a table showing device details:

DEVICE ID	STATUS	LAST ACTIVITY	LAST STATUS UPD...	AUTHENTICATION...	CLOUD TO DEVICE ...
test-docs-device	Enabled	Mon Sep 17 201...	Sas	0	

5. If your device was provisioned successfully, the device ID should appear in the list, with **Status** set as *enabled*. If you don't see your device, select **Refresh** at the top of the page.

The screenshot shows the same Azure IoT Hub Device Management interface as the previous one. The 'IoT devices' item in the sidebar is highlighted with a red box. The main pane displays a query editor and a table showing device details. In the table, the 'Java-device' row is highlighted with a red box.

DEVICE ID	STATUS	LAST ACTIVITY	LAST STATUS UPD...	AUTHENTICATION...	CLOUD TO DEVICE ...
Java-device	Enabled		Sas	0	

NOTE

If you changed the *initial device twin state* from the default value in the enrollment entry for your device, it can pull the desired twin state from the hub and act accordingly. For more information, see [Understand and use device twins in IoT Hub](#).

Clean up resources

If you plan to continue working on and exploring the device client sample, don't clean up the resources created in this quickstart. If you don't plan to continue, use the following steps to delete all resources created by this quickstart.

Delete your device enrollment

1. Close the device client sample output window on your machine.
2. From the left-hand menu in the Azure portal, select **All resources**.
3. Select your Device Provisioning Service.
4. In the **Settings** menu, select **Manage enrollments**.
5. Select the **Individual Enrollments** tab.
6. Select the check box next to the *REGISTRATION ID* of the device you enrolled in this quickstart.
7. At the top of the page, select **Delete**.

Delete your device registration from IoT Hub

1. From the left-hand menu in the Azure portal, select **All resources**.
2. Select your IoT hub.
3. In the **Explorers** menu, select **IoT devices**.
4. Select the check box next to the *DEVICE ID* of the device you registered in this quickstart.
5. At the top of the page, select **Delete**.

Next steps

In this quickstart, you've created a TPM simulated device on your machine and provisioned it to your IoT hub using the IoT Hub Device Provisioning Service. To learn how to enroll your TPM device programmatically, continue to the quickstart for programmatic enrollment of a TPM device.

[Create an individual enrollment for a TPM device using the DPS service SDK](#)

[Create an individual enrollment for a TPM device using the DPS service SDK](#)

[Create an individual enrollment for a TPM device using the DPS service SDK](#)

Tutorial: Provision devices using symmetric key enrollment groups

8/22/2022 • 9 minutes to read • [Edit Online](#)

This tutorial shows how to securely provision multiple simulated symmetric key devices to a single IoT Hub using an enrollment group.

Some devices may not have a certificate, TPM, or any other security feature that can be used to securely identify the device. The Device Provisioning Service includes [symmetric key attestation](#). Symmetric key attestation can be used to identify a device based off unique information like the MAC address or a serial number.

If you can easily install a [hardware security module \(HSM\)](#) and a certificate, then that may be a better approach for identifying and provisioning your devices. Using an HSM will allow you to bypass updating the code deployed to all your devices, and you would not have a secret key embedded in your device images. This tutorial assumes that neither an HSM or a certificate is a viable option. However, it is assumed that you do have some method of updating device code to use the Device Provisioning Service to provision these devices.

This tutorial also assumes that the device update takes place in a secure environment to prevent unauthorized access to the master group key or the derived device key.

This tutorial is oriented toward a Windows-based workstation. However, you can perform the procedures on Linux. For a Linux example, see [How to provision for multitenancy](#).

NOTE

The sample used in this tutorial is written in C. There is also a [C# device provisioning symmetric key sample](#) available. To use this sample, download or clone the [azure-iot-samples-csharp](#) repository and follow the in-line instructions in the sample code. You can follow the instructions in this tutorial to create a symmetric key enrollment group using the portal and to find the ID Scope and enrollment group primary and secondary keys needed to run the sample. You can also create individual enrollments using the sample.

Prerequisites

- Completion of the [Set up IoT Hub Device Provisioning Service with the Azure portal](#) quickstart.

The following prerequisites are for a Windows development environment. For Linux or macOS, see the appropriate section in [Prepare your development environment](#) in the SDK documentation.

- [Visual Studio 2019](#) with the '[Desktop development with C++](#)' workload enabled. Visual Studio 2015 and Visual Studio 2017 are also supported.
- Latest version of [Git](#) installed.

Overview

A unique registration ID will be defined for each device based on information that identifies that device. For example, the MAC address or a serial number.

An enrollment group that uses [symmetric key attestation](#) will be created with the Device Provisioning Service. The enrollment group will include a group master key. That master key will be used to hash each unique registration ID to produce a unique device key for each device. The device will use that derived device key with

its unique registration ID to attest with the Device Provisioning Service and be assigned to an IoT hub.

The device code demonstrated in this tutorial will follow the same pattern as the [Quickstart: Provision a simulated symmetric key device](#). The code will simulate a device using a sample from the [Azure IoT C SDK](#). The simulated device will attest with an enrollment group instead of an individual enrollment as demonstrated in the quickstart.

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

Prepare an Azure IoT C SDK development environment

In this section, you will prepare a development environment used to build the [Azure IoT C SDK](#).

The SDK includes the sample code for the simulated device. This simulated device will attempt provisioning during the device's boot sequence.

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, which builds a version of the SDK specific to your development client platform. A Visual Studio solution for the simulated device will be generated in the `cmake` directory.

```
cmake -Dhsm_type_symm_key:BOOL=ON -Duse_prov_client:BOOL=ON ..
```

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

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

```
$ cmake -Dhsm_type_symm_key:BOOL=ON -Duse_prov_client:BOOL=ON ...
-- 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 a symmetric key enrollment group

1. Sign in to the [Azure portal](#), and open your Device Provisioning Service instance.
2. Select the **Manage enrollments** tab, and then click the **Add enrollment group** button at the top of the page.
3. On **Add Enrollment Group**, enter the following information, and click the **Save** button.
 - **Group name:** Enter `mylegacydevices`. The enrollment group name is a case-insensitive string (up to 128 characters long) of alphanumeric characters plus the special characters: `'-'`, `'.'`, `'_'`, `:`. The last character must be alphanumeric or dash (`'-'`).
 - **Attestation Type:** Select **Symmetric Key**.
 - **Auto Generate Keys:** Check this box.
 - **Select how you want to assign devices to hubs:** Select **Static configuration** so you can assign to a specific hub.
 - **Select the IoT hubs this group can be assigned to:** Select one of your hubs.

Add Enrollment Group

Save

Group name
mylegacydevices

Attestation Type (preview)
 Certificate Symmetric Key (preview)

Auto-generate keys (preview)

Primary Key (preview)
Enter your primary key here

Secondary Key (preview)
Enter your secondary key here

Device Twin is only supported for standard tier IoT hubs. Learn more about standard tier.

Select how you want to assign devices to hubs (preview for enrollment)

Lowest latency
 Evenly weighted distribution
 Static configuration (via enrollment list only)

* Select the IoT hubs this group can be assigned to: (preview)
YourHub.azure-devices.net

Link a new IoT hub

- Once you saved your enrollment, the **Primary Key** and **Secondary Key** will be generated and added to the enrollment entry. Your symmetric key enrollment group appears as **mylegacydevices** under the *Group Name* column in the *Enrollment Groups* tab.

Open the enrollment and copy the value of your generated **Primary Key**. This key is your master group key.

Choose a unique registration ID for the device

A unique registration ID must be defined to identify each device. You can use the MAC address, serial number, or any unique information from the device.

In this example, we use a combination of a MAC address and serial number forming the following string for a registration ID.

```
sn-007-888-abc-mac-a1-b2-c3-d4-e5-f6
```

Create unique registration IDs for each device. The registration ID is a case-insensitive string (up to 128 characters long) of alphanumeric characters plus the special characters: `'-'`, `'.'`, `'_'`, `':'`. The last character must be alphanumeric or dash (`'-'`).

Derive a device key

To generate device keys, use the enrollment group master key to compute an **HMAC-SHA256** of the registration ID for each device. The result is then converted into Base64 format for each device.

WARNING

Your device code for each device should only include the corresponding derived device key for that device. Do not include your group master key in your device code. A compromised master key has the potential to compromise the security of all devices being authenticated with it.

- [Azure CLI](#)
- [Windows](#)
- [Linux](#)

The IoT extension for the Azure CLI provides the `compute-device-key` command for generating derived device keys. This command can be used from a Windows-based or Linux systems, in PowerShell or a Bash shell.

Replace the value of `--key` argument with the **Primary Key** from your enrollment group.

Replace the value of `--registration-id` argument with your registration ID.

```
az iot dps compute-device-key --key  
8isrFI1sGslvvFSSFRiMfCNzv21fjbE/+ah/lSh3lF8e2YG1Te7w1KpZhJFFXJrqYKi9yegxkqIChbqOS9Egw== --registration-id  
sn-007-888-abc-mac-a1-b2-c3-d4-e5-f6
```

Example result:

```
"Jsm0lyGpjAVYVP2g3FnmmnG9dI/9qU24wNoykUmermc="
```

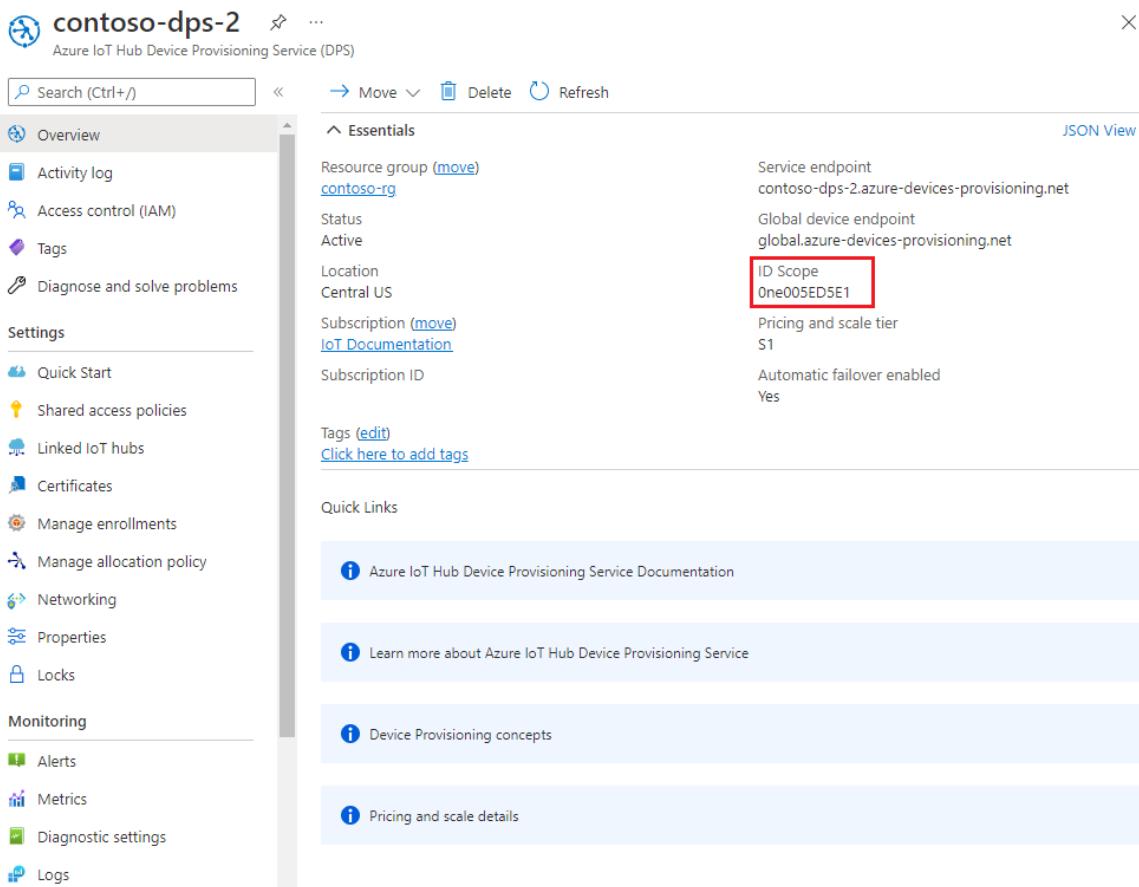
Each device uses its derived device key and unique registration ID to perform symmetric key attestation with the enrollment group during provisioning.

Create a device image to provision

In this section, you will update a provisioning sample named `prov_dev_client_sample` located in the Azure IoT C SDK you set up earlier.

This sample code simulates a device boot sequence that sends the provisioning request to your Device Provisioning Service instance. The boot sequence will cause the device to be recognized and assigned to the IoT hub you configured on the enrollment group. This would be completed for each device that would be provisioned using the enrollment group.

1. In the Azure portal, select the **Overview** tab for your Device Provisioning Service and note down the **ID Scope** value.



The screenshot shows the Azure IoT Hub Device Provisioning Service (DPS) Overview page for a resource named "contoso-dps-2". The left sidebar contains navigation links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings (Quick Start, Shared access policies, Linked IoT hubs, Certificates, Manage enrollments, Manage allocation policy, Networking, Properties, Locks), Monitoring (Alerts, Metrics, Diagnostic settings, Logs), and Quick Links. The main content area displays the following details:

Setting	Value
Resource group (move)	contoso-rg
Status	Active
Location	Central US
Subscription (move)	IoT Documentation
Subscription ID	
Tags (edit)	Click here to add tags
ID Scope	One005ED5E1
Pricing and scale tier	S1
Automatic failover enabled	Yes

2. In Visual Studio, open the `azure_iot_sdks.sln` solution file that was generated by running CMake earlier. The solution file should be in the following location:

```
\azure-iot-sdk-c\cmake\azure_iot_sdks.sln
```

3. In Visual Studio's *Solution Explorer* window, navigate to the **Provision_Samples** folder. Expand the sample project named `prov_dev_client_sample`. Expand **Source Files**, and open `prov_dev_client_sample.c`.
4. Find the `id_scope` constant, and replace the value with your **ID Scope** value that you copied earlier.

```
static const char* id_scope = "One00002193";
```

5. Find the definition for the `main()` function in the same file. Make sure the `hsm_type` variable is set to `SECURE_DEVICE_TYPE_SYMMETRIC_KEY` as shown below:

```
SECURE_DEVICE_TYPE hsm_type;
//hsm_type = SECURE_DEVICE_TYPE_TPM;
//hsm_type = SECURE_DEVICE_TYPE_X509;
hsm_type = SECURE_DEVICE_TYPE_SYMMETRIC_KEY;
```

6. Find the call to `prov_dev_set_symmetric_key_info()` in `prov_dev_client_sample.c` which is commented out.

```
// Set the symmetric key if using they auth type
//prov_dev_set_symmetric_key_info("<symm_registration_id>", "<symmetric_Key>");
```

Uncomment the function call, and replace the placeholder values (including the angle brackets) with the unique registration ID for your device and the derived device key you generated.

```
// Set the symmetric key if using they auth type
prov_dev_set_symmetric_key_info("sn-007-888-abc-mac-a1-b2-c3-d4-e5-f6",
"Jsm0lyGpjVVVP2g3FnmnmG9dI/9qU24wNoykUmcmc=");
```

Save the file.

7. Right-click the `prov_dev_client_sample` project and select **Set as Startup Project**.
8. On the Visual Studio menu, select **Debug > Start without debugging** to run the solution. In the prompt to rebuild the project, click **Yes**, to rebuild the project before running.

The following output is an example of the simulated device successfully booting up, and connecting to the provisioning Service instance to be assigned to an IoT hub:

```
Provisioning API Version: 1.2.8

Registering Device

Provisioning Status: PROV_DEVICE_REG_STATUS_CONNECTED
Provisioning Status: PROV_DEVICE_REG_STATUS_ASSIGNING
Provisioning Status: PROV_DEVICE_REG_STATUS_ASSIGNING

Registration Information received from service:
test-docs-hub.azure-devices.net, deviceId: sn-007-888-abc-mac-a1-b2-c3-d4-e5-f6

Press enter key to exit:
```

9. In the portal, navigate to the IoT hub your simulated device was assigned to and click the **IoT Devices** tab. On successful provisioning of the simulated to the hub, its device ID appears on the **IoT Devices** blade, with **STATUS** as **enabled**. You might need to click the **Refresh** button at the top.

You can use this tool to view, create, update, and delete devices on your IoT Hub.

Query ?

SELECT * FROM devices WHERE
optional (e.g. tags.location='US')

Execute

DEVICE ID	STATUS	AUTHENTICATION TYPE
sn-007-888-abc-mac-a1-b2-c3-d4-e5-f6	Enabled		Sas

Security concerns

Be aware that this leaves the derived device key included as part of the image for each device, which is not a recommended security best practice. This is one reason why security and ease-of-use are often tradeoffs. You must fully review the security of your devices based on your own requirements.

Next steps

- To learn more about Reprovisioning, see

[IoT Hub Device reprovisioning concepts](#)

[Quickstart: Provision a simulated symmetric key device](#)

- To learn more about Deprovisioning, see

[How to deprovision devices that were previously auto-provisioned](#)

Tutorial: Use custom allocation policies with Device Provisioning Service (DPS)

8/22/2022 • 17 minutes to read • [Edit Online](#)

A custom allocation policy gives you more control over how devices are assigned to an IoT hub. This is accomplished by using custom code in an [Azure Function](#) that runs during provisioning to assign devices to an IoT hub. The device provisioning service calls your Azure Function code providing all relevant information about the device and the enrollment. Your function code is executed and returns the IoT hub information used to provisioning the device.

By using custom allocation policies, you define your own allocation policies when the policies provided by the Device Provisioning Service don't meet the requirements of your scenario.

For example, maybe you want to examine the certificate a device is using during provisioning and assign the device to an IoT hub based on a certificate property. Or, maybe you have information stored in a database for your devices and need to query the database to determine which IoT hub a device should be assigned to.

This article demonstrates an enrollment group with a custom allocation policy that uses an Azure Function written in C# to provision toaster devices using symmetric keys. Any device not recognized as a toaster device will not be provisioned to an IoT hub.

Devices will request provisioning using provisioning sample code included in the [Azure IoT C SDK](#).

In this tutorial you will do the following:

- Create a new Azure Function App to support a custom allocation function
- Create a new group enrollment using an Azure Function for the custom allocation policy
- Create device keys for two devices
- Set up the development environment for example device code from the [Azure IoT C SDK](#)
- Run the devices and verify that they are provisioned according to the custom allocation policy

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

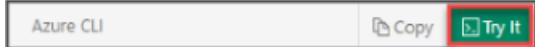
Prerequisites

- This article assumes you've completed the steps in [Set up IoT Hub Device Provisioning Service with the Azure portal](#) to create your IoT Hub and DPS instance.
- Latest version of [Git](#) installed.
- For a Windows development environment, [Visual Studio](#) 2019 with the '[Desktop development with C++](#)' workload enabled is required. Visual Studio 2015 and Visual Studio 2017 are also supported.
- For Linux or macOS, see the appropriate section in [Prepare your development environment](#) in the [Azure IoT C SDK](#) documentation.

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 the custom allocation function

In this section, you create an Azure function that implements your custom allocation policy. This function decides whether a device should be registered to your IoT Hub based on whether its registration ID contains the string prefix **contoso-toaster**.

1. Sign in to the [Azure portal](#). From your home page, select + **Create a resource**.
2. In the *Search the Marketplace* search box, type "Function App". From the drop-down list select **Function App**, and then select **Create**.
3. On **Function App** create page, under the **Basics** tab, enter the following settings for your new function app and select **Review + create**:

Subscription: If you have multiple subscriptions and the desired subscription is not selected, select the subscription you want to use.

Resource Group: This field allows you to create a new resource group, or choose an existing one to contain the function app. Choose the same resource group that contains the IoT hub you created for testing previously, for example, **TestResources**. By putting all related resources in a group together, you can manage them together.

Function App name: Enter a unique function app name. This example uses **contoso-function-app**.

Publish: Verify that **Code** is selected.

Runtime Stack: Select **.NET Core** from the drop-down.

Region: Select the same region as your resource group. This example uses **West US**.

NOTE

By default, Application Insights is enabled. Application Insights is not necessary for this article, but it might help you understand and investigate any issues you encounter with the custom allocation. If you prefer, you can disable Application Insights by selecting the **Monitoring** tab and then selecting **No** for **Enable Application Insights**.

The screenshot shows the 'Function App' creation wizard in the Azure portal. The top navigation bar includes 'Home > New > Function App > Function App'. The main title is 'Function App'. A blue banner at the top says 'Looking for the classic Function App create experience? →'. Below are tabs: 'Basics' (underlined), 'Hosting', 'Monitoring', 'Tags', and 'Review + create'. A descriptive text block states: 'Create a function app, which lets you group functions as a logical unit for easier management, deployment and sharing of resources. Functions lets you execute your code in a serverless environment without having to first create a VM or publish a web application.' The 'Project Details' section contains fields for 'Subscription' (set to 'Internal use'), 'Resource Group' (set to 'contoso-us-resource-group'), and 'Function App name' (set to 'contoso-function-app'). The 'Instance Details' section includes 'Runtime stack' (set to '.NET Core') and 'Region' (set to 'West US'). At the bottom are buttons for 'Review + create' (highlighted with a red box), '< Previous', and 'Next : Hosting >'.

4. On the **Summary** page, select **Create** to create the function app. Deployment may take several minutes. When it completes, select **Go to resource**.
5. On the left pane under **Functions** click **Functions** and then **+ Add** to add a new function.
6. On the templates page, select the **HTTP Trigger** tile, then select **Create Function**. A function named **HttpTrigger1** is created, and the portal displays the overview page for your function.
7. Click **Code + Test** for your new function. The portal displays the contents of the **run.csx** code file.
8. Replace the code for the **HttpTrigger1** function with the following code and select **Save**. Your custom allocation code is ready to be used.

```
#r "Newtonsoft.Json"

using System.Net;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Primitives;
using Newtonsoft.Json;
```

```

public static async Task<IActionResult> Run(HttpRequest req, ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a request.");

    // Get request body
    string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
    dynamic data = JsonConvert.DeserializeObject(requestBody);

    log.LogInformation("Request.Body:....");
    log.LogInformation(requestBody);

    // Get registration ID of the device
    string regId = data?.deviceRuntimeContext?.registrationId;

    string message = "Uncaught error";
    bool fail = false;
    ResponseObj obj = new ResponseObj();

    if (regId == null)
    {
        message = "Registration ID not provided for the device.";
        log.LogInformation("Registration ID : NULL");
        fail = true;
    }
    else
    {
        string[] hubs = data?.linkedHubs.ToObject<string[]>()[];

        // Must have hubs selected on the enrollment
        if (hubs == null)
        {
            message = "No hub group defined for the enrollment.";
            log.LogInformation("linkedHubs : NULL");
            fail = true;
        }
        else
        {
            // This is a Contoso Toaster
            if (regId.Contains("contoso-toaster"))
            {
                //Log IoT hubs configured for the enrollment
                foreach(string hubString in hubs)
                {
                    log.LogInformation("linkedHub : " + hubString);
                }

                obj.iotHubHostName = hubs[0];
                log.LogInformation("Selected hub : " + obj.iotHubHostName);
            }
            else
            {
                fail = true;
                message = "Unrecognized device registration.";
                log.LogInformation("Unknown device registration");
            }
        }
    }

    log.LogInformation("\nResponse");
    log.LogInformation((obj.iotHubHostName != null) ? JsonConvert.SerializeObject(obj) : message);

    return (fail)
        ? new BadRequestObjectResult(message)
        : (ActionResult)new OkObjectResult(obj);
}

public class ResponseObj
{
}

```

```

    public string iotHubHostName {get; set;}
}

```

- Just below the bottom of the `run.csx` code file, click **Logs** to monitor the logging from the custom allocation function.

Create the enrollment

In this section, you'll create a new enrollment group that uses the custom allocation policy. For simplicity, this article uses [Symmetric key attestation](#) with the enrollment. For a more secure solution, consider using [X.509 certificate attestation](#) with a chain of trust.

- Still on the [Azure portal](#), open your provisioning service.
- Select **Manage enrollments** on the left pane, and then select the **Add enrollment group** button at the top of the page.
- On **Add Enrollment Group**, enter the information in the table below and click the **Save** button.

FIELD	DESCRIPTION AND/OR SUGGESTED VALUE
Group name	Enter contoso-custom-allocated-devices . The enrollment group name is a case-insensitive string (up to 128 characters long) of alphanumeric characters plus the special characters: <code>'-'</code> , <code>'.'</code> , <code>'_'</code> , <code>:</code> . The last character must be alphanumeric or dash (<code>'-'</code>).
Attestation Type	Select Symmetric Key
Auto Generate Keys	This checkbox should already be checked.
Select how you want to assign devices to hubs	Select Custom (Use Azure Function)
Select the IoT hubs this group can be assigned to	Select the IoT hub you created previously when you completed the quick start.
Select Azure Function	Select the subscription that contains the function app you created. Then select the contoso-function-app and HttpTrigger1 for the function.

Add Enrollment Group

 Save

Group name *

contoso-custom-allocated-devices



Attestation Type ⓘ

Certificate Symmetric Key

Auto-generate keys ⓘ



Primary Key ⓘ

Enter your primary key

Secondary Key ⓘ

Enter your secondary key

IoT Edge device ⓘ

True False

Select how you want to assign devices to hubs ⓘ

Custom (Use Azure Function)



Select the IoT hubs this group can be assigned to: ⓘ

YourIoTHub.azure-devices.net



[Link a new IoT hub](#)

 You can use Azure Functions to write your own custom allocation policy. Select the Functions app you want to use, and the provisioning service will trigger the app via an HTTP PUT request. The app will return the desired IoT hub and initial twin for provisioning the device.

[Learn more.](#)

Select Azure Function

Subscription *

Azure IoT Content



Function App *

contoso-function-app



[Create a new function app](#)

Function *

HttpTrigger1



[Create a new function](#)

Select how you want device data to be handled on re-provisioning * ⓘ

Re-provision and migrate data



- After saving the enrollment, reopen it and make a note of the **Primary Key**. You must save the enrollment first to have the keys generated. This primary symmetric key will be used to generate unique device keys for devices that attempt provisioning later.

Derive unique device keys

Devices don't use the primary symmetric key directly. Instead you use the primary key to derive a device key for each device. In this section, you create two unique device keys. One key will be used for a simulated toaster device. The other key will be used for a simulated heat pump device. The keys generated will allow both devices to attempt a registration. Only one device registration ID will have a valid suffix to be accepted by custom allocation policy example code. As a result, one will be accepted and the other rejected

To derive the device key, you use the symmetric key you noted earlier to compute the [HMAC-SHA256](#) of the device registration ID for each device and convert the result into Base64 format. For more information on creating derived device keys with enrollment groups, see the group enrollments section of [Symmetric key attestation](#).

For the example in this article, use the following two device registration IDs with the code below to compute a device key for both devices:

- contoso-toaster-007
 - contoso-heatpump-088
-
- [Azure CLI](#)
 - [PowerShell](#)
 - [Bash](#)

The IoT extension for the Azure CLI provides the `compute-device-key` command for generating derived device keys. This command can be used on Windows-based or Linux systems, from PowerShell or a Bash shell.

Replace the value of `--key` argument with the **Primary Key** from your enrollment group.

```
az iot dps compute-device-key --key  
oiK770y7rBw8YB6IS6ukRChAw+Yq6GC61RMrPLSTi00tdI+XDu0LmLuNm11p+qv2I+adqGUdZHm46zXAQdZo0A== --registration-id  
contoso-toaster-007  
  
"JC8F96eayuQwz+PkE7IzjH2lIAjCUnAa61tDigBnSs="
```

```
az iot dps compute-device-key --key  
oiK770y7rBw8YB6IS6ukRChAw+Yq6GC61RMrPLSTi00tdI+XDu0LmLuNm11p+qv2I+adqGUdZHm46zXAQdZo0A== --registration-id  
contoso-heatpump-088  
  
"6uejA9PfkQgmYy1j8Zerp3kcbeVrGZ172YLa7VSnJzg="
```

Prepare an Azure IoT C SDK development environment

Devices will request provisioning using provisioning sample code included in the [Azure IoT C SDK](#).

In this section, you prepare the development environment used to build the [Azure IoT C SDK](#). The SDK includes the sample code for the simulated device. This simulated device will attempt provisioning during the device's boot sequence.

This section is oriented toward a Windows-based workstation. For a Linux example, see the set-up of the VMs in [How to provision for multitenancy](#).

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, which builds a version of the SDK specific to your development client platform. A Visual Studio solution for the simulated device will be generated in the `cmake` directory.

```
cmake -Dhsm_type_symm_key:BOOL=ON -Duse_prov_client:BOOL=ON ..
```

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

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

```
$ cmake -Dhsm_type_symm_key:BOOL=ON -Duse_prov_client:BOOL=ON ..
-- 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
```

Simulate the devices

In this section, you update a provisioning sample named `prov_dev_client_sample` located in the Azure IoT C SDK you set up previously.

This sample code simulates a device boot sequence that sends the provisioning request to your Device Provisioning Service instance. The boot sequence will cause the toaster device to be recognized and assigned to the IoT hub using the custom allocation policy.

1. In the Azure portal, select the **Overview** tab for your Device Provisioning Service and note down the *ID Scope* value.

2. In Visual Studio, open the `azure_iot_sdks.sln` solution file that was generated by running CMake earlier. The solution file should be in the following location:

```
azure-iot-sdk-c\cmake\azure_iot_sdks.sln
```

3. In Visual Studio's *Solution Explorer* window, navigate to the **Provision_Samples** folder. Expand the sample project named **prov_dev_client_sample**. Expand **Source Files**, and open **prov_dev_client_sample.c**.
4. Find the `id_scope` constant, and replace the value with your **ID Scope** value that you copied earlier.

```
static const char* id_scope = "0ne00002193";
```

5. Find the definition for the `main()` function in the same file. Make sure the `hsm_type` variable is set to `SECURE_DEVICE_TYPE_SYMMETRIC_KEY` as shown below:

```
SECURE_DEVICE_TYPE hsm_type;
//hsm_type = SECURE_DEVICE_TYPE_TPM;
//hsm_type = SECURE_DEVICE_TYPE_X509;
hsm_type = SECURE_DEVICE_TYPE_SYMMETRIC_KEY;
```

6. In the `main()` function, find the call to `Prov_Device_Register_Device()`. Just before that call, add the following lines of code that use `Prov_Device_Set_Provisioning_Payload()` to pass a custom JSON payload during provisioning. This can be used to provide more information to your custom allocation functions. This could also be used to pass the device type instead of examining the registration ID. For more information on sending and receiving custom data payloads with DPS, see [How to transfer payloads between devices and DPS](#).

```

// An example custom payload
const char* custom_json_payload = "
{\\"MyDeviceFirmwareVersion\\\":\\"12.0.2.5\\",\\"MyDeviceProvisioningVersion\\\":\\"1.0.0.0\\\"}";

prov_device_result = Prov_Device_Set_Provisioning_Payload(prov_device_handle, custom_json_payload);
if (prov_device_result != PROV_DEVICE_RESULT_OK)
{
    (void)printf("\r\nFailure setting provisioning payload: %s\r\n",
    MU_ENUM_TO_STRING(PROV_DEVICE_RESULT, prov_device_result));
}

```

7. Right-click the `prov_dev_client_sample` project and select **Set as Startup Project**.

Simulate the Contoso toaster device

1. To simulate the toaster device, find the call to `prov_dev_set_symmetric_key_info()` in `prov_dev_client_sample.c` which is commented out.

```

// Set the symmetric key if using they auth type
//prov_dev_set_symmetric_key_info("<symm_registration_id>", "<symmetric_Key>");
```

Uncomment the function call and replace the placeholder values (including the angle brackets) with the toaster registration ID and derived device key you generated previously. The key value `JC8F96eayuQwwz+PkE7IzjH2lIAjCUnAa61tDigBnSs=` shown below is only given as an example.

```

// Set the symmetric key if using they auth type
prov_dev_set_symmetric_key_info("contoso-toaster-007",
"JC8F96eayuQwwz+PkE7IzjH2lIAjCUnAa61tDigBnSs=");
```

Save the file.

2. On the Visual Studio menu, select **Debug > Start without debugging** to run the solution. In the prompt to rebuild the project, select **Yes**, to rebuild the project before running.

The following text is example logging output from the custom allocation function code running for the toaster device. Notice a hub is successfully selected for a toaster device. Also notice the `payload` member that contains the custom JSON content you added to the code. This is available for your code to use within the `deviceRuntimeContext`.

This logging is available by clicking **Logs** under the function code in the portal:

```

2020-09-23T11:44:37.505 [Information] Executing 'Functions.HttpTrigger1' (Reason='This function was
programmatically called via the host APIs.', Id=4596d45e-086f-4e86-929b-4a02814eee40)
2020-09-23T11:44:41.380 [Information] C# HTTP trigger function processed a request.
2020-09-23T11:44:41.381 [Information] Request.Body:...
2020-09-23T11:44:41.381 [Information] {"enrollmentGroup":{"enrollmentGroupId":"contoso-custom-
allocated-devices","attestation":{"type":"symmetricKey"},"capabilities":
{"iotEdge":false},"etag":"\"e8002126-0000-0100-0000-
5f6b2a570000\"","provisioningStatus":"enabled","reprovisionPolicy":
{"updateHubAssignment":true,"migrateDeviceData":true},"createdDateTimeUtc":"2020-09-
23T10:58:31.62286Z","lastUpdatedDateTimeUtc":"2020-09-
23T10:58:31.62286Z","allocationPolicy":"custom","iotHubs":["contoso-toasters-hub-1098.azure-
devices.net"],"customAllocationDefinition":{"webhookUrl":"https://contoso-function-
app.azurewebsites.net/api/HttpTrigger1?*****","apiVersion":"2019-03-31"}}, "deviceRuntimeContext":
{"registrationId":"contoso-toaster-007","symmetricKey":{}}, "payload":
{"MyDeviceFirmwareVersion":"12.0.2.5","MyDeviceProvisioningVersion":"1.0.0.0"}}, "linkedHubs":
["contoso-toasters-hub-1098.azure-devices.net"]}

2020-09-23T11:44:41.687 [Information] linkedHub : contoso-toasters-hub-1098.azure-devices.net
2020-09-23T11:44:41.688 [Information] Selected hub : contoso-toasters-hub-1098.azure-devices.net
2020-09-23T11:44:41.688 [Information] Response
2020-09-23T11:44:41.688 [Information] {"iotHubHostName":"contoso-toasters-hub-1098.azure-
devices.net"}
2020-09-23T11:44:41.689 [Information] Executed 'Functions.HttpTrigger1' (Succeeded, Id=4596d45e-086f-
4e86-929b-4a02814eee40, Duration=4347ms)

```

The following example device output shows the simulated toaster device successfully booting up and connecting to the provisioning service instance to be assigned to the toasters IoT hub by the custom allocation policy:

```

Provisioning API Version: 1.3.6

Registering Device

Provisioning Status: PROV_DEVICE_REG_STATUS_CONNECTED
Provisioning Status: PROV_DEVICE_REG_STATUS_ASSIGNING
Provisioning Status: PROV_DEVICE_REG_STATUS_ASSIGNING

Registration Information received from service: contoso-toasters-hub-1098.azure-devices.net,
deviceId: contoso-toaster-007

Press enter key to exit:

```

Simulate the Contoso heat pump device

- To simulate the heat pump device, update the call to `prov_dev_set_symmetric_key_info()` in `prov_dev_client_sample.c` again with the heat pump registration ID and derived device key you generated earlier. The key value `6uejA9PfkQgmYylj8Zerp3kcbeVrGZ172YLa7VSnJzg=` shown below is also only given as an example.

```

// Set the symmetric key if using they auth type
prov_dev_set_symmetric_key_info("contoso-heatpump-088",
"6uejA9PfkQgmYylj8Zerp3kcbeVrGZ172YLa7VSnJzg=");

```

Save the file.

- On the Visual Studio menu, select **Debug > Start without debugging** to run the solution. In the prompt to rebuild the project, select **Yes** to rebuild the project before running.

The following text is example logging output from the custom allocation function code running for the heat pump device. The custom allocation policy rejects this registration with a HTTP error 400 Bad Request. Notice the `payload` member that contains the custom JSON content you added to the code. This

is available for your code to use within the `deviceRuntimeContext`.

This logging is available by clicking **Logs** under the function code in the portal:

```
2020-09-23T11:50:23.652 [Information] Executing 'Functions.HttpTrigger1' (Reason='This function was programmatically called via the host APIs.', Id=2fa77f10-42f8-43fe-88d9-a8c01d4d3f68)
2020-09-23T11:50:23.653 [Information] C# HTTP trigger function processed a request.
2020-09-23T11:50:23.654 [Information] Request.Body:...
2020-09-23T11:50:23.654 [Information] {"enrollmentGroup":{"enrollmentGroupId":"contoso-custom-allocated-devices","attestation":{"type":"symmetricKey"},"capabilities":{"iotEdge":false}, "etag": "\"e8002126-0000-0100-0000-5f6b2a570000\""}, "provisioningStatus": "enabled", "reprovisionPolicy": {"updateHubAssignment": true, "migrateDeviceData": true}, "createdDateTimeUtc": "2020-09-23T10:58:31.62286Z", "lastUpdatedDateTimeUtc": "2020-09-23T10:58:31.62286Z", "allocationPolicy": "custom", "iotHubs": ["contoso-toasters-hub-1098.azure-devices.net"], "customAllocationDefinition": {"webhookUrl": "https://contoso-function-app.azurewebsites.net/api/HttpTrigger1?*****", "apiVersion": "2019-03-31"}}, "deviceRuntimeContext": {"registrationId": "contoso-heatpump-088", "symmetricKey": {}, "payload": {"MyDeviceFirmwareVersion": "12.0.2.5", "MyDeviceProvisioningVersion": "1.0.0.0"}}, "linkedHubs": ["contoso-toasters-hub-1098.azure-devices.net"]}
2020-09-23T11:50:23.654 [Information] Unknown device registration
2020-09-23T11:50:23.654 [Information] Response
2020-09-23T11:50:23.654 [Information] Unrecognized device registration.
2020-09-23T11:50:23.655 [Information] Executed 'Functions.HttpTrigger1' (Succeeded, Id=2fa77f10-42f8-43fe-88d9-a8c01d4d3f68, Duration=11ms)
```

The following example device output shows the simulated heat pump device booting up and connecting to the provisioning service instance to attempt registration to an IoT hub using the custom allocation policy. This fails with error (`Custom allocation failed with status code: 400`) since the custom allocation policy was designed to only allow toaster devices:

```
Provisioning API Version: 1.3.7

Registering Device

Provisioning Status: PROV_DEVICE_REG_STATUS_CONNECTED
Provisioning Status: PROV_DEVICE_REG_STATUS_ASSIGNING
Provisioning Status: PROV_DEVICE_REG_STATUS_ASSIGNING
Error: Time:Wed Sep 23 13:06:01 2020 File:d:\testing\azure-iot-sdk-c\provisioning_client\src\prov_device_ll_client.c Func:_prov_transport_process_json_reply Line:658
Provisioning Failure: OperationId: 4.eb89f3e8407a3711.2525bd34-02e9-4e91-a9c0-4dbc4ad5de66 - Date: 2020-09-23T17:05:58.2363145Z - Msg: Custom allocation failed with status code: 400
Error: Time:Wed Sep 23 13:06:01 2020 File:d:\testing\azure-iot-sdk-c\provisioning_client\src\prov_transport_mqtt_common.c Func:_prov_transport_common_mqtt_dowork Line:1014 Unable to process registration reply.
Error: Time:Wed Sep 23 13:06:01 2020 File:d:\testing\azure-iot-sdk-c\provisioning_client\src\prov_device_ll_client.c Func:_on_transport_registration_data Line:770
Failure retrieving data from the provisioning service

Failure registering device: PROV_DEVICE_RESULT_DEV_AUTH_ERROR
Press enter key to exit:
```

Clean up resources

If you plan to continue working with the resources created in this article, you can leave them. If you don't plan to continue using the resources, use the following steps to delete all of the resources created in this article to avoid unnecessary charges.

The steps here assume you created all resources in this article as instructed in the same resource group named **contoso-us-resource-group**.

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 you want to keep, only delete the IoT Hub resource itself instead of deleting the resource group.

To delete the 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 resources, **contoso-us-resource-group**.
3. To the right of your resource group in the result list, select ... then **Delete resource group**.
4. You'll 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

For a more in-depth custom allocation policy example, see

[How to use custom allocation policies](#)

- To learn more Reprovisioning, see

[IoT Hub Device reprovisioning concepts](#)

- To learn more Deprovisioning, see

[How to deprovision devices that were previously autoprovisioned](#)

Tutorial: Provision multiple X.509 devices using enrollment groups

8/22/2022 • 26 minutes to read • [Edit Online](#)

In this tutorial, you'll learn how to provision groups of IoT devices that use X.509 certificates for authentication. Sample device code from the [Azure IoT C SDK](#) will be executed on your development machine to simulate provisioning of X.509 devices. On real devices, device code would be deployed and run from the IoT device.

The Azure IoT Hub Device Provisioning Service supports two types of enrollments for provisioning devices:

- [Enrollment groups](#): Used to enroll multiple related devices. This tutorial demonstrates provisioning with enrollment groups.
- [Individual Enrollments](#): Used to enroll a single device.

The Azure IoT Hub Device Provisioning Service supports three forms of authentication for provisioning devices:

- [X.509 certificates](#). This tutorial demonstrates X.509 certificate attestation.
- [Trusted platform module \(TPM\)](#)
- [Symmetric keys](#)

This tutorial uses the [custom HSM sample](#), which provides a stub implementation for interfacing with hardware-based secure storage. A [Hardware Security Module \(HSM\)](#) is used for secure, hardware-based storage of device secrets. An HSM can be used with symmetric key, X.509 certificate, or TPM attestation to provide secure storage for secrets. Hardware-based storage of device secrets isn't required, but it is strongly recommended to help protect sensitive information like your device certificate's private key.

In this tutorial, you'll complete the following objectives:

- Create a certificate chain of trust to organize a set of devices using X.509 certificates.
- Complete proof of possession with a signing certificate used with the certificate chain.
- Create a new group enrollment that uses the certificate chain.
- Set up the development environment for provisioning a device using code from the [Azure IoT C SDK](#).
- Provision a device using the certificate chain with the custom Hardware Security Module (HSM) sample in the SDK.

Prerequisites

- If you don't have an Azure subscription, create a [free account](#) before you begin.
- Complete the steps in [Set up IoT Hub Device Provisioning Service with the Azure portal](#).

The following prerequisites are for a Windows development environment used to simulate the devices. For Linux or macOS, see the appropriate section in [Prepare your development environment](#) in the SDK documentation.

- Install [Visual Studio](#) 2022 with the '[Desktop development with C++](#)' workload enabled. Visual Studio 2015, Visual Studio 2017, and Visual Studio 19 are also supported.
- Install the latest [CMake build system](#). Make sure you check the option that adds the CMake executable to your path.

IMPORTANT

Confirm 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. Also, be aware that older versions of the CMake build system fail to generate the solution file used in this tutorial. Make sure to use the latest version of CMake.

- Install the latest version of [Git](#). Make sure that Git is added to the environment variables accessible to the command window. See [Software Freedom Conservancy's Git client tools](#) for the latest version of `git` tools to install, which includes *Git Bash*, the command-line app that you can use to interact with your local Git repository.
- Make sure [OpenSSL](#) is installed on your machine. On Windows, your installation of Git includes an installation of OpenSSL. You can access OpenSSL from the Git Bash prompt. To verify that OpenSSL is installed, open a Git Bash prompt and enter `openssl version`.

NOTE

Unless you're familiar with OpenSSL and already have it installed on your Windows machine, we recommend using OpenSSL from the Git Bash prompt. Alternatively, you can choose to download the source code and build OpenSSL. To learn more, see the [OpenSSL Downloads](#) page. Or, you can download OpenSSL pre-built from a third-party. To learn more, see the [OpenSSL wiki](#). Microsoft makes no guarantees about the validity of packages downloaded from third-parties. If you do choose to build or download OpenSSL make sure that the OpenSSL binary is accessible in your path and that the `OPENSSL_CNF` environment variable is set to the path of your `openssl.cnf` file.

- Open both a Windows command prompt and a Git Bash prompt.

The steps in this tutorial assume that you're using a Windows machine and the OpenSSL installation that is installed as part of Git. You'll use the Git Bash prompt to issue OpenSSL commands and the Windows command prompt for everything else. If you're using Linux, you can issue all commands from a Bash shell.

Prepare the Azure IoT C SDK development environment

In this section, you'll prepare a development environment used to build the [Azure IoT C SDK](#). The SDK includes sample code and tools used by devices provisioning with DPS.

1. Open a web browser, and go to the [Release page of the Azure IoT C SDK](#).
2. Select the **Tags** tab at the top of the page.
3. Copy the tag name for the latest release of the Azure IoT C SDK.
4. In your Windows command prompt, run the following commands to clone the latest release of the [Azure IoT C SDK](#) GitHub repository. Replace `<release-tag>` with the tag you copied in the previous step.

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

This operation could take several minutes to complete.

5. When the operation is complete, run the following commands from the `azure-iot-sdk-c` directory:

```
mkdir cmake  
cd cmake
```

6. The code sample uses an X.509 certificate to provide attestation via X.509 authentication. Run the following command to build a version of the SDK specific to your development platform that includes the device provisioning client. A Visual Studio solution for the simulated device is generated in the `cmake` directory.

When specifying the path used with `-Dhsm_custom_lib` in the command below, make sure to use the absolute path to the library in the `cmake` directory you previously created. The path shown below assumes that you cloned the C SDK in the root directory of the C drive. If you used another directory, adjust the path accordingly.

```
cmake -Duse_prov_client:BOOL=ON -Dhsm_custom_lib=c:/azure-iot-sdk-c/cmake/provisioning_client/samples/custom_hsm_example/Debug/custom_hsm_example.lib ..
```

TIP

If `cmake` doesn't find your C++ compiler, you may get build errors while running the above command. If that happens, try running the command in the [Visual Studio command prompt](#).

7. When the build succeeds, the last few output lines look similar to the following output:

```
cmake -Duse_prov_client:BOOL=ON -Dhsm_custom_lib=c:/azure-iot-sdk-c/cmake/provisioning_client/samples/custom_hsm_example/Debug/custom_hsm_example.lib ..  
-- Building for: Visual Studio 17 2022  
-- Selecting Windows SDK version 10.0.19041.0 to target Windows 10.0.22000.  
-- The C compiler identification is MSVC 19.32.31329.0  
-- The CXX compiler identification is MSVC 19.32.31329.0  
  
...  
  
-- Configuring done  
-- Generating done  
-- Build files have been written to: C:/azure-iot-sdk-c/cmake
```

Create an X.509 certificate chain

In this section, you'll generate an X.509 certificate chain of three certificates for testing each device with this tutorial. The certificates have the following hierarchy.



Root certificate: You'll complete [proof of possession](#) to verify the root certificate. This verification enables DPS to trust that certificate and verify certificates signed by it.

Intermediate certificate: It's common to use intermediate certificates to group devices logically by product lines, company divisions, or other criteria. This tutorial uses a certificate chain with one intermediate certificate, but in a production scenario you may have several. The intermediate certificate in this chain is signed by the root certificate. This certificate is provided to the enrollment group created in DPS to logically group a set of devices.

This configuration allows managing a whole group of devices that have device certificates signed by the same intermediate certificate.

Device certificates: The device certificates (sometimes called leaf certificates) will be signed by the intermediate certificate and stored on the device along with its private key. Ideally these sensitive items would be stored securely with an HSM. Each device presents its certificate and private key, along with the certificate chain, when attempting provisioning.

Set up the X.509 OpenSSL environment

In this section, you'll create the Openssl configuration files, directory structure, and other files used by the Openssl commands.

1. In your Git Bash command prompt, navigate to a folder where you want to generate the X.509 certificates and keys for this tutorial.
2. Create an OpenSSL configuration file for your root CA certificate. OpenSSL configuration files contain policies and definitions that are consumed by OpenSSL commands. Copy and paste the following text into a file named *openssl_root_ca.cnf*.

```
# OpenSSL root CA configuration file.

[ ca ]
default_ca = CA_default

[ CA_default ]
# Directory and file locations.
dir          = .
certs        = $dir/certs
crl_dir      = $dir/crl
new_certs_dir = $dir/newcerts
database     = $dir/index.txt
serial       = $dir/serial
RANDFILE     = $dir/private/.rand

# The root key and root certificate.
private_key   = $dir/private/azure-iot-test-only.root.ca.key.pem
certificate   = $dir/certs/azure-iot-test-only.root.ca.cert.pem

# For certificate revocation lists.
crlnumber    = $dir/crlnumber
crl         = $dir/crl/azure-iot-test-only.intermediate.crl.pem
crl_extensions = crl_ext
default_crl_days = 30

# SHA-1 is deprecated, so use SHA-2 instead.
default_md    = sha256

name_opt      = ca_default
cert_opt      = ca_default
default_days  = 375
preserve      = no
policy        = policy_loose

[ policy_strict ]
# The root CA should only sign intermediate certificates that match.
countryName   = optional
stateOrProvinceName = optional
organizationName = optional
organizationalUnitName = optional
commonName    = supplied
emailAddress  = optional

[ policy_loose ]
# Allow the intermediate CA to sign a more diverse range of certificates.
countryName   = optional
```

```

countryName          = optional
stateOrProvinceName = optional
localityName         = optional
organizationName    = optional
organizationalUnitName = optional
commonName           = supplied
emailAddress         = optional

[ req ]
default_bits        = 2048
distinguished_name = req_distinguished_name
string_mask         = utf8only

# SHA-1 is deprecated, so use SHA-2 instead.
default_md          = sha256

# Extension to add when the -x509 option is used.
x509_extensions     = v3_ca

[ req_distinguished_name ]
# See <https://en.wikipedia.org/wiki/Certificate_signing_request>.
countryName          = Country Name (2 letter code)
stateOrProvinceName = State or Province Name
localityName         = Locality Name
0.organizationName   = Organization Name
organizationalUnitName = Organizational Unit Name
commonName           = Common Name
emailAddress         = Email Address

# Optionally, specify some defaults.
countryName_default  = US
stateOrProvinceName_default = WA
localityName_default  =
0.organizationName_default = My Organization
organizationalUnitName_default =
emailAddress_default  =

[ v3_ca ]
# Extensions for a typical CA.
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid(always,issuer)
basicConstraints = critical, CA:true
keyUsage = critical, digitalSignature, cRLSign, keyCertSign

[ v3_intermediate_ca ]
# Extensions for a typical intermediate CA.
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid(always,issuer)
basicConstraints = critical, CA:true
keyUsage = critical, digitalSignature, cRLSign, keyCertSign

[ usr_cert ]
# Extensions for client certificates.
basicConstraints = CA:FALSE
nsComment = "OpenSSL Generated Client Certificate"
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid,issuer
keyUsage = critical, nonRepudiation, digitalSignature, keyEncipherment
extendedKeyUsage = clientAuth

[ server_cert ]
# Extensions for server certificates.
basicConstraints = CA:FALSE
nsComment = "OpenSSL Generated Server Certificate"
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid,issuer:always
keyUsage = critical, digitalSignature, keyEncipherment
extendedKeyUsage = serverAuth

```

```

[ cri_ext ]
# Extension for CRLs.
authorityKeyIdentifier=keyid(always

[ ocsp ]
# Extension for OCSP signing certificates.
basicConstraints = CA:FALSE
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid,issuer
keyUsage = critical, digitalSignature
extendedKeyUsage = critical, OCSPSigning

```

3. Create an OpenSSL configuration file to use for intermediate and device certificates. Copy and paste the following text into a file named *openssl_device_intermediate_ca.cnf*.

```

# OpenSSL root CA configuration file.

[ ca ]
default_ca = CA_default

[ CA_default ]
# Directory and file locations.
dir          = .
certs        = $dir/certs
crl_dir      = $dir/crl
new_certs_dir = $dir/newcerts
database     = $dir/index.txt
serial       = $dir/serial
RANDFILE     = $dir/private/.rand

# The root key and root certificate.
private_key   = $dir/private/azure-iot-test-only.intermediate.key.pem
certificate   = $dir/certs/azure-iot-test-only.intermediate.cert.pem

# For certificate revocation lists.
crlnumber    = $dir/crlnumber
crl         = $dir/crl/azure-iot-test-only.intermediate.crl.pem
crl_extensions = crl_ext
default_crl_days = 30

# SHA-1 is deprecated, so use SHA-2 instead.
default_md    = sha256

name_opt      = ca_default
cert_opt      = ca_default
default_days  = 375
preserve      = no
policy        = policy_loose

[ policy_strict ]
# The root CA should only sign intermediate certificates that match.
countryName   = optional
stateOrProvinceName = optional
organizationName = optional
organizationalUnitName = optional
commonName    = supplied
emailAddress  = optional

[ policy_loose ]
# Allow the intermediate CA to sign a more diverse range of certificates.
countryName   = optional
stateOrProvinceName = optional
localityName   = optional
organizationName = optional
organizationalUnitName = optional
commonName    = supplied
emailAddress  = optional

```

```

[ req ]
default_bits      = 2048
distinguished_name = req_distinguished_name
string_mask       = utf8only

# SHA-1 is deprecated, so use SHA-2 instead.
default_md        = sha256

# Extension to add when the -x509 option is used.
x509_extensions   = v3_ca

[ req_distinguished_name ]
# See <https://en.wikipedia.org/wiki/Certificate_signing_request>.
countryName          = Country Name (2 letter code)
stateOrProvinceName = State or Province Name
localityName         = Locality Name
0.organizationName   = Organization Name
organizationalUnitName = Organizational Unit Name
commonName           = Common Name
emailAddress         = Email Address

# Optionally, specify some defaults.
countryName_default = US
stateOrProvinceName_default = WA
localityName_default =
0.organizationName_default = My Organization
organizationalUnitName_default =
emailAddress_default =

[ v3_ca ]
# Extensions for a typical CA.
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid(always,issuer
basicConstraints = critical, CA:true
keyUsage = critical, digitalSignature, cRLSign, keyCertSign

[ v3_intermediate_ca ]
# Extensions for a typical intermediate CA.
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid(always,issuer
basicConstraints = critical, CA:true
keyUsage = critical, digitalSignature, cRLSign, keyCertSign

[ usr_cert ]
# Extensions for client certificates.
basicConstraints = CA:FALSE
nsComment = "OpenSSL Generated Client Certificate"
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid,issuer
keyUsage = critical, nonRepudiation, digitalSignature, keyEncipherment
extendedKeyUsage = clientAuth

[ server_cert ]
# Extensions for server certificates.
basicConstraints = CA:FALSE
nsComment = "OpenSSL Generated Server Certificate"
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid,issuer:always
keyUsage = critical, digitalSignature, keyEncipherment
extendedKeyUsage = serverAuth

[ crl_ext ]
# Extension for CRLs.
authorityKeyIdentifier=keyid:always

[ ocsp ]
# Extension for OCSP signing certificates.
basicConstraints = CA:FALSE

```

```
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid,issuer
keyUsage = critical, digitalSignature
extendedKeyUsage = critical, OCSPSigning
```

4. Create the directory structure, the database file (index.txt), and the serial number file (serial) used by OpenSSL commands in this tutorial:

```
mkdir certs csr newcerts private
touch index.txt
openssl rand -hex 16 > serial
```

Create the root CA certificate

Run the following commands to create the root CA private key and the root CA certificate. You'll use this certificate and key to sign your intermediate certificate.

1. Create the root CA private key:

```
openssl genrsa -aes256 -passout pass:1234 -out ./private/azure-iot-test-only.root.ca.key.pem 4096
```

2. Create the root CA certificate:

- [Windows](#)
- [Linux](#)

```
openssl req -new -x509 -config ./openssl_root_ca.cnf -passin pass:1234 -key ./private/azure-iot-test-only.root.ca.key.pem -subj '//CN=Azure IoT Hub CA Cert Test Only' -days 30 -sha256 -extensions v3_ca -out ./certs/azure-iot-test-only.root.ca.cert.pem
```

IMPORTANT

The extra forward slash given for the subject name (`//CN=Azure IoT Hub CA Cert Test Only`) is only required to escape the string with Git on Windows platforms.

3. Examine the root CA certificate:

```
openssl x509 -noout -text -in ./certs/azure-iot-test-only.root.ca.cert.pem
```

Observe that the **Issuer** and the **Subject** are both the root CA.

```
Certificate:  
Data:  
    Version: 3 (0x2)  
    Serial Number:  
        1d:93:13:0e:54:07:95:1d:8c:57:4f:12:14:b9:5e:5f:15:c3:a9:d4  
    Signature Algorithm: sha256WithRSAEncryption  
    Issuer: CN = Azure IoT Hub CA Cert Test Only  
    Validity  
        Not Before: Jun 20 22:52:23 2022 GMT  
        Not After : Jul 20 22:52:23 2022 GMT  
    Subject: CN = Azure IoT Hub CA Cert Test Only  
    Subject Public Key Info:  
        Public Key Algorithm: rsaEncryption  
        RSA Public-Key: (4096 bit)
```

Create the intermediate CA certificate

Run the following commands to create the intermediate CA private key and the intermediate CA certificate. You'll use this certificate and key to sign your device certificate(s).

1. Create the intermediate CA private key:

```
openssl genrsa -aes256 -passout pass:1234 -out ./private/azure-iot-test-only.intermediate.key.pem  
4096
```

2. Create the intermediate CA certificate signing request (CSR):

- [Windows](#)
- [Linux](#)

```
openssl req -new -sha256 -passin pass:1234 -config ./openssl_device_intermediate_ca.cnf -subj  
'//CN=Azure IoT Hub Intermediate Cert Test Only' -key ./private/azure-iot-test-  
only.intermediate.key.pem -out ./csr/azure-iot-test-only.intermediate.csr.pem
```

IMPORTANT

The extra forward slash given for the subject name (`//CN=Azure IoT Hub Intermediate Cert Test Only`) is only required to escape the string with Git on Windows platforms.

3. Sign the intermediate certificate with the root CA certificate

```
openssl ca -batch -config ./openssl_root_ca.cnf -passin pass:1234 -extensions v3_intermediate_ca -  
days 30 -notext -md sha256 -in ./csr/azure-iot-test-only.intermediate.csr.pem -out ./certs/azure-iot-  
test-only.intermediate.cert.pem
```

4. Examine the intermediate CA certificate:

```
openssl x509 -noout -text -in ./certs/azure-iot-test-only.intermediate.cert.pem
```

Observe that the **Issuer** is the root CA, and the **Subject** is the intermediate CA.

```
Certificate:  
Data:  
    Version: 3 (0x2)  
    Serial Number:  
        d9:55:87:57:41:c8:4c:47:6c:ee:ba:83:5d:ae:db:39  
    Signature Algorithm: sha256WithRSAEncryption  
    Issuer: CN = Azure IoT Hub CA Cert Test Only  
    Validity  
        Not Before: Jun 20 22:54:01 2022 GMT  
        Not After : Jul 20 22:54:01 2022 GMT  
    Subject: CN = Azure IoT Hub Intermediate Cert Test Only  
    Subject Public Key Info:  
        Public Key Algorithm: rsaEncryption  
        RSA Public-Key: (4096 bit)
```

Create the device certificates

In this section, you create two device certificates and their full chain certificates. The full chain certificate contains the device certificate, the intermediate CA certificate, and the root CA certificate. The device must present its full chain certificate when it registers with DPS.

1. Create the first device private key.

```
openssl genrsa -out ./private/device-01.key.pem 4096
```

2. Create the device certificate CSR.

The subject common name (CN) of the device certificate must be set to the [registration ID](#) that your device will use to register with DPS. The registration ID is a case-insensitive string of alphanumeric characters plus the special characters: `'-'`, `'.'`, `'_'`, `'::'`. The last character must be alphanumeric or dash (`'-'`). The common name must adhere to this format. DPS supports registration IDs up to 128 characters long; however, the maximum length of the subject common name in an X.509 certificate is 64 characters. The registration ID, therefore, is limited to 64 characters when using X.509 certificates. For group enrollments, the registration ID is also used as the device ID in IoT Hub.

The subject common name is set using the `-subj` parameter. In the following command, the common name is set to **device-01**.

- [Windows](#)
- [Linux](#)

```
openssl req -config ./openssl_device_intermediate_ca.cnf -key ./private/device-01.key.pem -subj  
'//CN=device-01' -new -sha256 -out ./csr/device-01.csr.pem
```

IMPORTANT

The extra forward slash given for the subject name (`//CN=device-01`) is only required to escape the string with Git on Windows platforms.

3. Sign the device certificate.

```
openssl ca -batch -config ./openssl_device_intermediate_ca.cnf -passin pass:1234 -extensions usr_cert  
-days 30 -notext -md sha256 -in ./csr/device-01.csr.pem -out ./certs/device-01.cert.pem
```

4. Examine the device certificate:

```
openssl x509 -noout -text -in ./certs/device-01.cert.pem
```

Observe that the **Issuer** is the intermediate CA, and the **Subject** is the device registration ID, `device-01`.

```
Certificate:
Data:
Version: 3 (0x2)
Serial Number:
d9:55:87:57:41:c8:4c:47:6c:ee:ba:83:5d:ae:db:3a
Signature Algorithm: sha256WithRSAEncryption
Issuer: CN = Azure IoT Hub Intermediate Cert Test Only
Validity
Not Before: Jun 20 22:55:39 2022 GMT
Not After : Jul 20 22:55:39 2022 GMT
Subject: CN = device-01
Subject Public Key Info:
Public Key Algorithm: rsaEncryption
RSA Public-Key: (4096 bit)
```

5. The device must present the full certificate chain when it authenticates with DPS. Use the following command to create the certificate chain:

```
cat ./certs/device-01.cert.pem ./certs/azure-iot-test-only.intermediate.cert.pem ./certs/azure-iot-test-only.root.ca.cert.pem > ./certs/device-01-full-chain.cert.pem
```

6. Open the certificate chain file, `./certs/device-01-full-chain.cert.pem`, in a text editor to examine it. The certificate chain text contains the full chain of all three certificates. You'll use this text as the certificate chain with in the custom HSM device code later in this tutorial for `device-01`.

The full chain text has the following format:

```
-----BEGIN CERTIFICATE-----
<Text for the device certificate includes public key>
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
<Text for the intermediate certificate includes public key>
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
<Text for the root certificate includes public key>
-----END CERTIFICATE-----
```

7. To create the private key, X.509 certificate, and full chain certificate for the second device, copy and paste this script into your GitBash command prompt. To create certificates for more devices, you can modify the `registration_id` variable declared at the beginning of the script.

```
registration_id=device-02
echo $registration_id
openssl genrsa -out ./private/${registration_id}.key.pem 4096
openssl req -config ./openssl_device_intermediate_ca.cnf -key ./private/${registration_id}.key.pem -
subj "//CN=$registration_id" -new -sha256 -out ./csr/${registration_id}.csr.pem
openssl ca -batch -config ./openssl_device_intermediate_ca.cnf -passin pass:1234 -extensions usr_cert
-days 30 -notext -md sha256 -in ./csr/${registration_id}.csr.pem -out
./certs/${registration_id}.cert.pem
cat ./certs/${registration_id}.cert.pem ./certs/azure-iot-test-only.intermediate.cert.pem
./certs/azure-iot-test-only.root.ca.cert.pem > ./certs/${registration_id}-full-chain.cert.pem
```

NOTE

This script uses the registration ID as the base filename for the private key and certificate files. If your registration ID contains characters that aren't valid filename characters, you'll need to modify the script accordingly.

WARNING

The text for the certificates only contains public key information.

However, the device must also have access to the private key for the device certificate. This is necessary because the device must perform verification using that key at runtime when it attempts to provision. The sensitivity of this key is one of the main reasons it is recommended to use hardware-based storage in a real HSM to help secure private keys.

You'll use the following files in the rest of this tutorial:

CERTIFICATE	FILE	DESCRIPTION
root CA certificate.	<i>certs/azure-iot-test-only.root.ca.cert.pem</i>	Will be uploaded to DPS and verified.
intermediate CA certificate	<i>certs/azure-iot-test-only.intermediate.cert.pem</i>	Will be used to create an enrollment group in DPS.
device-01 private key	<i>private/device-01.key.pem</i>	Used by the device to verify ownership of the device certificate during authentication with DPS.
device-01 full chain certificate	<i>certs/device-01-full-chain.cert.pem</i>	Presented by the device to authenticate and register with DPS.
device-02 private key	<i>private/device-02.key.pem</i>	Used by the device to verify ownership of the device certificate during authentication with DPS.
device-02 full chain certificate	<i>certs/device-01-full-chain.cert.pem</i>	Presented by the device to authenticate and register with DPS.

Verify ownership of the root certificate

For DPS to be able to validate the device's certificate chain during authentication, you must upload and verify ownership of the root CA certificate. Because you created the root CA certificate in the last section, you'll auto-verify that it's valid when you upload it. Alternatively, you can do manual verification of the certificate if you're using a CA certificate from a 3rd-party. To learn more about verifying CA certificates, see [How to do proof-of-possession for X.509 CA certificates](#).

To add the root CA certificate to your DPS instance, follow these steps:

1. Sign in to the [Azure portal](#), select the **All resources** button on the left-hand menu and open your Device Provisioning Service instance.
2. Open **Certificates** from the left-hand menu and then select **+ Add** at the top of the panel to add a new certificate.
3. Enter a friendly display name for your certificate. Browse to the location of the root CA certificate file

`certs/azure-iot-test-only.root.ca.cert.pem`. Select **Upload**.

4. Select the box next to **Set certificate status to verified on upload**.

Certificate name * ⓘ
azure-iot-test-only-root

Certificate .pem or .cer file. ⓘ
"azure-iot-test-only.root.ca.cert.pem"

Set certificate status to verified on upload ⓘ

ⓘ We'll verify this certificate automatically, with no manual verification steps required. [Learn more](#)

Save

5. Select **Save**.

6. Make sure your certificate is shown in the certificate tab with a status of *Verified*.

contoso-dps-2 | Certificates ⚙️

Azure IoT Hub Device Provisioning Service (DPS)

Upload and manage certificates for authenticating device enrollments here. [Learn more](#)

ⓘ When you upload a new certificate, we can automatically verify it for you (note that the status of existing certificates won't change). [Learn more](#)

Add **Refresh**

Name	Created	Expires	Subject	Thumbprint	Status
azure-iot-test-only-root	12/31/2000	7/20/2022	Azure IoT Hub CA Cert...	5A28D3716EFE455676...	Verified

Update the certificate store on Windows-based devices

On non-Windows devices, you can pass the certificate chain from the code as the certificate store.

On Windows-based devices, you must add the signing certificates (root and intermediate) to a Windows [certificate store](#). Otherwise, the signing certificates won't be transported to DPS by a secure channel with Transport Layer Security (TLS).

TIP

It's also possible to use OpenSSL instead of secure channel (Schannel) with the C SDK. For more information on using OpenSSL, see [Using OpenSSL in the SDK](#).

To add the signing certificates to the certificate store in Windows-based devices:

1. In a Git bash prompt, convert your signing certificates to `.pfx` as follows.

Root CA certificate:

```
openssl pkcs12 -inkey ./private/azure-iot-test-only.root.ca.key.pem -in ./certs/azure-iot-test-only.root.ca.cert.pem -export -passin pass:1234 -passout pass:1234 -out ./certs/root.pfx
```

Intermediate CA certificate:

```
openssl pkcs12 -inkey ./private/azure-iot-test-only.intermediate.key.pem -in ./certs/azure-iot-test-only.intermediate.cert.pem -export -passin pass:1234 -passout pass:1234 -out ./certs/intermediate.pfx
```

2. Right-click the Windows **Start** button. Then select **Run**. Enter `certmgr.msc` and select **Ok** to start certificate manager MMC snap-in.
3. In certificate manager, under **Certificates - Current User**, select **Trusted Root Certification Authorities**. Then on the menu, select **Action > All Tasks > Import** to import `root.pfx`.
 - Make sure to search by **Personal information Exchange (.pfx)**
 - Use `1234` as the password.
 - Place the certificate in the **Trusted Root Certification Authorities** certificate store.
4. In certificate manager, under **Certificates - Current User**, select **Intermediate Certification Authorities**. Then on the menu, select **Action > All Tasks > Import** to import `intermediate.pfx`.
 - Make sure to search by **Personal information Exchange (.pfx)**
 - Use `1234` as the password.
 - Place the certificate in the **Intermediate Certification Authorities** certificate store.

Your signing certificates are now trusted on the Windows-based device and the full chain can be transported to DPS.

Create an enrollment group

1. From your DPS instance in Azure portal, select the **Manage enrollments** tab. then select the **Add enrollment group** button at the top.
2. In the **Add Enrollment Group** panel, enter the following information, then select **Save**.

FIELD	VALUE
Group name	For this tutorial, enter <code>custom-hsm-x509-devices</code> . The enrollment group name is a case-insensitive string (up to 128 characters long) of alphanumeric characters plus the special characters: <code>'-'</code> , <code>'.'</code> , <code>'_'</code> , <code>:</code> . The last character must be alphanumeric or dash (<code>'-'</code>).
Attestation Type	Select Certificate
IoT Edge device	Select False
Certificate Type	Select Intermediate Certificate

FIELD	VALUE
Primary certificate .pem or .cer file	Navigate to the intermediate certificate that you created earlier (<code>./certs/azure-iot-test-only.intermediate.cert.pem</code>). This intermediate certificate is signed by the root certificate that you already uploaded and verified. DPS trusts that root once it's verified. DPS can verify that the intermediate provided with this enrollment group is truly signed by the trusted root. DPS will trust each intermediate truly signed by that root certificate, and therefore be able to verify and trust leaf certificates signed by the intermediate.

 Add Enrollment Group X

 Save

Group name *
 ✓

Attestation Type (i)
 Certificate Symmetric Key

IoT Edge device (i)
 True False

Certificate Type (i)
 CA Certificate Intermediate Certificate

Primary Certificate .pem or .cer file (i)
 [Remove]

Secondary Certificate .pem or .cer file (i)
 [Remove]

Select how you want to assign devices to hubs (i)

Select the IoT hubs this group can be assigned to: (i)

Configure the provisioning device code

In this section, you update the sample code with your Device Provisioning Service instance information. If a device is authenticated, it will be assigned to an IoT hub linked to the Device Provisioning Service instance configured in this section.

1. In the Azure portal, select the **Overview** tab for your Device Provisioning Service instance and note the **ID Scope** value.

The screenshot shows the Azure IoT Hub Device Provisioning Service (DPS) Overview page for a resource group named 'contoso-rg'. The 'ID Scope' field is highlighted with a red box. Other visible details include:

- Resource group (move):** contoso-rg
- Status:** Active
- Location:** Central US
- Subscription (move):** IoT Documentation
- Subscription ID:** One005ED5E1
- Pricing and scale tier:** S1
- Automatic failover enabled:** Yes

2. Launch Visual Studio and open the new solution file that was created in the `cmake` directory you created in the root of the azure-iot-sdk-c git repository. The solution file is named `azure_iot_sdks.sln`.
3. In Solution Explorer for Visual Studio, navigate to **Provision_Samples > prov_dev_client_sample > Source Files** and open `prov_dev_client_sample.c`.
4. Find the `id_scope` constant, and replace the value with your **ID Scope** value that you copied earlier. For example:

```
static const char* id_scope = "One00000A0A";
```

5. Find the definition for the `main()` function in the same file. Make sure the `hsm_type` variable is set to `SECURE_DEVICE_TYPE_X509` and that all other `hsm_type` lines are commented out. For example:

```
SECURE_DEVICE_TYPE hsm_type;
//hsm_type = SECURE_DEVICE_TYPE TPM;
hsm_type = SECURE_DEVICE_TYPE X509;
//hsm_type = SECURE_DEVICE_TYPE SYMMETRIC_KEY;
```

6. Save your changes.
7. Right-click the `prov_dev_client_sample` project and select **Set as Startup Project**.

Configure the custom HSM stub code

The specifics of interacting with actual secure hardware-based storage vary depending on the device hardware. The certificate chains used by the simulated devices in this tutorial will be hardcoded in the custom HSM stub code. In a real-world scenario, the certificate chain would be stored in the actual HSM hardware to provide better security for sensitive information. Methods similar to the stub methods used in this sample would then be implemented to read the secrets from that hardware-based storage.

While HSM hardware isn't required, it is recommended to protect sensitive information like the certificate's private key. If an actual HSM was being called by the sample, the private key wouldn't be present in the source code. Having the key in the source code exposes the key to anyone that can view the code. This is only done in this tutorial to assist with learning.

To update the custom HSM stub code to simulate the identity of the device with ID `device-01`, perform the following steps:

1. In Solution Explorer for Visual Studio, navigate to **Provision_Samples > custom_hsm_example > Source Files** and open *custom_hsm_example.c*.
2. Update the string value of the `COMMON_NAME` string constant using the common name you used when generating the device certificate.

```
static const char* const COMMON_NAME = "device-01";
```

3. Update the string value of the `CERTIFICATE` constant string using the certificate chain you saved in `/certs/device-01-full-chain.cert.pem` after generating your certificates.

The syntax of certificate text must follow the pattern below with no extra spaces or parsing done by Visual Studio.

```
// <Device/leaf cert>
// <intermediates>
// <root>
static const char* const CERTIFICATE = "-----BEGIN CERTIFICATE-----\n"
"MIIFOjCCAYKgAwIBAgIJAPzMa6s7mj7+MA0GCSqGSIb3DQEBCwUAMCoxKDAmBgNV\n"
...
"MDMwWhcNMjAxMTIyMjEzMjMwWjAqMSgwJgYDVQQDB9BenVyzSBjb1QgSHViIENB\n"
"-----END CERTIFICATE-----\n"
"-----BEGIN CERTIFICATE-----\n"
"MIIFPDCCAYSgAwIBAgIBATANBkqhkiG9w0BAQsFADAqMSgwJgYDVQQDB9BenVy\n"
...
"MTEyMjIxMzAzM1owNDEyMDAGA1UEAwppQXp1cmUgSW9UIEh1YiBJbnRlcmlZG1h\n"
"-----END CERTIFICATE-----\n"
"-----BEGIN CERTIFICATE-----\n"
"MIIFOjCCAYKgAwIBAgIJAPzMa6s7mj7+MA0GCSqGSIb3DQEBCwUAMCoxKDAmBgNV\n"
...
"MDMwWhcNMjAxMTIyMjEzMjMwWjAqMSgwJgYDVQQDB9BenVyzSBjb1QgSHViIENB\n"
"-----END CERTIFICATE-----";
```

Updating this string value manually can be prone to error. To generate the proper syntax, you can copy and paste the following command into your **Git Bash prompt**, and press **ENTER**. This command generates the syntax for the `CERTIFICATE` string constant value and writes it to the output.

```
sed -e 's/^"/;$ !s/$"/\\n"/;$ s/$//' ./certs/device-01-full-chain.cert.pem
```

Copy and paste the output certificate text for the constant value.

4. Update the string value of the `PRIVATE_KEY` constant with the private key for your device certificate.

The syntax of the private key text must follow the pattern below with no extra spaces or parsing done by Visual Studio.

```
static const char* const PRIVATE_KEY = "-----BEGIN RSA PRIVATE KEY-----\n"
"MIJJKwIBAAKCAgEAtjvKQjIhp0EE1PoADL1rfF/W6v4v1AzOSifKSQsaPeebqg8U\\n"
...
"X7fi90Z26QpnkS5QjjPTYI/wn0J9YAwNfKS1NeXTJDFJ+KpjXBcvaLxeBQbQhij\\n"
-----END RSA PRIVATE KEY-----";
```

Updating this string value manually can be prone to error. To generate the proper syntax, you can copy and paste the following command into your **Git Bash prompt**, and press **ENTER**. This command generates the syntax for the `PRIVATE_KEY` string constant value and writes it to the output.

```
sed -e 's/^"/;$ !s/"\\\"\\n"/;$ s//\' ./private/device-01.key.pem
```

Copy and paste the output private key text for the constant value.

5. Save your changes.
6. Right-click the `custom_hsm_example` project and select **Build**.

IMPORTANT

You must build the `custom_hsm_example` project before you build the rest of the solution in the next section.

Run the sample

1. On the Visual Studio menu, select **Debug > Start without debugging** to run the solution. When prompted to rebuild the project, select **Yes** to rebuild the project before running.

The following output is an example of simulated device `device-01` successfully booting up and connecting to the provisioning service. The device was assigned to an IoT hub and registered:

```
Provisioning API Version: 1.8.0

Registering Device

Provisioning Status: PROV_DEVICE_REG_STATUS_CONNECTED
Provisioning Status: PROV_DEVICE_REG_STATUS_ASSIGNING

Registration Information received from service: contoso-hub-2.azure-devices.net, deviceId: device-01
Press enter key to exit:
```

2. Repeat the steps in [Configure the custom HSM stub code](#) for your second device (`device-02`) and run the sample again. Use the following values for that device:

DESCRIPTION	VALUE
Common name	"device-02"
Full certificate chain	Generate the text using <code>./certs/device-02-full-chain.cert.pem</code>
Private key	Generate the text using <code>./private/device-02.key.pem</code>

The following output is an example of simulated device `device-02` successfully booting up, and connecting to the provisioning service. The device was assigned to an IoT hub and registered:

```
Provisioning API Version: 1.8.0

Registering Device

Provisioning Status: PROV_DEVICE_REG_STATUS_CONNECTED
Provisioning Status: PROV_DEVICE_REG_STATUS_ASSIGNING

Registration Information received from service: contoso-hub-2.azure-devices.net, deviceId: device-02
Press enter key to exit:
```

Confirm your device provisioning registration

Examine the registration records of the enrollment group to see the registration details for your devices:

1. In the Azure portal, go to your Device Provisioning Service instance.
2. In the **Settings** menu, select **Manage enrollments**.
3. Select **Enrollment Groups**. The X.509 enrollment group entry that you created previously, *custom-hsm-devices*, should appear in the list.
4. Select the enrollment entry. Then select the **Registration Records** tab to see the devices that have been registered through the enrollment group. The IoT hub that each of your devices was assigned to, their device IDs, and the dates and times they were registered appear in the list.

The screenshot shows the 'custom-hsm-x509-devices' enrollment group details page in the Azure portal. The 'Registration Records' tab is selected and highlighted with a red box. A tooltip message says: 'You can view devices that have provisioned via this enrollment group and remove the registration records for previously provisioned devices'. Below the tabs, there is a search bar labeled 'Search devices in this enrollment group' and a table with two rows of data. The table columns are: Device Id, Assigned IoT Hub, and Registration Date. The data rows are: device-01 (contoso-hub-2.azure-devices.net, 2022-06-21T05:21:09.5701841Z) and device-02 (contoso-hub-2.azure-devices.net, 2022-06-21T05:31:21.6935104Z).

Device Id	Assigned IoT Hub	Registration Date
device-01	contoso-hub-2.azure-devices.net	2022-06-21T05:21:09.5701841Z
device-02	contoso-hub-2.azure-devices.net	2022-06-21T05:31:21.6935104Z

5. You can select one of the devices to see further details for that device.

To verify the devices on your IoT hub:

1. In Azure portal, go to the IoT hub that your device was assigned to.
2. In the **Device management** menu, select **Devices**.
3. If your devices were provisioned successfully, their device IDs, *device-01* and *device-02*, should appear in the list, with **Status** set as *enabled*. If you don't see your devices, select **Refresh**.

The screenshot shows the Azure IoT Hub Devices page for the hub 'contoso-hub-2'. The left sidebar includes links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Events, Pricing and scale, Device management (with 'Devices' selected and highlighted with a red box), and IoT Edge. The main area has a search bar ('Search (Ctrl+/)') and a 'Find devices' button. A table lists two devices: 'device-01' and 'device-02', both of which are Enabled, SelfSigned, and have 0 cloud messages. Buttons for 'Add Device', 'Refresh', and 'Delete' are also present.

Device ID	Status	Last Status Update	Authentication Type	Cloud ...
device-01	Enabled	--	SelfSigned	0
device-02	Enabled	--	SelfSigned	0

Clean up resources

When you're finished testing and exploring this device client sample, use the following steps to delete all resources created by this tutorial.

1. Close the device client sample output window on your machine.
2. From the left-hand menu in the Azure portal, select **All resources** and then select your Device Provisioning Service instance. Open **Manage Enrollments** for your service, and then select the **Enrollment Groups** tab. Select the check box next to the *Group Name* of the device group you created in this tutorial, and select **Delete** at the top of the pane.
3. Select **Certificates** in DPS. For each certificate you uploaded and verified in this tutorial, select the certificate and select **Delete** to remove it.
4. From the left-hand menu in the Azure portal, select **All resources** and then select your IoT hub. Open **IoT devices** for your hub. Select the check box next to the *DEVICE ID* of the device that you registered in this tutorial. Select **Delete** at the top of the pane.

Next steps

In this tutorial, you provisioned an X.509 device using a custom HSM to your IoT hub. To learn how to provision IoT devices to multiple hubs continue to the next tutorial.

[Tutorial: Provision devices across load-balanced IoT hubs](#)

Tutorial: Provision devices across load-balanced IoT hubs

8/22/2022 • 3 minutes to read • [Edit Online](#)

This tutorial shows how to provision devices for multiple, load-balanced IoT hubs using the Device Provisioning Service. In this tutorial, you learn how to:

- Use the Azure portal to provision a second device to a second IoT hub
- Add an enrollment list entry to the second device
- Set the Device Provisioning Service allocation policy to **even distribution**
- Link the new IoT hub to the Device Provisioning Service

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

Use the Azure portal to provision a second device to a second IoT hub

Follow the steps in the quickstarts to link a second IoT hub to your DPS instance and provision a device to that hub:

- [Set up the Device Provisioning Service](#)
- [Provision a simulated symmetric key device](#)

Add an enrollment list entry to the second device

The enrollment list tells the Device Provisioning Service which method of attestation (the method for confirming a device identity) it is using with the device. The next step is to add an enrollment list entry for the second device.

1. In the page for your Device Provisioning Service, click **Manage enrollments**. The **Add enrollment list entry** page appears.
2. At the top of the page, click **Add**.
3. Complete the fields and then click **Save**.

Set the Device Provisioning Service allocation policy

The allocation policy is a Device Provisioning Service setting that determines how devices are assigned to an IoT hub. There are three supported allocation policies:

1. **Lowest latency**: Devices are provisioned to an IoT hub based on the hub with the lowest latency to the device.
2. **Evenly weighted distribution** (default): Linked IoT hubs are equally likely to have devices provisioned to them. This is the default setting. If you are provisioning devices to only one IoT hub, you can keep this setting. If you plan to use on IoT hub, but expect to increase the number of hubs as the number of devices increases, it's important to note that, when assigning to an IoT Hub, the policy doesn't take into account previously registered devices. All linked hubs hold an equal chance of getting a device registration based on the weight of the linked IoT Hub. However, if an IoT hub has reached its device capacity limit, it will no longer receive device registrations. You can, however, adjust the weight of allocation for each linked IoT Hub.

3. **Static configuration via the enrollment list:** Specification of the desired IoT hub in the enrollment list takes priority over the Device Provisioning Service-level allocation policy.

How the allocation policy assigns devices to IoT Hubs

It may be desirable to use only one IoT Hub, until a specific number of devices is reached. In that scenario, it's important to note that, once a new IoT Hub is added, a new device has the potential to be provisioned to any one of the IoT Hubs. If you wish to balance all devices, registered and unregistered, then you'll need to re-provision all devices.

Follow these steps to set the allocation policy:

1. To set the allocation policy, in the Device Provisioning Service page click **Manage allocation policy**.
2. Set the allocation policy to **Evenly weighted distribution**.
3. Click **Save**.

Link the new IoT hub to the Device Provisioning Service

Link the Device Provisioning Service and IoT hub so that the Device Provisioning Service can register devices to that hub.

1. In the **All resources** page, click the Device Provisioning Service you created previously.
2. In the Device Provisioning Service page, click **Linked IoT hubs**.
3. Click **Add**.
4. In the **Add link to IoT hub** page, use the radio buttons to specify whether the linked IoT hub is located in the current subscription, or in a different subscription. Then, choose the name of the IoT hub from the **IoT hub** box.
5. Click **Save**.

In this tutorial, you learned how to:

- Use the Azure portal to provision a second device to a second IoT hub
- Add an enrollment list entry to the second device
- Set the Device Provisioning Service allocation policy to **even distribution**
- Link the new IoT hub to the Device Provisioning Service

Next steps

Tutorial: Provision for multitenancy

8/22/2022 • 12 minutes to read • [Edit Online](#)

This tutorial shows how to securely provision multiple simulated symmetric key devices to a group of IoT Hubs using an [allocation policy](#). Allocation policies that are defined by the provisioning service support a variety of allocation scenarios. Two common scenarios are:

- **Geolocation / GeoLatency:** As a device moves between locations, network latency is improved by having the device provisioned to the IoT hub that's closest to each location. In this scenario, a group of IoT hubs, which span across regions, are selected for enrollments. The **Lowest latency** allocation policy is selected for these enrollments. This policy causes the Device Provisioning Service to evaluate device latency and determine the closest IoT hub out of the group of IoT hubs.
- **Multi-tenancy:** Devices used within an IoT solution may need to be assigned to a specific IoT hub or group of IoT hubs. The solution may require all devices for a particular tenant to communicate with a specific group of IoT hubs. In some cases, a tenant may own IoT hubs and require devices to be assigned to their IoT hubs.

It's common to combine these two scenarios. For example, a multitenant IoT solution commonly assigns tenant devices using a group of IoT hubs that are scattered across different regions. These tenant devices can be assigned to the IoT hub in the group that has the lowest latency based on geographic location.

This tutorial uses a simulated device sample from the [Azure IoT C SDK](#) to demonstrate how to provision devices in a multitenant scenario across regions. You will perform the following steps in this tutorial:

- Use the Azure CLI to create two regional IoT hubs (**West US 2** and **East US**)
- Create a multitenant enrollment
- Use the Azure CLI to create two regional Linux VMs to act as devices in the same regions (**West US 2** and **East US**)
- Set up the development environment for the Azure IoT C SDK on both Linux VMs
- Simulate the devices to see that they are provisioned for the same tenant in the closest region.

IMPORTANT

Some regions may, from time to time, enforce restrictions on the creation of Virtual Machines. At the time of writing this guide, the *westus2* and *eastus* regions permitted the creation of VMs. If you're unable to create in either one of those regions, you can try a different region. To learn more about choosing Azure geographical regions when creating VMs, see [Regions for virtual machines in Azure](#)

Prerequisites

- If you don't have an Azure subscription, create a [free account](#) before you begin.
- Complete the steps in [Set up IoT Hub Device Provisioning Service with the Azure portal](#).
- Use the Bash environment in [Azure Cloud Shell](#). For more information, see [Azure Cloud Shell Quickstart - Bash](#).
- If you prefer to run CLI reference commands locally, [install](#) the Azure CLI. If you're running on Windows



[Launch Cloud Shell](#)

or macOS, consider running Azure CLI in a Docker container. For more information, see [How to run the Azure CLI in a Docker container](#).

- If you're using a local installation, sign in to the Azure CLI by using the `az login` command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see [Sign in with the Azure CLI](#).
- When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see [Use extensions with the Azure CLI](#).
- Run `az version` to find the version and dependent libraries that are installed. To upgrade to the latest version, run `az upgrade`.

Create two regional IoT hubs

In this section, you'll create an Azure resource group, and two new regional IoT hub resources for a tenant. One IoT hub will be for the **West US 2** region and the other will be for the **East US** region.

IMPORTANT

It's recommended that you use the same resource group for all resources created in this tutorial. This will make clean up easier after you are finished.

1. In the Azure Cloud Shell, create a resource group with the following `az group create` command:

```
az group create --name contoso-us-resource-group --location eastus
```

2. Create an IoT hub in the *eastus* location, and add it to the resource group you created with the following `az iot hub create` command(replace `{unique-hub-name}` with your own unique name):

```
az iot hub create --name {unique-hub-name} --resource-group contoso-us-resource-group --location eastus --sku S1
```

This command may take a few minutes to complete.

3. Now, create an IoT hub in the *westus2* location, and add it to the resource group you created with the following `az iot hub create` command(replace `{unique-hub-name}` with your own unique name):

```
az iot hub create --name {unique-hub-name} --resource-group contoso-us-resource-group --location westus2 --sku S1
```

This command may take a few minutes to complete.

Create the multitenant enrollment

In this section, you'll create a new enrollment group for the tenant devices.

For simplicity, this tutorial uses [Symmetric key attestation](#) with the enrollment. For a more secure solution, consider using [X.509 certificate attestation](#) with a chain of trust.

1. In the Azure portal, select your Device Provisioning Service.
2. In the **Settings** menu, select **Manage enrollments**.
3. Select **+ Add enrollment group**.

4. On the Add Enrollment Group page, enter the following information:

Group name: Enter *contoso-us-devices*. The enrollment group name is a case-insensitive string (up to 128 characters long) of alphanumeric characters plus the special characters: `'-'`, `'. '`, `'_'`, `': '`. The last character must be alphanumeric or dash (`'-'`).

Attestation Type: Select *Symmetric Key*.

Auto Generate Keys: This checkbox should already be checked.

Select how you want to assign devices to hubs: Select *Lowest latency*.

5. Select Link a new IoT Hub

The screenshot shows the 'Add Enrollment Group' page with the following fields filled out:

- Group name ***: contoso-us-devices
- Attestation Type**: Symmetric Key
- Auto-generate keys**: checked
- Primary Key**: Enter your primary key
- Secondary Key**: Enter your secondary key
- IoT Edge device**: False
- Select how you want to assign devices to hubs**: Lowest latency
- Select the IoT hubs this group can be assigned to:**: Contoso-IotHub-2.azure-devices.net
- Link a new IoT hub**: A blue button.
- Select how you want device data to be handled on re-provisioning ***: Re-provision and migrate data
- Initial Device Twin State**: A JSON editor showing: `{"tags": {}}`

6. On the Add link to IoT hub page, enter the following information:

Subscription: If you have multiple subscriptions, choose the subscription where you created the regional IoT hubs.

IoT hub: Select the IoT hub that you created for the *eastus* location.

Access Policy: Select *iothubowner*.

Add link to IoT hub

X

Learn more about linking IoT hubs.

Subscription * ⓘ

IoT Documentation - Aquent vendors

IoT hub *

contoso-east-hub2

Access Policy * ⓘ

iothubowner

Hostname contoso-east-hub2.azure-devices.net

Status Active

Pricing Tier S1

Location East US

Save

7. Select **Save**.
8. Repeat Steps 5 through 7 for the second IoT hub that you created for the *westus* location.
9. Select the two IoT Hubs you created in the **Select the IoT hubs this group can be assigned to** drop down.

Add Enrollment Group

 Save

Group name *

contoso-us-devices 

Attestation Type 

Certificate Symmetric Key

Auto-generate keys 



Primary Key 

Enter your primary key

Secondary Key 

Enter your secondary key

IoT Edge device 

Select all

Contoso-iotHub-2.azure-devices.net

contoso-east-hub2.azure-devices.net

contoso-west-hub2.azure-devices.net

2 selected 

Link a new IoT hub

Select how you want device data to be handled on re-provisioning * 

Re-provision and migrate data 

Device Twin is only supported for standard tier IoT hubs. Learn more about standard tier.

Initial Device Twin State

{ 

10. Select Save

11. Select *contoso-us-devices* in the enrollment groups list.

12. Copy the *Primary Key*. This key will be used later to generate unique device keys for both simulated devices.

Attestation Type
Symmetric Key

Primary Key

Secondary Key

IoT Edge device ⓘ
True False

Select how you want to assign devices to hubs
Lowest latency

Select the IoT hubs this group can be assigned to: ⓘ
2 selected

Link a new IoT hub

Create regional Linux VMs

In this section, you'll create two regional Linux virtual machines (VMs). These VMs will run a device simulation sample from each region to demonstrate device provisioning for tenant devices from both regions.

To make clean-up easier, these VMs will be added to the same resource group that contains the IoT hubs that were created, *contoso-us-resource-group*. However, the VMs will run in separate regions (**West US 2** and **East US**).

1. In the Azure Cloud Shell, run the following command to create an **East US** region VM after making the following parameter changes in the command:

--name: Enter a unique name for your **East US** regional device VM.

--admin-username: Use your own admin user name.

--admin-password: Use your own admin password.

```
az vm create \
--resource-group contoso-us-resource-group \
--name ContosoSimDeviceEast \
--location eastus \
--image Canonical:UbuntuServer:18.04-LTS:18.04.201809110 \
--admin-username contosoadmin \
--admin-password myContosoPassword2018 \
--authentication-type password
--public-ip-sku Standard
```

This command will take a few minutes to complete.

2. Once the command has completed, copy the **publicIpAddress** value for your East US region VM.
3. In the Azure Cloud Shell, run the command to create a **West US 2** region VM after making the following parameter changes in the command:

--name: Enter a unique name for your **West US 2** regional device VM.

--admin-username: Use your own admin user name.

--admin-password: Use your own admin password.

```
az vm create \
--resource-group contoso-us-resource-group \
--name ContosoSimDeviceWest2 \
--location westus2 \
--image Canonical:UbuntuServer:18.04-LTS:18.04.201809110 \
--admin-username contosoadmin \
--admin-password myContosoPassword2018 \
--authentication-type password
--public-ip-sku Standard
```

This command will take a few minutes to complete.

4. Once the command has completed, copy the **publicIpAddress** value for your West US 2 region VM.
5. Open two command-line shells.
6. Connect to one of the regional VMs in each shell using SSH.

Pass your admin username and the public IP address that you copied as parameters to SSH. Enter the admin password when prompted.

```
ssh contosoadmin@1.2.3.4
contosoadmin@ContosoSimDeviceEast:~$
```

```
ssh contosoadmin@5.6.7.8
contosoadmin@ContosoSimDeviceWest:~$
```

Prepare the Azure IoT C SDK development environment

In this section, you'll clone the Azure IoT C SDK on each VM. The SDK contains a sample that simulates a tenant's device provisioning from each region.

For each VM:

1. Install **CMake**, **g++**, **gcc**, and **Git** using the following commands:

```
sudo apt-get update
sudo apt-get install cmake build-essential libssl-dev libcurl4-openssl-dev uuid-dev git-all
```

2. Find and copy the tag name for the [latest release](#) of the SDK.
3. Clone the [Azure IoT C SDK](#) on both VMs. 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 new **cmake** folder inside the repository and change to that folder.

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

5. Run the following command, which builds a version of the SDK specific to your development client platform:

```
cmake -Dhsm_type_symm_key:BOOL=ON -Duse_prov_client:BOOL=ON ..
```

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

```
-- IoT Client SDK Version = 1.7.0
-- Provisioning SDK Version = 1.7.0
-- Looking for include file stdint.h
-- Looking for include file stdint.h - found
-- Looking for include file stdbool.h
-- Looking for include file stdbool.h - found
-- target architecture: x86_64
-- Performing Test CXX_FLAG_CXX11
-- Performing Test CXX_FLAG_CXX11 - Success
-- Found OpenSSL: /usr/lib/x86_64-linux-gnu/libcrypto.so (found version "1.1.1")
-- Found CURL: /usr/lib/x86_64-linux-gnu/libcurl.so (found version "7.58.0")
-- Found CURL: /usr/lib/x86_64-linux-gnu/libcurl.so
-- target architecture: x86_64
-- iothub architecture: x86_64
-- Configuring done
-- Generating done
-- Build files have been written to: /home/contosoadmin/azure-iot-sdk-c/azure-iot-sdk-c
```

Derive unique device keys

When using symmetric key attestation with group enrollments, you don't use the enrollment group keys directly. Instead, you derive a unique key from the enrollment group key for each device. For more information, see [Group Enrollments with symmetric keys](#).

In this part of the tutorial, you'll generate a device key from the group master key to compute an [HMAC-SHA256](#) of the unique registration ID for the device. The result will then be converted into Base64 format.

IMPORTANT

Don't include your group master key in your device code.

For **both** *eastus* and *westus 2* devices:

1. Generate your unique key using `openssl`. You'll use the following Bash shell script (replace `{primary-key}` with the enrollment group's **Primary Key** that you copied earlier and replace `{contoso-simdevice}` with your own unique registration ID for each device. The registration ID is a case-insensitive string (up to 128 characters long) of alphanumeric characters plus the special characters: `'-'`,

'.' , '_' , ':'. The last character must be alphanumeric or dash ('-').

```
KEY={primary-key}
REG_ID={contoso-simdevice}

keybytes=$(echo $KEY | base64 --decode | xxd -p -u -c 1000)
echo -n $REG_ID | openssl sha256 -mac HMAC -macopt hexkey:$keybytes -binary | base64
```

2. The script will output something like the following key:

```
p3w2DQr9WqEGBLUS1Fi1jPQ7UWQL4siAGy75HFTFbf8=
```

3. Now each tenant device has their own derived device key and unique registration ID to perform symmetric key attestation with the enrollment group during the provisioning process.

Simulate the devices from each region

In this section, you'll update a provisioning sample in the Azure IoT C SDK for both of the regional VMs.

The sample code simulates a device boot sequence that sends the provisioning request to your Device Provisioning Service instance. The boot sequence causes the device to be recognized and assigned to the IoT hub that is closest based on latency.

1. In the Azure portal, select the **Overview** tab for your Device Provisioning Service and note down the *ID Scope* value.

The screenshot shows the Azure portal interface for a Device Provisioning Service named 'Contoso-DPS'. The left sidebar contains navigation links: Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings (Quick Start, Shared access policies, Linked IoT hubs, Certificates, Manage enrollments, Manage allocation policy, Networking, Properties, Locks, Monitoring, Alerts), and a bottom row of Monitoring, Metrics, and Log Analytics. The main content area is titled 'Essentials' and displays service endpoint information: Contoso-DPS.azure-devices-provisioning.net, Global device endpoint global.azure-devices-provisioning.net, and the highlighted 'ID Scope' value 'One00364DC0'. Below this, it shows subscription details (Internal use - Aquent vendors, Subscription ID, Tags), and a 'Quick Links' section with three items: Azure IoT Hub Device Provisioning Service Documentation, Learn more about IoT Hub Device Provisioning Service, and Device Provisioning concepts.

2. On both VMS, open `~/azure-iot-sdk-c/provisioning_client/samples/prov_dev_client_sample/prov_dev_client_sample.c` for editing.

```
vi ~/azure-iot-sdk-c/provisioning_client/samples/prov_dev_client_sample/prov_dev_client_sample.c
```

3. On both VMs, find the `id_scope` constant, and replace the value with your ID Scope value that you copied earlier.

```
static const char* id_scope = "0ne00002193";
```

4. On both VMs, find the definition for the `main()` function in the same file. Make sure the `hsm_type` variable is set to `SECURE_DEVICE_TYPE_SYMMETRIC_KEY` as shown below to match the enrollment group attestation method.

Save your changes to the files on both VMs.

```
SECURE_DEVICE_TYPE hsm_type;
//hsm_type = SECURE_DEVICE_TYPE TPM;
//hsm_type = SECURE_DEVICE_TYPE_X509;
hsm_type = SECURE_DEVICE_TYPE_SYMMETRIC_KEY;
```

5. On both VMs, find the call to `prov_dev_set_symmetric_key_info()` in `prov_dev_client_sample.c` which is commented out.

```
// Set the symmetric key if using they auth type
//prov_dev_set_symmetric_key_info("<symm_registration_id>", "<symmetric_Key>");
```

Uncomment the function calls, and replace the placeholder values (including the angle brackets) with the unique registration IDs and derived device keys for each device that you derived in the previous section. The keys shown below are for example purposes only. Use the keys you generated earlier.

East US:

```
// Set the symmetric key if using they auth type
prov_dev_set_symmetric_key_info("contoso-simdevice-east",
"p3w2DQr9WqEGBLUS1Fi1jPQ7UWQL4siAGy75HFTFbf8=");
```

West US:

```
// Set the symmetric key if using they auth type
prov_dev_set_symmetric_key_info("contoso-simdevice-west",
"J5n4NY2GiBYy7Mp4lDDa5CbEe6zDU/c62rhjCuFWxnc=");
```

6. On both VMs, save the file.
7. On both VMs, navigate to the sample folder shown below, and build the sample.

```
cd ~/azure-iot-sdk-c/cmake/provisioning_client/samples/prov_dev_client_sample/
cmake --build . --target prov_dev_client_sample --config Debug
```

8. Once the build succeeds, run `prov_dev_client_sample.exe` on both VMs to simulate a tenant device from each region. Notice that each device is allocated to the tenant IoT hub closest to the simulated device's regions.

Run the simulation:

```
~/azure-iot-sdk-c/cmake/provisioning_client/samples/prov_dev_client_sample/prov_dev_client_sample
```

Example output from the East US VM:

```
contosoadmin@ContosoSimDeviceEast:~/azure-iot-sdk-c/cmake/provisioning_client/samples/prov_dev_client_sample$ ./prov_dev_client_sample
Provisioning API Version: 1.2.9

Registering Device

Provisioning Status: PROV_DEVICE_REG_STATUS_CONNECTED
Provisioning Status: PROV_DEVICE_REG_STATUS_ASSIGNING
Provisioning Status: PROV_DEVICE_REG_STATUS_ASSIGNING

Registration Information received from service: contoso-east-hub.azure-devices.net, deviceId:
contoso-simdevice-east
Press enter key to exit:
```

Example output from the West US VM:

```
contosoadmin@ContosoSimDeviceWest:~/azure-iot-sdk-c/cmake/provisioning_client/samples/prov_dev_client_sample$ ./prov_dev_client_sample
Provisioning API Version: 1.2.9

Registering Device

Provisioning Status: PROV_DEVICE_REG_STATUS_CONNECTED
Provisioning Status: PROV_DEVICE_REG_STATUS_ASSIGNING
Provisioning Status: PROV_DEVICE_REG_STATUS_ASSIGNING

Registration Information received from service: contoso-west-hub.azure-devices.net, deviceId:
contoso-simdevice-west
Press enter key to exit:
```

Clean up resources

If you plan to continue working with resources created in this tutorial, you can leave them. Otherwise, use the following steps to delete all resources created by this tutorial to avoid unnecessary charges.

The steps here assume that you created all resources in this tutorial as instructed in the same resource group named **contoso-us-resource-group**.

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 the resource group by name:

1. Sign in to the [Azure portal](#).
2. Select **Resource groups**.
3. In the **Filter by name...** textbox, type the name of the resource group containing your resources, **contoso-us-resource-group**.
4. To the right of your resource group in the result list, click ... then **Delete resource group**.
5. You'll 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

- To learn more about reprovisioning, see:

[IoT Hub Device reprovisioning concepts](#)

- To learn more about deprovisioning, see

[How to deprovision devices that were previously auto-provisioned](#)

IoT Hub Device Provisioning Service (DPS) terminology

8/22/2022 • 7 minutes to read • [Edit Online](#)

IoT Hub Device Provisioning Service is a helper service for IoT Hub that you use to configure zero-touch device provisioning to a specified IoT hub. With the Device Provisioning Service, you can [provision](#) millions of devices in a secure and scalable manner.

Device provisioning is a two part process. The first part is establishing the initial connection between the device and the IoT solution by *registering* the device. The second part is applying the proper *configuration* to the device based on the specific requirements of the solution. Once both steps have been completed, the device has been fully *provisioned*. Device Provisioning Service automates both steps to provide a seamless provisioning experience for the device.

This article gives an overview of the provisioning concepts most applicable to managing the *service*. This article is most relevant to personas involved in the [cloud setup step](#) of getting a device ready for deployment.

Service operations endpoint

The service operations endpoint is the endpoint for managing the service settings and maintaining the enrollment list. This endpoint is only used by the service administrator; it is not used by devices.

Device provisioning endpoint

The device provisioning endpoint is the single endpoint all devices use for auto-provisioning. The URL is the same for all provisioning service instances, to eliminate the need to reflash devices with new connection information in supply chain scenarios. The ID scope ensures tenant isolation.

Linked IoT hubs

The Device Provisioning Service can only provision devices to IoT hubs that have been linked to it. Linking an IoT hub to an instance of the Device Provisioning Service gives the service read/write permissions to the IoT hub's device registry; with the link, a Device Provisioning Service can register a device ID and set the initial configuration in the device twin. Linked IoT hubs may be in any Azure region. You may link hubs in other subscriptions to your provisioning service.

Allocation policy

The service-level setting that determines how Device Provisioning Service assigns devices to an IoT hub. There are four supported allocation policies:

- **Evenly weighted distribution:** linked IoT hubs are equally likely to have devices provisioned to them. The default setting. If you are provisioning devices to only one IoT hub, you can keep this setting.
- **Lowest latency:** devices are provisioned to an IoT hub with the lowest latency to the device. If multiple linked IoT hubs would provide the same lowest latency, the provisioning service hashes devices across those hubs
- **Static configuration via the enrollment list:** specification of the desired IoT hub in the enrollment list takes priority over the service-level allocation policy.

- **Custom (Use Azure Function):** A [custom allocation policy](#) gives you more control over how devices are assigned to an IoT hub. This is accomplished by using custom code in an Azure Function to assign devices to an IoT hub. The device provisioning service calls your Azure Function code providing all relevant information about the device and the enrollment to your code. Your function code is executed and returns the IoT hub information used to provisioning the device.

Enrollment

An enrollment is the record of devices or groups of devices that may register through auto-provisioning. The enrollment record contains information about the device or group of devices, including:

- the [attestation mechanism](#) used by the device
- the optional initial desired configuration
- desired IoT hub
- the desired device ID

There are two types of enrollments supported by Device Provisioning Service:

Enrollment group

An enrollment group is a group of devices that share a specific attestation mechanism. Enrollment groups support X.509 certificate or symmetric key attestation. Devices in an X.509 enrollment group present X.509 certificates that have been signed by the same root or intermediate Certificate Authority (CA). The subject common name (CN) of each device's end-entity (leaf) certificate becomes the registration ID for that device. Devices in a symmetric key enrollment group present SAS tokens derived from the group symmetric key.

The name of the enrollment group as well as the registration IDs presented by devices must be case-insensitive strings of alphanumeric characters plus the special characters: `'-'`, `'.'`, `'_'`, `'::'`. The last character must be alphanumeric or dash (`'-'`). The enrollment group name can be up to 128 characters long. In symmetric key enrollment groups, the registration IDs presented by devices can be up to 128 characters long. However, in X.509 enrollment groups, because the maximum length of the subject common name in an X.509 certificate is 64 characters, the registration IDs are limited to 64 characters.

For devices in an enrollment group, the registration ID is also used as the device ID that is registered to IoT Hub.

TIP

We recommend using an enrollment group for a large number of devices that share a desired initial configuration, or for devices all going to the same tenant.

Individual enrollment

An individual enrollment is an entry for a single device that may register. Individual enrollments may use either X.509 leaf certificates or SAS tokens (from a physical or virtual TPM) as the attestation mechanisms. The registration ID in an individual enrollment is a case-insensitive string of alphanumeric characters plus the special characters: `'-'`, `'.'`, `'_'`, `'::'`. The last character must be alphanumeric or dash (`'-'`). DPS supports registration IDs up to 128 characters long.

For X.509 individual enrollments, the subject common name (CN) of the certificate becomes the registration ID, so the common name must adhere to the registration ID string format. The subject common name has a maximum length of 64 characters, so the registration ID is limited to 64 characters for X.509 enrollments.

Individual enrollments may have the desired IoT hub device ID specified in the enrollment entry. If it's not specified, the registration ID becomes the device ID that's registered to IoT Hub.

TIP

We recommend using individual enrollments for devices that require unique initial configurations, or for devices that can only authenticate using SAS tokens via TPM attestation.

Attestation mechanism

An attestation mechanism is the method used for confirming a device's identity. The attestation mechanism is configured on an enrollment entry and tells the provisioning service which method to use when verifying the identity of a device during registration.

NOTE

IoT Hub uses "authentication scheme" for a similar concept in that service.

The Device Provisioning Service supports the following forms of attestation:

- **X.509 certificates** based on the standard X.509 certificate authentication flow. For more information, see [X.509 attestation](#).
- **Trusted Platform Module (TPM)** based on a nonce challenge, using the TPM standard for keys to present a signed Shared Access Signature (SAS) token. This does not require a physical TPM on the device, but the service expects to attest using the endorsement key per the [TPM spec](#). For more information, see [TPM attestation](#).
- **Symmetric Key** based on shared access signature (SAS) [SAS tokens](#), which include a hashed signature and an embedded expiration. For more information, see [Symmetric key attestation](#).

Hardware security module

The hardware security module, or HSM, is used for secure, hardware-based storage of device secrets, and is the most secure form of secret storage. Both X.509 certificates and SAS tokens can be stored in the HSM. HSMs can be used with both attestation mechanisms the provisioning service supports.

TIP

We strongly recommend using an HSM with devices to securely store secrets on your devices.

Device secrets may also be stored in software (memory), but it is a less secure form of storage than an HSM.

ID scope

The ID scope is assigned to a Device Provisioning Service when it is created by the user and is used to uniquely identify the specific provisioning service the device will register through. The ID scope is generated by the service and is immutable, which guarantees uniqueness.

NOTE

Uniqueness is important for long-running deployment operations and merger and acquisition scenarios.

Registration

A registration is the record of a device successfully registering/provisioning to an IoT Hub via the Device

Provisioning Service. Registration records are created automatically; they can be deleted, but they cannot be updated.

Registration ID

The registration ID is used to uniquely identify a device registration with the Device Provisioning Service. The registration ID must be unique in the provisioning service [ID scope](#). Each device must have a registration ID. The registration ID is a case-insensitive string of alphanumeric characters plus the special characters: `'-'`, `'.'`, `'_'`, `'::'`. The last character must be alphanumeric or dash (`'-'`). DPS supports registration IDs up to 128 characters long.

- In the case of TPM, the registration ID is provided by the TPM itself.
- In the case of X.509-based attestation, the registration ID is set to the subject common name (CN) of the device certificate. For this reason, the common name must adhere to the registration ID string format. However, the registration ID is limited to 64 characters because that's the maximum length of the subject common name in an X.509 certificate.

Device ID

The device ID is the ID as it appears in IoT Hub. The desired device ID may be set in the enrollment entry, but it is not required to be set. Setting the desired device ID is only supported in individual enrollments. If no desired device ID is specified in the enrollment list, the registration ID is used as the device ID when registering the device. Learn more about [device IDs in IoT Hub](#).

Operations

Operations are the billing unit of the Device Provisioning Service. One operation is the successful completion of one instruction to the service. Operations include device registrations and re-registrations; operations also include service-side changes such as adding enrollment list entries, and updating enrollment list entries.

Symmetric key attestation

8/22/2022 • 6 minutes to read • [Edit Online](#)

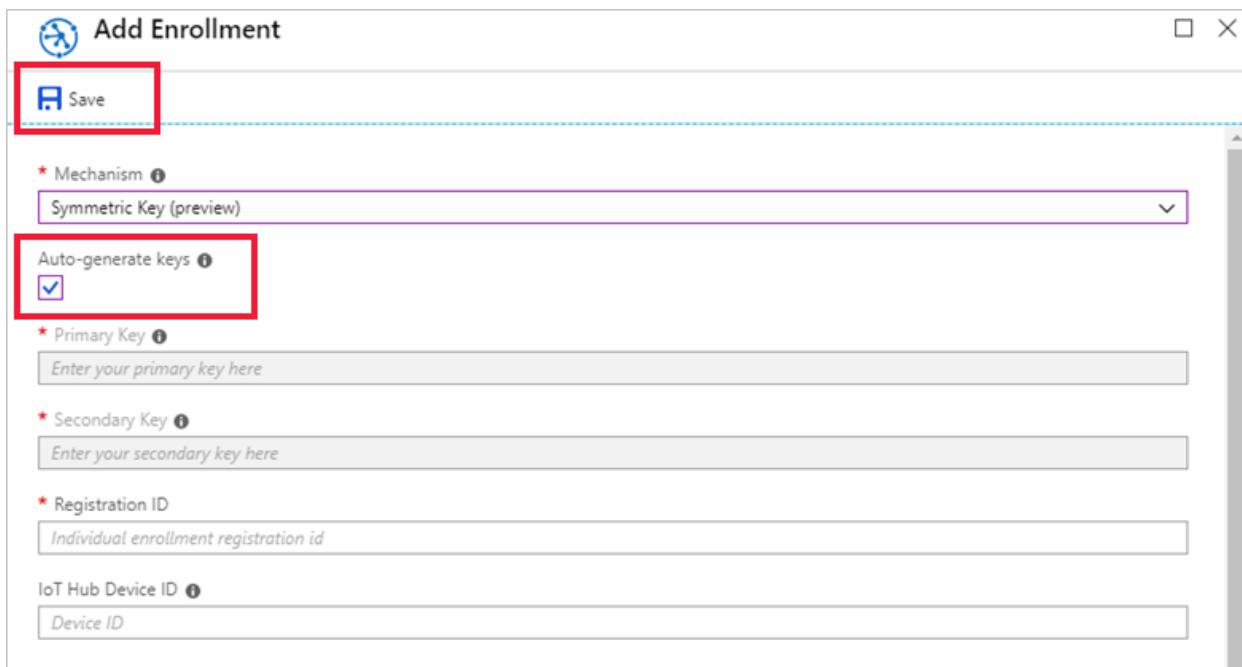
This article describes the identity attestation process when using symmetric keys with the Device Provisioning Service.

Symmetric key attestation is a simple approach to authenticating a device with a Device Provisioning Service instance. This attestation method represents a "Hello world" experience for developers who are new to device provisioning, or do not have strict security requirements. Device attestation using a [TPM](#) or an [X.509 certificate](#) is more secure, and should be used for more stringent security requirements.

Symmetric key enrollments also provide a great way for legacy devices, with limited security functionality, to bootstrap to the cloud via Azure IoT. For more information on symmetric key attestation with legacy devices, see [How to use symmetric keys with legacy devices](#).

Symmetric key creation

By default, the Device Provisioning Service creates new symmetric keys with a default length of 64 bytes when new enrollments are saved with the **Auto-generate keys** option enabled.



You can also provide your own symmetric keys for enrollments by disabling this option. When specifying your own symmetric keys, your keys must have a key length between 16 bytes and 64 bytes. Also, symmetric keys must be provided in valid Base64 format.

Detailed attestation process

Symmetric key attestation with the Device Provisioning Service is performed using the same [security tokens](#) supported by IoT hubs to identify devices. These security tokens are [Shared Access Signature \(SAS\) tokens](#).

SAS tokens have a hashed *signature* that is created using the symmetric key. The signature is recreated by the Device Provisioning Service to verify whether a security token presented during attestation is authentic or not.

SAS tokens have the following form:

```
SharedAccessSignature sig={signature}&se={expiry}&skn={policyName}&sr={URL-encoded-resourceURI}
```

Here are the components of each token:

VALUE	DESCRIPTION
{signature}	An HMAC-SHA256 signature string. For individual enrollments, this signature is produced by using the symmetric key (primary or secondary) to perform the hash. For enrollment groups, a key derived from the enrollment group key is used to perform the hash. The hash is performed on a message of the form: <code>URL-encoded-resourceURI + "\n" + expiry</code> . Important: The key must be decoded from base64 before being used to perform the HMAC-SHA256 computation. Also, the signature result must be URL-encoded.
{resourceURI}	URI of the registration endpoint that can be accessed with this token, starting with scope ID for the Device Provisioning Service instance. For example, <code>{Scope ID}/registrations/{Registration ID}</code>
{expiry}	UTF8 strings for number of seconds since the epoch 00:00:00 UTC on 1 January 1970.
{URL-encoded-resourceURI}	Lower case URL-encoding of the lower case resource URI
{policyName}	The name of the shared access policy to which this token refers. The policy name used when provisioning with symmetric key attestation is registration .

When a device is attesting with an individual enrollment, the device uses the symmetric key defined in the individual enrollment entry to create the hashed signature for the SAS token.

For code examples that create a SAS token, see [SAS tokens](#).

Creating security tokens for symmetric key attestation is supported by the Azure IoT C SDK. For an example using the Azure IoT C SDK to attest with an individual enrollment, see [Provision a simulated symmetric key device](#).

Group Enrollments

Unlike an individual enrollment, the symmetric key of an enrollment group isn't used directly by devices when they provision. Instead, devices that provision through an enrollment group do so using a derived device key. The derived device key is a hash of the device's registration ID and is computed using the symmetric key of the enrollment group. The device can then use its derived device key to sign the SAS token it uses to register with DPS. Because the device sends its registration ID when it registers, DPS can use the enrollment group symmetric key to regenerate the device's derived device key and verify the signature on the SAS token. For details, continue reading.

First, a unique registration ID is defined for each device authenticating through an enrollment group. The registration ID is a case-insensitive string (up to 128 characters long) of alphanumeric characters plus the special characters: `'-'`, `'.'`, `'_'`, `':'`. The last character must be alphanumeric or dash (`'-'`). The registration ID should be something unique that identifies the device. For example, a legacy device may not support many security features. The legacy device may only have a MAC address or serial number available to uniquely identify that device. In that case, a registration ID can be composed of the MAC address and serial number similar to the following:

```
sn-007-888-abc-mac-a1-b2-c3-d4-e5-f6
```

This exact example is used in the [How to provision legacy devices using symmetric keys](#) article.

Once a registration ID has been defined for the device, the symmetric key for the enrollment group is used to compute an [HMAC-SHA256](#) hash of the registration ID to produce a derived device key. Some example approaches to computing the derived device key are given in the tabs below.

- [Azure CLI](#)
- [Windows](#)
- [Linux](#)
- [CSharp](#)

The IoT extension for the Azure CLI provides the `compute-device-key` command for generating derived device keys. This command can be used from Windows-based or Linux systems, in PowerShell or a Bash shell.

Replace the value of `--key` argument with the **Primary Key** from your enrollment group.

Replace the value of `--registration-id` argument with your registration ID.

```
az iot dps compute-device-key --key  
8isrFI1sGsIlvvFSSFRiMfcNzv21fjbE/+ah/lSh31F8e2YG1Te7w1KpZhJFFXJrqYKi9yegxkqIChbqOS9Egw== --registration-id  
sn-007-888-abc-mac-a1-b2-c3-d4-e5-f6
```

Example result:

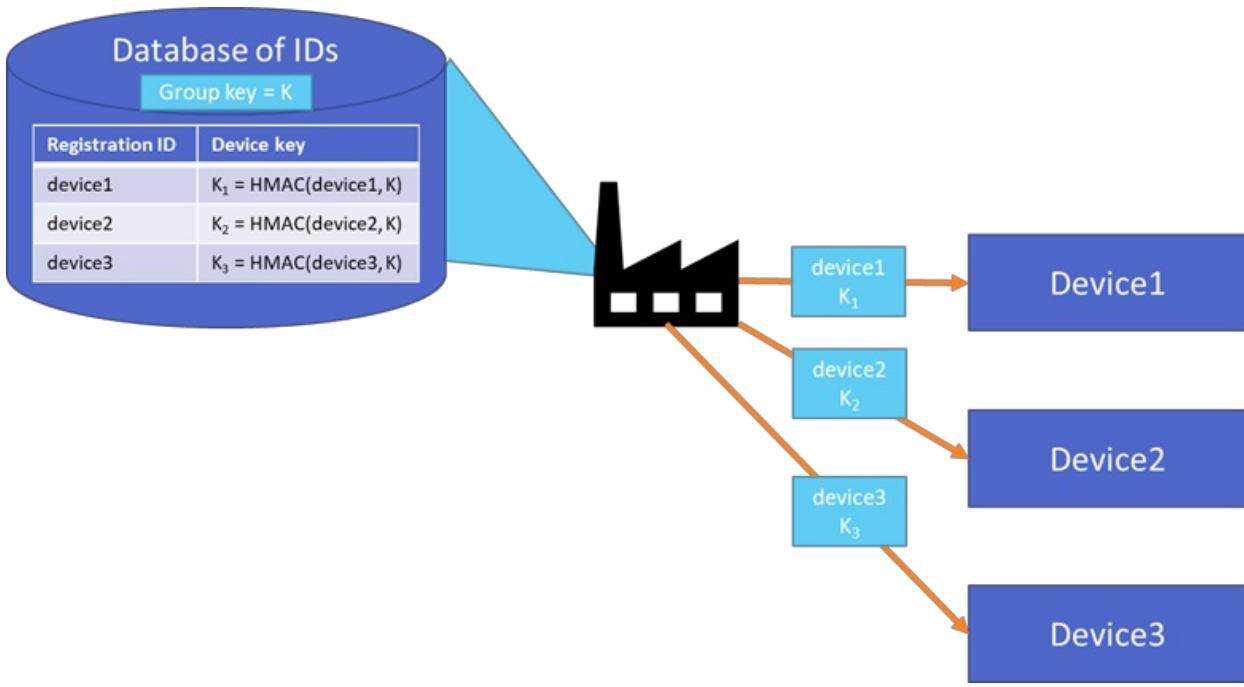
```
"Jsm0lyGpjavYVP2g3FnmmG9dI/9qu24wNoykUmermc="
```

The resulting device key is then used to generate a SAS token to be used for attestation. Each device in an enrollment group is required to attest using a security token generated from a unique derived key. The enrollment group symmetric key cannot be used directly for attestation.

Installation of the derived device key

Ideally the device keys are derived and installed in the factory. This method guarantees the group key is never included in any software deployed to the device. When the device is assigned a MAC address or serial number, the key can be derived and injected into the device however the manufacturer chooses to store it.

Consider the following diagram that shows a table of device keys generated in a factory by hashing each device registration ID with the group enrollment key (K).



The identity of each device is represented by the registration ID and derived device key that is installed at the factory. The device key is never copied to another location and the group key is never stored on a device.

If the device keys are not installed in the factory, a [hardware security module HSM](#) should be used to securely store the device identity.

Next steps

Now that you have an understanding of Symmetric Key attestation, check out the following articles to learn more:

- [Quickstart: Provision a simulated symmetric key device](#)
- [Learn about the concepts of provisioning](#)
- [Get started using auto-provisioning](#)

X.509 certificate attestation

8/22/2022 • 6 minutes to read • [Edit Online](#)

This article gives an overview of the Device Provisioning Service (DPS) concepts involved when provisioning devices using X.509 certificate attestation. This article is relevant to all personas involved in getting a device ready for deployment.

X.509 certificates can be stored in a hardware security module HSM.

TIP

We strongly recommend using an HSM with devices to securely store secrets, like the X.509 certificate, on your devices in production.

X.509 certificates

Using X.509 certificates as an attestation mechanism is an excellent way to scale production and simplify device provisioning. X.509 certificates are typically arranged in a certificate chain of trust in which each certificate in the chain is signed by the private key of the next higher certificate, and so on, terminating in a self-signed root certificate. This arrangement establishes a delegated chain of trust from the root certificate generated by a trusted root certificate authority (CA) down through each intermediate CA to the end-entity "leaf" certificate installed on a device. To learn more, see [Device Authentication using X.509 CA Certificates](#).

Often the certificate chain represents some logical or physical hierarchy associated with devices. For example, a manufacturer may:

- issue a self-signed root CA certificate
- use the root certificate to generate a unique intermediate CA certificate for each factory
- use each factory's certificate to generate a unique intermediate CA certificate for each production line in the plant
- and finally use the production line certificate, to generate a unique device (end-entity) certificate for each device manufactured on the line.

To learn more, see [Conceptual understanding of X.509 CA certificates in the IoT industry](#).

Root certificate

A root certificate is a self-signed X.509 certificate representing a certificate authority (CA). It is the terminus, or trust anchor, of the certificate chain. Root certificates can be self-issued by an organization or purchased from a root certificate authority. To learn more, see [Get X.509 CA certificates](#). The root certificate can also be referred to as a root CA certificate.

Intermediate certificate

An intermediate certificate is an X.509 certificate, which has been signed by the root certificate (or by another intermediate certificate with the root certificate in its chain). The last intermediate certificate in a chain is used to sign the leaf certificate. An intermediate certificate can also be referred to as an intermediate CA certificate.

Why are intermediate certs useful?

Intermediate certificates are used in a variety of ways. For example, intermediate certificates can be used to group devices by product lines, customers purchasing devices, company divisions, or factories.

Imagine that Contoso is a large corporation with its own Public Key Infrastructure (PKI) using the root certificate

named *ContosoRootCert*. Each subsidiary of Contoso has their own intermediate certificate that is signed by *ContosoRootCert*. Each subsidiary will then use their intermediate certificate to sign their leaf certificates for each device. In this scenario, Contoso can use a single DPS instance where *ContosoRootCert* has been verified with [proof-of-possession](#). They can have an enrollment group for each subsidiary. This way each individual subsidiary will not have to worry about verifying certificates.

End-entity "leaf" certificate

The leaf certificate, or end-entity certificate, identifies the certificate holder. It has the root certificate in its certificate chain as well as zero or more intermediate certificates. The leaf certificate is not used to sign any other certificates. It uniquely identifies the device to the provisioning service and is sometimes referred to as the device certificate. During authentication, the device uses the private key associated with this certificate to respond to a proof of possession challenge from the service.

Leaf certificates used with [Individual enrollment](#) or [Enrollment group](#) entries must have the subject common name (CN) set to the registration ID. The registration ID identifies the device registration with DPS and must be unique to the DPS instance (ID scope) where the device registers. The registration ID is a case-insensitive string of alphanumeric characters plus the special characters: `'-'`, `'.'`, `'_'`, `:`. The last character must be alphanumeric or dash (`'-'`). DPS supports registration IDs up to 128 characters long; however, the maximum length of the subject common name in an X.509 certificate is 64 characters. The registration ID, therefore, is limited to 64 characters when using X.509 certificates.

For enrollment groups, the subject common name (CN) also sets the device ID that is registered with IoT Hub. The device ID will be shown in the [Registration Records](#) for the authenticated device in the enrollment group. For individual enrollments, the device ID can be set in the enrollment entry. If it's not set in the enrollment entry, then the subject common name (CN) is used.

To learn more, see [Authenticating devices signed with X.509 CA certificates](#).

Controlling device access to the provisioning service with X.509 certificates

The provisioning service exposes two enrollment types that you can use to control device access with the X.509 attestation mechanism:

- [Individual enrollment](#) entries are configured with the device certificate associated with a specific device. These entries control enrollments for specific devices.
- [Enrollment group](#) entries are associated with a specific intermediate or root CA certificate. These entries control enrollments for all devices that have that intermediate or root certificate in their certificate chain.

DPS device chain requirements

When a device is attempting registration through DPS using an enrollment group, the device must send the certificate chain from the leaf certificate to a certificate verified with [proof-of-possession](#). Otherwise, authentication will fail.

For example, if only the root certificate is verified and an intermediate certificate is uploaded to the enrollment group, the device should present the certificate chain from leaf certificate all the way to the verified root certificate. This certificate chain would include any intermediate certificates in-between. Authentication will fail if DPS cannot traverse the certificate chain to a verified certificate.

For example, consider a corporation using the following device chain for a device.



Only the root certificate is verified, and *intermediate2* certificate is uploaded on the enrollment group.



If the device only sends the following device chain during provisioning, authentication will fail. Because DPS can't attempt authentication assuming the validity of *intermediate1* certificate



If the device sends the full device chain as follows during provisioning, then DPS can attempt authentication of the device.



NOTE

Intermediate certificates can also be verified with [proof-of-possession..](#)

DPS order of operations with certificates

When a device connects to the provisioning service, the service prioritizes more specific enrollment entries over less specific enrollment entries. That is, if an individual enrollment for the device exists, the provisioning service applies that entry. If there is no individual enrollment for the device and an enrollment group for the first intermediate certificate in the device's certificate chain exists, the service applies that entry, and so on, down the chain to the root. The service applies the first applicable entry that it finds, such that:

- If the first enrollment entry found is enabled, the service provisions the device.
- If the first enrollment entry found is disabled, the service does not provision the device.
- If no enrollment entry is found for any of the certificates in the device's certificate chain, the service does not provision the device.

This mechanism and the hierarchical structure of certificate chains provides powerful flexibility in how you can control access for individual devices as well as for groups of devices. For example, imagine five devices with the following certificate chains:

- *Device 1*: root certificate -> certificate A -> device 1 certificate
- *Device 2*: root certificate -> certificate A -> device 2 certificate
- *Device 3*: root certificate -> certificate A -> device 3 certificate
- *Device 4*: root certificate -> certificate B -> device 4 certificate
- *Device 5*: root certificate -> certificate B -> device 5 certificate

Initially, you can create a single enabled group enrollment entry for the root certificate to enable access for all five devices. If certificate B later becomes compromised, you can create a disabled enrollment group entry for certificate B to prevent *Device 4* and *Device 5* from enrolling. If still later *Device 3* becomes compromised, you

can create a disabled individual enrollment entry for its certificate. This revokes access for *Device 3*, but still allows *Device 1* and *Device 2* to enroll.

TPM attestation

8/22/2022 • 4 minutes to read • [Edit Online](#)

IoT Hub Device Provisioning Service is a helper service for IoT Hub that you use to configure zero-touch device provisioning to a specified IoT hub. With the Device Provisioning Service, you can provision millions of devices in a secure manner.

This article describes the identity attestation process when using a Trusted Platform Module (TPM). A TPM is a type of hardware security module (HSM). This article assumes you are using a discrete, firmware, or integrated TPM. Software emulated TPMs are well-suited for prototyping or testing, but they do not provide the same level of security as discrete, firmware, or integrated TPMs do. We do not recommend using software TPMs in production. For more information about types of TPMs, see [A Brief Introduction to TPM](#).

This article is only relevant for devices using TPM 2.0 with HMAC key support and their endorsement keys. It is not for devices using X.509 certificates for authentication. TPM is an industry-wide, ISO standard from the Trusted Computing Group, and you can read more about TPM at the [complete TPM 2.0 spec](#) or the [ISO/IEC 11889 spec](#). This article also assumes you are familiar with public and private key pairs, and how they are used for encryption.

The Device Provisioning Service device SDKs handle everything that is described in this article for you. There is no need for you to implement anything additional if you are using the SDKs on your devices. This article helps you understand conceptually what's going on with your TPM security chip when your device provisions and why it's so secure.

Overview

TPMs use something called the endorsement key (EK) as the secure root of trust. The EK is unique to the TPM and changing it essentially changes the device into a new one.

There's another type of key that TPMs have, called the storage root key (SRK). An SRK may be generated by the TPM's owner after it takes ownership of the TPM. Taking ownership of the TPM is the TPM-specific way of saying "someone sets a password on the HSM." If a TPM device is sold to a new owner, the new owner can take ownership of the TPM to generate a new SRK. The new SRK generation ensures the previous owner can't use the TPM. Because the SRK is unique to the owner of the TPM, the SRK can be used to seal data into the TPM itself for that owner. The SRK provides a sandbox for the owner to store their keys and provides access revocability if the device or TPM is sold. It's like moving into a new house: taking ownership is changing the locks on the doors and destroying all furniture left by the previous owners (SRK), but you can't change the address of the house (EK).

Once a device has been set up and ready to use, it will have both an EK and an SRK available for use.



One note on taking ownership of the TPM: Taking ownership of a TPM depends on many things, including TPM manufacturer, the set of TPM tools being used, and the device OS. Follow the instructions relevant to your system to take ownership.

The Device Provisioning Service uses the public part of the EK (EK_pub) to identify and enroll devices. The device vendor can read the EK_pub during manufacture or final testing and upload the EK_pub to the provisioning service so that the device will be recognized when it connects to provision. The Device Provisioning Service does not check the SRK or owner, so “clearing” the TPM erases customer data, but the EK (and other vendor data) is preserved and the device will still be recognized by the Device Provisioning Service when it connects to provision.

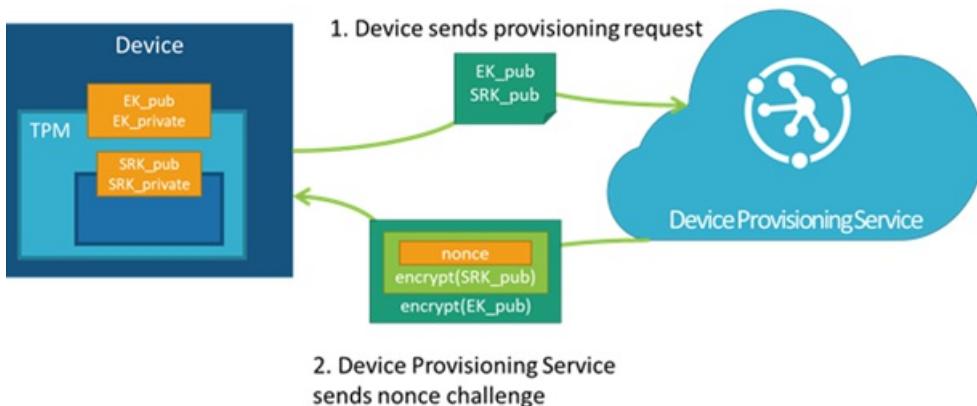
Detailed attestation process

When a device with a TPM first connects to the Device Provisioning Service, the service first checks the provided EK_pub against the EK_pub stored in the enrollment list. If the EK_pubs do not match, the device is not allowed to provision. If the EK_pubs do match, the service then requires the device to prove ownership of the private portion of the EK via a nonce challenge, which is a secure challenge used to prove identity. The Device Provisioning Service generates a nonce and then encrypts it with the SRK and then the EK_pub, both of which are provided by the device during the initial registration call. The TPM always keeps the private portion of the EK secure. This prevents counterfeiting and ensures SAS tokens are securely provisioned to authorized devices.

Let's walk through the attestation process in detail.

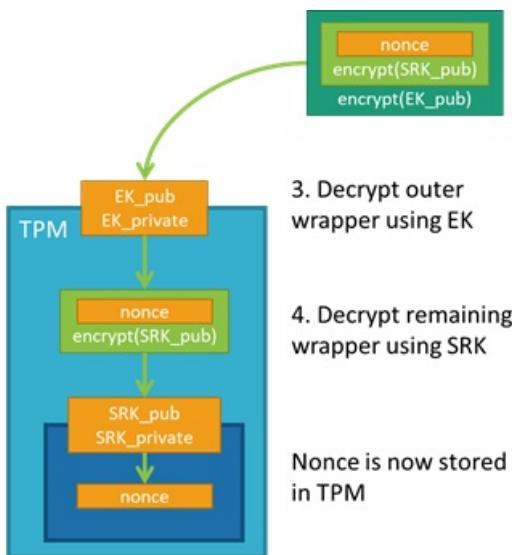
Device requests an IoT Hub assignment

First the device connects to the Device Provisioning Service and requests to provision. In doing so, the device provides the service with its registration ID, an ID scope, and the EK_pub and SRK_pub from the TPM. The service passes the encrypted nonce back to the device and asks the device to decrypt the nonce and use that to sign a SAS token to connect again and finish provisioning.



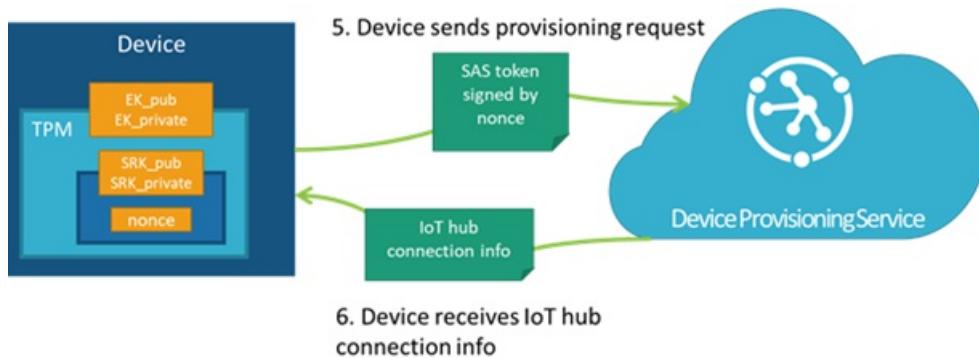
Nonce challenge

The device takes the nonce and uses the private portions of the EK and SRK to decrypt the nonce into the TPM; the order of nonce encryption delegates trust from the EK, which is immutable, to the SRK, which can change if a new owner takes ownership of the TPM.



Validate the nonce and receive credentials

The device can then sign a SAS token using the decrypted nonce and reestablish a connection to the Device Provisioning Service using the signed SAS token. With the Nonce challenge completed, the service allows the device to provision.



Next steps

Now the device connects to IoT Hub, and you rest secure in the knowledge that your devices' keys are securely stored. Now that you know how the Device Provisioning Service securely verifies a device's identity using TPM, check out the following articles to learn more:

- [Learn about the concepts of provisioning](#)
- [Get started using auto-provisioning](#)
- [Create TPM enrollments using the SDKs](#)

Control access to Azure IoT Hub Device Provisioning Service (DPS)

8/22/2022 • 2 minutes to read • [Edit Online](#)

This article describes the available options for securing your Azure IoT Hub Device Provisioning Service (DPS). The provisioning service uses *authentication* and *permissions* to grant access to each endpoint. Permissions allow the authentication process to limit access to a service instance based on functionality.

There are two different ways for controlling access to DPS:

- **Shared access signatures** lets you group permissions and grant them to applications using access keys and signed security tokens. To learn more, see [Control access to DPS with shared access signatures and security tokens](#).
- **Azure Active Directory (Azure AD) integration (public preview)** for service APIs. Azure provides identity-based authentication with Azure Active Directory and fine-grained authorization with Azure role-based access control (Azure RBAC). Azure AD and RBAC integration is supported for DPS service APIs only. To learn more, see [Control access to DPS with Azure Active Directory \(Public Preview\)](#).

Next steps

- [Control access to DPS with shared access signatures and security tokens](#)
- [Control access to DPS with Azure Active Directory \(public preview\)](#)

Control access to Azure IoT Hub Device Provisioning Service (DPS) with shared access signatures and security tokens

8/22/2022 • 8 minutes to read • [Edit Online](#)

This article describes the available options for securing your Azure IoT Hub Device Provisioning Service (DPS). The provisioning service uses *authentication* and *permissions* to grant access to each endpoint. Permissions allow the authentication process to limit access to a service instance based on functionality.

This article discusses:

- The authentication process and the tokens the provisioning service uses to verify permissions against both the [Service and Device REST APIs](#).
- The different permissions that you can grant to a backend app to access the Service API.

Authentication

The Device API supports key-based and X.509 certificate-based device authentication.

The Service API supports key-based authentication for backend apps.

When using key-based authentication, the Device Provisioning Service uses security tokens to authenticate services to avoid sending keys on the wire. Additionally, security tokens are limited in time validity and scope. Azure IoT Device Provisioning SDKs automatically generate tokens without requiring any special configuration.

In some cases you may need to use the HTTP Device Provisioning Service REST APIs directly, without using the SDKs. The following sections describe how to authenticate directly against the REST APIs.

Device API authentication

The [Device API](#) is used by devices to attest to the Device Provisioning Service and receive an IoT Hub connection.

NOTE

In order to receive an authenticated connection, devices must first be registered in the Device Provisioning Service through an enrollment. Use the Service API to programmatically register a device through an enrollment.

A device must authenticate to the Device API as part of the provisioning process. The method a device uses to authenticate is defined when you set up an enrollment group or individual enrollment. Whatever the authentication method, the device must issue an HTTPS PUT to the following URL to provision itself.

```
https://global.azure-devices-provisioning.net/[ID_Scope]/registrations/[registration_id]/register?api-version=2021-06-01
```

If using key-based authentication, a security token is passed in the `HTTPAuthorization` request header in the following format:

```
SharedAccessSignature sig={signature}&se={expiry}&skn={policyName}&sr={URL-encoded-resourceURI}
```

Security token structure for key-based authentication

The security token is passed in the `HTTPAuthorization` request header in the following format:

```
SharedAccessSignature sig={signature}&se={expiry}&skn={policyName}&sr={URL-encoded-resourceURI}
```

The expected values are:

VALUE	DESCRIPTION
{signature}	An HMAC-SHA256 signature string of the form: <code>{URL-encoded-resourceURI} + "\n" + expiry</code> Important: The key is decoded from base64 and used as key to do the HMAC-SHA256 computation.
{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 <code>{ID_Scope}/registrations/{registration_id}</code>
{policyName}	For the Device API, this policy is always "registration".

The following Python snippet shows a function called `generate_sas_token` that computes the token from the inputs `uri`, `key`, `policy_name`, `expiry` for an individual enrollment using a symmetric key authentication type.

```
frombase64importb64encode,b64decode,encode
fromhashlibimportsha256
fromtimeimporttime
fromurllib.parseimportquote_plus,urlencode
fromhmacimportHMAC

defgenerate_sas_token(uri,key,policy_name,expiry=3600):
    ttl=time()+expiry
    sign_key="%s\n%d"%(quote_plus(uri),int(ttl))
    signature=b64encode(HMAC(b64decode(key),sign_key.encode('utf-8'),sha256).digest())

    rawtoken={
        'sr':uri,
        'sig':signature,
        'se':str(int(ttl)),
        'skn':policy_name
    }

    return'SharedAccessSignature'+urlencode(rawtoken)

print(generate_sas_token("myIdScope/registrations/mydeviceregistrationid","00mysymmetrickey","registration")
)
```

The result should resemble the following output:

```
SharedAccessSignature
sr=myIdScope%2Fregistrations%2Fmydeviceregistrationid&sig=SDpdbUNK%2F1DSjEpeb29BLVe6gRDZI7T41Y4BPsHHoUg%3D&s
e=1630175722&skn=registration
```

The following example shows how the shared access signature is then used to authenticate with the Device API.

```
curl -L -i -X PUT -H 'Content-Type: application/json' -H 'Content-Encoding: utf-8' -H 'Authorization: [token]' -d '{"registrationId": "[registration_id]"}' https://global.azure-devices-provisioning.net/[ID_Scope]/registrations/[registration_id]/register?api-version=2021-06-01
```

If using a symmetric key-based enrollment group, you'll need to first generate a `device_symmetric` key using the enrollment group key. Use the enrollment group primary or secondary key to compute an HMAC-SHA256 of the registration ID for the device. The result is then converted into Base64 format to obtain the derived device key. To view code examples, see [How to provision devices using symmetric key enrollment groups](#). Once the device symmetric key has been derived, you can register the device using the previous examples.

WARNING

To avoid including the group master key in your device code, the process of deriving device key should be done off the device.

Certificate-based authentication

If you've set up an individual enrollment or enrollment group for X.509 certificated-based authentication, the device will need to use its issued X.509 certificate to attest to Device API. Refer to the following articles on how to set up the enrollment and generate the device certificate.

- Quickstart - [Provision simulated X.509 device to Azure IoT Hub](#)
- Quickstart - [Enroll X.509 devices to Azure Device Provisioning Service](#)

Once the enrollment has been set up and the device certificate issued, the following example demonstrates how to authenticate to Device API with the device's X.509 certificate.

```
curl -L -i -X PUT -cert ./[device_cert].pem -key ./[device_cert_private_key].pem -H 'Content-Type: application/json' -H 'Content-Encoding: utf-8' -d '{"registrationId": "[registration_id]"}' https://global.azure-devices-provisioning.net/[ID_Scope]/registrations/[registration_id]/register?api-version=2021-06-01
```

Service API authentication

The [Service API](#) is used to retrieve registration state and remove device registrations. The service is also used by backend apps to programmatically manage both [individual groups](#) and [enrollment groups](#). The Service API supports key-based authentication for backend apps.

You must have appropriate permissions to access any of the Service API endpoints. For example, a backend app must include a token containing security credentials along with every message it sends to the service.

Azure IoT Hub Device Provisioning Service grants access to endpoints by verifying the token against the shared access policies. Security credentials, such as symmetric keys, are never sent over the wire.

Access control and permissions

You can grant [permissions](#) in the following ways:

- **Shared access authorization policies.** Shared access policies can grant any combination of [permissions](#). You can define policies in the [Azure portal](#), or programmatically by using the [Device Provisioning Service REST APIs](#). A newly created provisioning service has the following default policy:

- **provisioningserviceowner**: Policy with all permissions. See [permissions](#) for detailed information.

NOTE

The Device Provisioning Service 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 the next section.

HTTP is the only supported protocol, and it implements authentication by including a valid token in the **Authorization** request header.

Example

```
SharedAccessSignature sr =
    mydps.azure-devices-
provisioning.net&sig=kPsxZZZZZZZZZZZZZZAhLT%2bV7o%3d&se=1487709501&skn=provisioningserviceowner` \
```

NOTE

The [Azure IoT Device Provisioning Service SDKs](#) automatically generate tokens when connecting to the service.

Security tokens

The Device Provisioning Service uses security tokens to authenticate services to avoid sending keys on the wire. Additionally, security tokens are limited in time validity and scope. [Azure IoT Device Provisioning Service 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 HTTP surface.

Security token structure

You use security tokens to grant time-bounded access for services to specific functionality in IoT Device Provisioning Service. To get authorization to connect to the provisioning service, services must send security tokens signed with either a shared access or symmetric key.

A token signed with a shared access key grants access to all the functionality associated with the shared access policy permissions.

The security token has the following format:

```
SharedAccessSignature sig={signature}&se={expiry}&skn={policyName}&sr={URL-encoded-resourceURI}
```

Here are the expected values

VALUE	DESCRIPTION
{signature}	An HMAC-SHA256 signature string of the form: <code>{URL-encoded-resourceURI} + "\n" + expiry</code> . Important: The key is decoded from base64 and used as key to do the HMAC-SHA256 computation.
{expiry}	UTF8 strings for number of seconds since the epoch 00:00:00 UTC on 1 January 1970.

VALUE	DESCRIPTION
{URL-encoded-resourceURI}	Lower case URL-encoding of the lower case resource URI. URI prefix (by segment) of the endpoints that can be accessed with this token, starting with host name of the IoT Device Provisioning Service (no protocol). For example, <code>mydps.azure-devices-provisioning.net</code> .
{policyName}	The name of the shared access policy to which this token refers.

NOTE

The URI prefix is computed by segment and not by character. For example `/a/b` is a prefix for `/a/b/c` but not for `/a/bc`.

The following Node.js snippet shows a function called `generateSasToken` that computes the token from the inputs `resourceUri, signingKey, policyName, expiresInMins`. The next sections detail how to initialize the different inputs for the different token use cases.

```
var generateSasToken = function(resourceUri, signingKey, policyName, expiresInMins) {
    resourceUri = encodeURIComponent(resourceUri);

    // Set expiration in seconds
    var expires = (Date.now() / 1000) + expiresInMins * 60;
    expires = Math.ceil(expires);
    var toSign = resourceUri + '\n' + expires;

    // Use crypto
    var hmac = crypto.createHmac('sha256', new Buffer(signingKey, 'base64'));
    hmac.update(toSign);
    var base64UriEncoded = encodeURIComponent(hmac.digest('base64'));

    // Construct authorization string
    var token = "SharedAccessSignature sr=" + resourceUri + "&sig=";
    token += base64UriEncoded + "&se=" + expires + "&skn=" + policyName;
    return token;
};
```

As a comparison, the equivalent Python code to generate a security token is:

```

from base64 import b64encode, b64decode
from hashlib import sha256
from time import time
from urllib import quote_plus, urlencode
from hmac import HMAC

def generate_sas_token(uri, key, policy_name, expiry=3600):
    ttl = time() + expiry
    sign_key = "%s\n%d" % ((quote_plus(uri)), int(ttl))
    print sign_key
    signature = b64encode(HMAC(b64decode(key), sign_key, sha256).digest())

    rawtoken = {
        'sr' : uri,
        'sig': signature,
        'se' : str(int(ttl)),
        'skn' : policy_name
    }

    return 'SharedAccessSignature ' + urlencode(rawtoken)

```

NOTE

Since the time validity of the token is validated on IoT Device Provisioning Service machines, the drift on the clock of the machine that generates the token must be minimal.

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
{your-service}.azure-devices-provisioning.net/enrollments	Provides device enrollment operations with the Device Provisioning Service.
{your-service}.azure-devices-provisioning.net/enrollmentGroups	Provides operations for managing device enrollment groups.
{your-service}.azure-devices-provisioning.net/registrations/{id}	Provides operations for retrieving and managing the status of device registrations.

As an example, a service generated using a pre-created shared access policy called `enrollmentread` would create a token with the following parameters:

- resource URI: `{mydps}.azure-devices-provisioning.net`,
- signing key: one of the keys of the `enrollmentread` policy,
- policy name: `enrollmentread`,
- any expiration time.backn

The screenshot shows the Azure portal interface for managing Device Provisioning Service. On the left, there's a sidebar with various options like Overview, Activity log, and Shared access policies (which is highlighted with a red box). In the main area, there's a table titled 'Add Access Policy' with columns 'POLICY' and 'PERMISSIONS'. A row shows a policy named 'provisioningserviceowner' with permissions 'ServiceConfig, DeviceConnect, EnrollmentWrite'. To the right of this table is another window titled 'Add Access Policy' with fields for 'Name' (set to 'enrollmentread') and a list of permissions (with 'Enrollment read' checked). A large red box highlights the 'Save' button at the bottom of this window.

```
var endpoint ="mydps.azure-devices-provisioning.net";
var policyName = 'enrollmentread';
var policyKey = '...';

var token = generateSasToken(endpoint, policyKey, policyName, 60);
```

The result, which would grant access to read all enrollment records, would be:

```
SharedAccessSignature sr=mydps.azure-devices-
provisioning.net&sig=JdyscqTpXdeEs49e1IUcohw2D1FDR3zfH5KqGJo4r4%3D&se=1456973447&skn=enrollmentread
```

SDKs and samples

- [Azure IoT SDK for Java Preview Release](#)
 - [Sample](#)
- [Microsoft Azure IoT SDKs for .NET Preview Release](#)
 - [Sample](#)

Reference topics:

The following reference topics provide you with more information about controlling access to your IoT Device Provisioning Service.

Device Provisioning Service permissions

The following table lists the permissions you can use to control access to your IoT Device Provisioning Service.

PERMISSION	NOTES
ServiceConfig	Grants access to change the service configurations. This permission is used by backend cloud services.
EnrollmentRead	Grants read access to the device enrollments and enrollment groups. This permission is used by backend cloud services.
EnrollmentWrite	Grants write access to the device enrollments and enrollment groups. This permission is used by backend cloud services.

PERMISSION	NOTES
RegistrationStatusRead	Grants read access to the device registration status. This permission is used by backend cloud services.
RegistrationStatusWrite	Grants delete access to the device registration status. This permission is used by backend cloud services.

Control access to Azure IoT Hub Device Provisioning Service (DPS) by using Azure Active Directory (preview)

8/22/2022 • 5 minutes to read • [Edit Online](#)

You can use Azure Active Directory (Azure AD) to authenticate requests to Azure IoT Hub Device Provisioning Service (DPS) APIs, like create device identity and invoke direct method. You can also use Azure role-based access control (Azure RBAC) to authorize those same service APIs. By using these technologies together, you can grant permissions to access Azure IoT Hub Device Provisioning Service (DPS) APIs to an Azure AD security principal. This security principal could be a user, group, or application service principal.

Authenticating access by using Azure AD and controlling permissions by using Azure RBAC provides improved security and ease of use over [security tokens](#). To minimize potential security issues inherent in security tokens, we recommend that you use Azure AD with your Azure IoT Hub Device Provisioning Service (DPS) whenever possible.

NOTE

Authentication with Azure AD isn't supported for the Azure IoT Hub Device Provisioning Service (DPS) *device APIs* (like register device or device registration status lookup). Use [symmetric keys](#), [X.509](#) or [TPM](#) to authenticate devices to Azure IoT Hub Device Provisioning Service (DPS).

Authentication and authorization

When an Azure AD security principal requests access to an Azure IoT Hub Device Provisioning Service (DPS) API, the principal's identity is first *authenticated*. For authentication, the request needs to contain an OAuth 2.0 access token at runtime. The resource name for requesting the token is <https://iothubs.azure.net>. If the application runs in an Azure resource like an Azure VM, Azure Functions app, or Azure App Service app, it can be represented as a [managed identity](#).

After the Azure AD principal is authenticated, the next step is *authorization*. In this step, Azure IoT Hub Device Provisioning Service (DPS) uses the Azure AD role assignment service to determine what permissions the principal has. If the principal's permissions match the requested resource or API, Azure IoT Hub Device Provisioning Service (DPS) authorizes the request. So this step requires one or more Azure roles to be assigned to the security principal. Azure IoT Hub Device Provisioning Service (DPS) provides some built-in roles that have common groups of permissions.

Manage access to Azure IoT Hub Device Provisioning Service (DPS) by using Azure RBAC role assignment

With Azure AD and RBAC, Azure IoT Hub Device Provisioning Service (DPS) requires the principal requesting the API to have the appropriate level of permission for authorization. To give the principal the permission, give it a role assignment.

- If the principal is a user, group, or application service principal, follow the guidance in [Assign Azure roles by using the Azure portal](#).
- If the principal is a managed identity, follow the guidance in [Assign a managed identity access to a resource by using the Azure portal](#).

To ensure least privilege, always assign the appropriate role at the lowest possible [resource scope](#), which is probably the Azure IoT Hub Device Provisioning Service (DPS) scope.

Azure IoT Hub Device Provisioning Service (DPS) provides the following Azure built-in roles for authorizing access to DPS APIs by using Azure AD and RBAC:

ROLE	DESCRIPTION
Device Provisioning Service Data Contributor	Allows for full access to Device Provisioning Service data-plane operations.
Device Provisioning Service Data Reader	Allows for full read access to Device Provisioning Service data-plane properties.

You can also define custom roles to use with Azure IoT Hub Device Provisioning Service (DPS) by combining the [permissions](#) that you need. For more information, see [Create custom roles for Azure role-based access control](#).

Resource scope

Before you assign an Azure RBAC role to a security principal, determine the scope of access that the security principal should have. It's always best to grant only the narrowest possible scope. Azure RBAC roles defined at a broader scope are inherited by the resources beneath them.

This list describes the levels at which you can scope access to IoT Hub, starting with the narrowest scope:

- **The Azure IoT Hub Device Provisioning Service (DPS).** At this scope, a role assignment applies to the Azure IoT Hub Device Provisioning Service (DPS). Role assignment at smaller scopes, like enrollment group or individual enrollment, isn't supported.
- **The resource group.** At this scope, a role assignment applies to all IoT hubs in the resource group.
- **The subscription.** At this scope, a role assignment applies to all IoT hubs in all resource groups in the subscription.
- **A management group.** At this scope, a role assignment applies to all IoT hubs in all resource groups in all subscriptions in the management group.

Permissions for Azure IoT Hub Device Provisioning Service (DPS) APIs

The following table describes the permissions available for Azure IoT Hub Device Provisioning Service (DPS) API operations. To enable a client to call a particular operation, ensure that the client's assigned RBAC role offers sufficient permissions for the operation.

RBAC ACTION	DESCRIPTION
Microsoft.Devices/provisioningServices/attestationmechanisms/read	Fetch attestation mechanism details
Microsoft.Devices/provisioningServices/enrollmentGroups/read	Read enrollment groups
Microsoft.Devices/provisioningServices/enrollmentGroups/write	Write enrollment groups
Microsoft.Devices/provisioningServices/enrollmentGroups/delete	Delete enrollment groups
Microsoft.Devices/provisioningServices/enrollments/read	Read enrollments
Microsoft.Devices/provisioningServices/enrollments/write	Write enrollments

RBAC ACTION	DESCRIPTION
Microsoft.Devices/provisioningServices/enrollments/delete	Delete enrollments
Microsoft.Devices/provisioningServices/registrationStates/read	Read registration states
Microsoft.Devices/provisioningServices/registrationStates/delete	Delete registration states

Azure IoT extension for Azure CLI

Most commands against Azure IoT Hub Device Provisioning Service (DPS) support Azure AD authentication. You can control the type of authentication used to run commands by using the `--auth-type` parameter, which accepts `key` or `login` values. The `key` value is the default.

- When `--auth-type` has the `key` value, the CLI automatically discovers a suitable policy when it interacts with Azure IoT Hub Device Provisioning Service (DPS).
- When `--auth-type` has the `login` value, an access token from the Azure CLI logged in the principal is used for the operation.
- The following commands currently support `--auth-type`:
 - `az iot dps enrollment`
 - `az iot dps enrollment-group`
 - `az iot dps registration`

For more information, see the [Azure IoT extension for Azure CLI release page](#).

SDKs and samples

- [Azure IoT SDKs for Node.js Provisioning Service](#)
 - [Sample](#)
- [Azure IoT SDK for Java Preview Release](#)
 - [Sample](#)
- [Microsoft Azure IoT SDKs for .NET Preview Release](#)
 - [Sample](#)

Azure AD access from the Azure portal

NOTE

Azure AD access from the Azure portal is currently not available during preview.

Next steps

- For more information on the advantages of using Azure AD in your application, see [Integrating with Azure Active Directory](#).
- For more information on requesting access tokens from Azure AD for users and service principals, see [Authentication scenarios for Azure AD](#).

Roles and operations

8/22/2022 • 6 minutes to read • [Edit Online](#)

The phases of developing an IoT solution can span weeks or months, due to production realities like manufacturing time, shipping, customs process, etc. In addition, they can span activities across multiple roles given the various entities involved. This topic takes a deeper look at the various roles and operations related to each phase, then illustrates the flow in a sequence diagram.

Provisioning also places requirements on the device manufacturer, specific to enabling the [attestation mechanism](#). Manufacturing operations can also occur independent of the timing of auto-provisioning phases, especially in cases where new devices are procured after auto-provisioning has already been established.

A series of Quickstarts are provided in the table of contents to the left, to help explain auto-provisioning through hands-on experience. In order to facilitate/simplify the learning process, software is used to simulate a physical device for enrollment and registration. Some Quickstarts require you to fulfill operations for multiple roles, including operations for non-existent roles, due to the simulated nature of the Quickstarts.

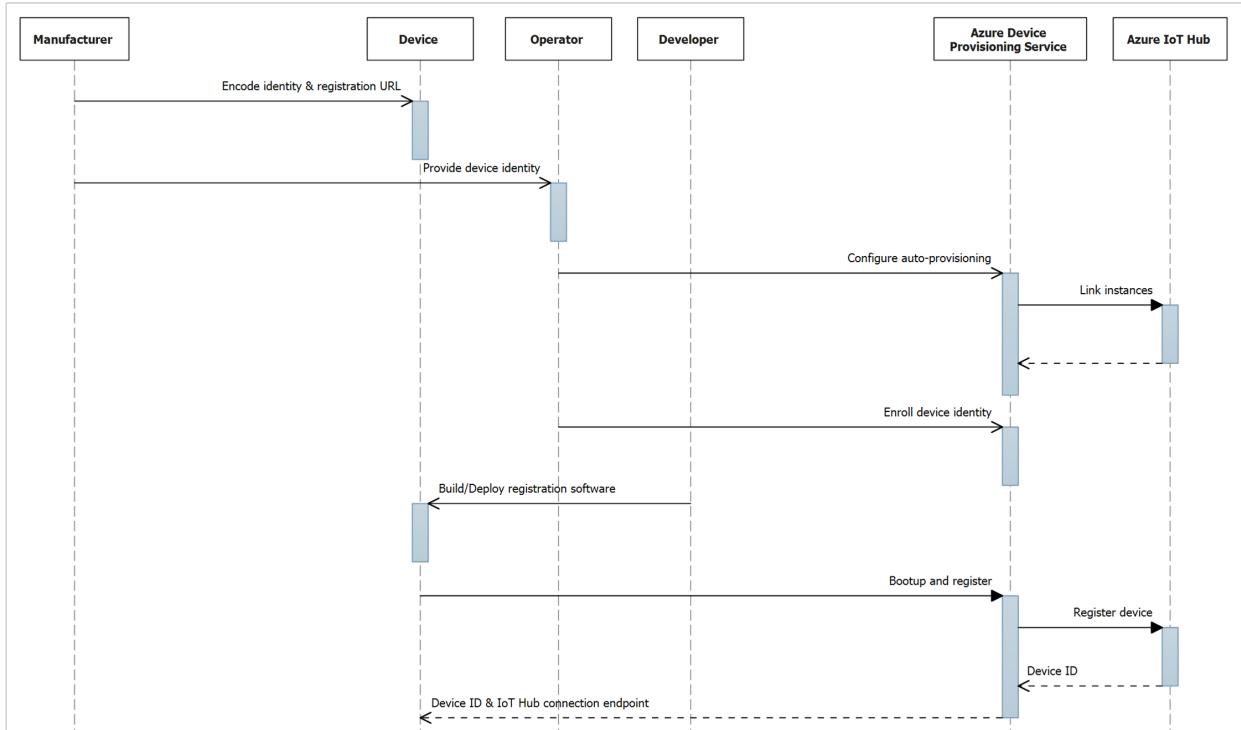
ROLE	OPERATION	DESCRIPTION
Manufacturer	Encode identity and registration URL	<p>Based on the attestation mechanism used, the manufacturer is responsible for encoding the device identity info, and the Device Provisioning Service registration URL.</p> <p>Quickstarts: since the device is simulated, there is no Manufacturer role. See the Developer role for details on how you get this information, which is used in coding a sample registration application.</p>
	Provide device identity	<p>As the originator of the device identity info, the manufacturer is responsible for communicating it to the operator (or a designated agent), or directly enrolling it to the Device Provisioning Service via APIs.</p> <p>Quickstarts: since the device is simulated, there is no Manufacturer role. See the Operator role for details on how you get the device identity, which is used to enroll a simulated device in your Device Provisioning Service instance.</p>
Operator	Configure auto-provisioning	<p>This operation corresponds with the first phase of auto-provisioning.</p> <p>Quickstarts: You perform the Operator role, configuring the Device Provisioning Service and IoT Hub instances in your Azure subscription.</p>

ROLE	OPERATION	DESCRIPTION
	Enroll device identity	<p>This operation corresponds with the second phase of auto-provisioning.</p> <p>Quickstarts: You perform the Operator role, enrolling your simulated device in your Device Provisioning Service instance. The device identity is determined by the attestation method being simulated in the Quickstart (TPM or X.509). See the Developer role for attestation details.</p>
Device Provisioning Service, IoT Hub	<all operations>	<p>For both a production implementation with physical devices, and Quickstarts with simulated devices, these roles are fulfilled via the IoT services you configure in your Azure subscription. The roles/operations function exactly the same, as the IoT services are indifferent to provisioning of physical vs. simulated devices.</p>

ROLE	OPERATION	DESCRIPTION
Developer	Build/Deploy registration software	<p>This operation corresponds with the third phase of auto-provisioning. The Developer is responsible for building and deploying the registration software to the device, using the appropriate SDK.</p> <p>Quickstarts: The sample registration application you build simulates a real device, for your platform/language of choice, which runs on your workstation (instead of deploying it to a physical device). The registration application performs the same operations as one deployed to a physical device. You specify the attestation method (TPM or X.509 certificate), plus the registration URL and "ID Scope" of your Device Provisioning Service instance. The device identity is determined by the SDK attestation logic at runtime, based on the method you specify:</p> <ul style="list-style-type: none"> • TPM attestation - your development workstation runs a TPM simulator application. Once running, a separate application is used to extract the TPM's "Endorsement Key" and "Registration ID" for use in enrolling the device identity. The SDK attestation logic also uses the simulator during registration, to present a signed SAS token for authentication and enrollment verification. • X509 attestation - you use a tool to generate a certificate. Once generated, you create the certificate file required for use in enrollment. The SDK attestation logic also uses the certificate during registration, to present for authentication and enrollment verification.

ROLE	OPERATION	DESCRIPTION
Device	Bootup and register	<p>This operation corresponds with the third phase of auto-provisioning, fulfilled by the device registration software built by the Developer. See the Developer role for details. Upon first boot:</p> <ol style="list-style-type: none"> 1. The application connects with the Device Provisioning Service instance, per the global URL and service "ID Scope" specified during development. 2. Once connected, the device is authenticated against the attestation method and identity specified during enrollment. 3. Once authenticated, the device is registered with the IoT Hub instance specified by the provisioning service instance. 4. Upon successful registration, a unique device ID and IoT Hub endpoint are returned to the registration application for communicating with IoT Hub. 5. From there, the device can pull down its initial device twin state for configuration, and begin the process of reporting telemetry data. <p>Quickstarts: since the device is simulated, the registration software runs on your development workstation.</p>

The following diagram summarizes the roles and sequencing of operations during device auto-provisioning:



NOTE

Optionally, the manufacturer can also perform the "Enroll device identity" operation using Device Provisioning Service APIs (instead of via the Operator). For a detailed discussion of this sequencing and more, see the [Zero touch device registration with Azure IoT video](#) (starting at marker 41:00)

Roles and Azure accounts

How each role is mapped to an Azure account is scenario-dependent, and there are quite a few scenarios that can be involved. The common patterns below should help provide a general understanding regarding how roles are generally mapped to an Azure account.

Chip manufacturer provides security services

In this scenario, the manufacturer manages security for level-one customers. This scenario may be preferred by these level-one customers as they don't have to manage detailed security.

The manufacturer introduces security into Hardware Security Modules (HSMs). This security can include the manufacturer obtaining keys, certificates, etc. from potential customers who already have DPS instances and enrollment groups setup. The manufacturer could also generate this security information for its customers.

In this scenario, there may be two Azure accounts involved:

- **Account #1:** Likely shared across the operator and developer roles to some degree. This party may purchase the HSM chips from the manufacturer. These chips are pointed to DPS instances associated with the Account #1. With DPS enrollments, this party can lease devices to multiple level-two customers by reconfiguring the device enrollment settings in DPS. This party may also have IoT hubs allocated for end-user backend systems to interface with in order to access device telemetry etc. In this latter case, a second account may not be needed.
- **Account #2:** End users, level-two customers may have their own IoT hubs. The party associated with Account #1 just points leased devices to the correct hub in this account. This configuration requires linking DPS and IoT hubs across Azure accounts, which can be done with Azure Resource Manager templates.

All-in-one OEM

The manufacturer could be an "All-in-one OEM" where only a single manufacturer account would be needed. The manufacturer handles security and provisioning end to end.

The manufacturer may provide a cloud-based application to customers who purchase devices. This application would interface with the IoT Hub allocated by the manufacturer.

Vending machines or automated coffee machines represent examples for this scenario.

Next steps

You may find it helpful to bookmark this article as a point of reference, as you work your way through the corresponding auto-provisioning Quickstarts.

Begin by completing a "Set up auto-provisioning" Quickstart that best suits your management tool preference, which walks you through the "Service configuration" phase:

- [Set up auto-provisioning using Azure CLI](#)
- [Set up auto-provisioning using the Azure portal](#)
- [Set up auto-provisioning using a Resource Manager template](#)

Then continue with a "Provision a device" Quickstart that suits your device attestation mechanism and Device

Provisioning Service SDK/language preference. In this Quickstart, you walk through the "Device enrollment" and "Device registration and configuration" phases:

DEVICE ATTESTATION MECHANISM	QUICKSTART
Symmetric key	Provision a simulated symmetric key device
X.509 certificate	Provision a simulated X.509 device
Simulated Trusted Platform Module (TPM)	Provision a simulated TPM device

Azure IoT Hub Device Provisioning Service (DPS) support for virtual networks

8/22/2022 • 7 minutes to read • [Edit Online](#)

This article introduces the virtual network (VNET) connectivity pattern for IoT devices provisioning with IoT hubs using DPS. This pattern provides private connectivity between the devices, DPS, and the IoT hub inside a customer-owned Azure VNET.

In most scenarios where DPS is configured with a VNET, your IoT Hub will also be configured in the same VNET. For more specific information on VNET support and configuration for IoT Hubs, see, [IoT Hub virtual network support](#).

Introduction

By default, DPS hostnames map to a public endpoint with a publicly routable IP address over the Internet. This public endpoint is visible to all customers. Access to the public endpoint can be attempted by IoT devices over wide-area networks and on-premises networks.

For several reasons, customers may wish to restrict connectivity to Azure resources, like DPS. These reasons include:

- Prevent connection exposure over the public Internet. Exposure can be reduced by introducing more layers of security via network level isolation for your IoT hub and DPS resources
- 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](#).

Common approaches to restricting connectivity include [DPS IP filter rules](#) and Virtual networking (VNET) with [private endpoints](#). The goal of this article is to describe the VNET approach for DPS using private endpoints.

Devices that operate in on-premises networks can use [Virtual Private Network \(VPN\)](#) or [ExpressRoute](#) private peering to connect to a VNET in Azure and access DPS resources through private endpoints.

A private endpoint is a private IP address allocated inside a customer-owned VNET by which an Azure resource is accessible. By having a private endpoint for your DPS resource, you will be able to allow devices operating inside your VNET to request provisioning by your DPS resource without allowing traffic to the public endpoint. Each DPS resource can support multiple private endpoints, each of which may be located in a VNET in a different region.

Prerequisites

Before proceeding ensure that the following prerequisites are met:

- Your DPS resource is already created and linked to your IoT hubs. For guidance on setting up a new DPS resource, see, [Set up IoT Hub Device Provisioning Service with the Azure portal](#)
- You have provisioned an Azure VNET with a subnet in which the private endpoint will be created. For more information, see, [create a virtual network using Azure CLI](#).

- For devices that operate inside of on-premises networks, set up [Virtual Private Network \(VPN\)](#) or [ExpressRoute](#) private peering into your Azure VNET.

Private endpoint limitations

Note the following current limitations for DPS when using private endpoints:

- Private endpoints will not work with DPS when the DPS resource and the linked Hub are in different clouds. For example, [Azure Government](#) and [global Azure](#).
- Currently, [custom allocation policies with Azure Functions](#) for DPS will not work when the Azure function is locked down to a VNET and private endpoints.
- Current DPS VNET support is for data ingress into DPS only. Data egress, which is the traffic from DPS to IoT Hub, uses an internal service-to-service mechanism rather than a dedicated VNET. Support for full VNET-based egress lockdown between DPS and IoT Hub is not currently available.
- The lowest latency allocation policy is used to assign a device to the IoT hub with the lowest latency. This allocation policy is not reliable in a virtual network environment.
- Enabling one or more private endpoints typically involves [disabling public access](#) to your DPS instance. This means that you can no longer use the Azure portal to manage enrollments. Instead you can manage enrollments using the Azure CLI, PowerShell, or service APIs from machines inside the VNET(s)/private endpoint(s) configured on the DPS instance.
- When using private endpoints, we recommend deploying DPS in one of the regions that support [Availability Zones](#). Otherwise, DPS instances with private endpoints enabled may see reduced availability in the event of outages.

NOTE

Data residency consideration:

DPS provides a **Global device endpoint** (`global.azure-devices-provisioning.net`). However, when you use the global endpoint, your data may be redirected outside of the region where the DPS instance was initially created. To ensure data residency within the initial DPS region, use private endpoints.

Set up a private endpoint

To set up a private endpoint, follow these steps:

- In the [Azure portal](#), open your DPS resource and click the **Networking** tab. Click **Private endpoint connections** and + **Private endpoint**.

The screenshot shows the Azure Device Provisioning Service (DPS) Networking blade. The 'Private endpoint connections' tab is active. At the top, there's a search bar and navigation buttons for 'Public access' and 'Private endpoint connections'. Below that is a toolbar with 'Approve', 'Reject', 'Refresh', and 'Remove' buttons. A dropdown menu shows 'All connection states'. The main area has columns for 'Connection Name', 'Connection State', 'Private Endpoint', and 'Description'. In the left sidebar, under 'Settings', the 'Networking' option is highlighted with a red box. Other options like 'Quick Start', 'Shared access policies', 'Linked IoT hubs', 'Certificates', 'Manage enrollments', and 'Manage allocation policy' are also listed.

2. On the *Create a private endpoint Basics* page, enter the information mentioned in the table below.

The screenshot shows the 'Create a private endpoint' Basics step. It includes tabs for 'Basics', 'Resource', 'Configuration', 'Tags', and 'Review + create'. A note says: 'Use private endpoints to privately connect to a service or resource. Your private endpoint must be in the same region as your virtual network, but can be in a different region from the private link resource that you are connecting to.' Below are sections for 'Project details' (Subscription and Resource group) and 'Instance details' (Name and Region). The 'Resource group' dropdown and the 'Name' input field are both highlighted with a red box. At the bottom, the 'Next : Resource >' button is also highlighted with a red box.

FIELD	VALUE
Subscription	Choose the desired Azure subscription to contain the private endpoint.
Resource group	Choose or create a resource group to contain the private endpoint
Name	Enter a name for your private endpoint
Region	The region chosen must be the same as the region that contains the VNET, but it does not have to be the same as the DPS resource.

Click **Next : Resource** to configure the resource that the private endpoint will point to.

3. On the *Create a private endpoint Resource* page, enter the information mentioned in the table below.

Create a private endpoint

X

 Basics Resource Configuration Tags Review + create

Private Link offers options to create private endpoints for different Azure resources, like your private link service, a SQL server, or an Azure storage account. Select which resource you would like to connect to using this private endpoint. [Learn more](#)

Connection method (1) Connect to an Azure resource in my directory. Connect to an Azure resource by resource ID or alias.Subscription (1)

Your Azure Subscription

Resource type (1)

Microsoft.Devices/ProvisioningServices

Resource (1)

test-dps-docs

Target sub-resource (1)

iotDps

< Previous

Next : Configuration >

FIELD	VALUE
Subscription	Choose the Azure subscription that contains the DPS resource that your private endpoint will point to.
Resource type	Choose Microsoft.Devices/ProvisioningServices .
Resource	Select the DPS resource that the private endpoint will map to.
Target sub-resource	Select iotDps .

TIP

Information on the **Connect to an Azure resource by resource ID or alias** setting is provided in the [Request a private endpoint](#) section in this article.

Click **Next : Configuration** to configure the VNET for the private endpoint.

4. On the *Create a private endpoint Configuration* page, choose your virtual network and subnet to create the private endpoint in.

Click **Next : Tags**, and optionally provide any tags for your resource.

5. Click **Review + create** and then **Create** to create your private endpoint resource.

Use private endpoints with devices

To use private endpoints with device provisioning code, your provisioning code must use the specific **Service endpoint** for your DPS instance as shown on the overview page of your DPS instance in the [Azure portal](#). The service endpoint has the following form.

```
<Your DPS Tenant Name>.azure-devices-provisioning.net
```

Most sample code demonstrated in our documentation and SDKs, use the **Global device endpoint** (`global.azure-devices-provisioning.net`) and **ID Scope** to resolve a particular DPS instance. Use the service endpoint in place of the global device endpoint when connecting to a DPS instance using private endpoints to provision your devices.

For example, the provisioning device client sample ([pro_dev_client_sample](#)) in the [Azure IoT C SDK](#) is designed to use the **Global device endpoint** as the global provisioning URI (`global_prov_uri`) in [prov_dev_client_sample.c](#)

```
MU_DEFINE_ENUM_STRINGS_WITHOUT_INVALID(PROV_DEVICE_RESULT, PROV_DEVICE_RESULT_VALUES);
MU_DEFINE_ENUM_STRINGS_WITHOUT_INVALID(PROV_DEVICE_REG_STATUS, PROV_DEVICE_REG_STATUS_VALUES);

static const char* global_prov_uri = "global.azure-devices-provisioning.net";
static const char* id_scope = "[ID Scope];
```

```
}
```

```
PROV_DEVICE_RESULT prov_device_result = PROV_DEVICE_RESULT_ERROR;
PROV_DEVICE_HANDLE prov_device_handle;
if ((prov_device_handle = Prov_Device_Create(global_prov_uri, id_scope, prov_transport)) == NULL)
{
    (void)printf("failed calling Prov_Device_Create\r\n");
```

To use the sample with a private endpoint, the highlighted code above would be changed to use the service

endpoint for your DPS resource. For example, if your service endpoint was `mydps.azure-devices-provisioning.net`, the code would look as follows.

```
static const char* global_prov_uri = "global.azure-devices-provisioning.net";
static const char* service_uri = "mydps.azure-devices-provisioning.net";
static const char* id_scope = "[ID Scope];
```

```
PROV_DEVICE_RESULT prov_device_result = PROV_DEVICE_RESULT_ERROR;
PROV_DEVICE_HANDLE prov_device_handle;
if ((prov_device_handle = Prov_Device_Create(service_uri, id_scope, prov_transport)) == NULL)
{
    (void)printf("failed calling Prov_Device_Create\r\n");
}
```

Request a private endpoint

You can request a private endpoint to a DPS instance by resource ID. In order to make this request, you need the resource owner to supply you with the resource ID.

1. The resource ID is provided on to the properties tab for DPS resource as shown below.

The screenshot shows the Azure portal interface for a DPS instance named 'DPS-test7'. The left sidebar has a 'Properties' tab highlighted with a red box. The main area displays various resource details: Name (DPS-test7), Resource type (Microsoft.Devices/ProvisioningServices), Location ID (eastus), Resource ID (/subscriptions/Your Azure Subscription ID/resourceGroups/DocRelatedTestingResources/providers/Microsoft.Devices/ProvisioningServices/DPS-test7), Resource group (DocRelatedTestingResources), Resource group ID (/subscriptions/Your Azure Subscription ID/resourceGroups/DocRelatedTestingResources), and Subscription (DocRelatedTesting). The Resource ID field is specifically highlighted with a red box.

Caution

Be aware that the resource ID does contain the subscription ID.

2. Once you have the resource ID, follow the steps above in [Set up a private endpoint](#) to step 3 on the *Create a private endpoint* Resource page. Click **Connect to an Azure resource by resource ID or alias** and enter the information in the following table.

FIELD	VALUE
Resource ID or alias	Enter the resource ID for the DPS resource.
Target sub-resource	Enter <code>iotDps</code>
Request message	Enter a request message for the DPS resource owner. For example, <code>Please approve this new private endpoint</code> <code>for IoT devices in site 23 to access this DPS instance</code>

Click **Next : Configuration** to configure the VNET for the private endpoint.

3. On the *Create a private endpoint Configuration* page, choose the virtual network and subnet to create the private endpoint in.
- Click **Next : Tags**, and optionally provide any tags for your resource.
4. Click **Review + create** and then **Create** to create your private endpoint request.
5. The DPS owner will see the private endpoint request in the **Private endpoint connections** list on DPS networking tab. On that page, the owner can **Approve** or **Reject** the private endpoint request as shown below.

The screenshot shows the 'Networking' blade for a DPS instance named 'DPS-test7'. The 'Private endpoint connections' tab is selected. A table lists two connections:

CONNECTION NAME	CONNECTION STATE	PRIVATE ENDPOINT	DESCRIPTION
DPS-test7.0d757774-c071...	Approved	dps-test7-private-endpoi...	Auto-Approved
DPS-test7.8e461870-ad42...	Pending	dps-test7-private-endpoi...	Please approve this new private endpoint for IoT devices in site 23 to access this DPS instance

Red boxes highlight several UI elements: the 'Private endpoint connections' tab, the 'Approve' and 'Reject' buttons at the top, and the checkbox column header in the table.

Pricing private endpoints

For pricing details, see [Azure Private Link pricing](#).

Next steps

Use the links below to learn more about DPS security features:

- [Security](#)
- [TLS 1.2 Support](#)

IoT Hub Device reprovisioning concepts

8/22/2022 • 6 minutes to read • [Edit Online](#)

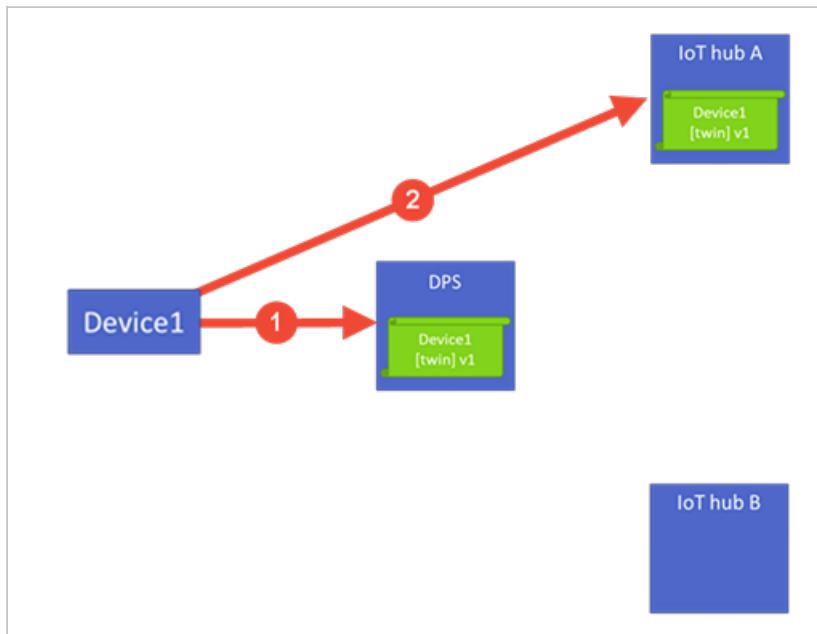
During the lifecycle of an IoT solution, it's common to move devices between IoT hubs. The reasons for this move may include the following scenarios:

- **Geolocation / GeoLatency:** As a device moves between locations, network latency is improved by having the device migrated to a closer IoT hub.
- **Multi-tenancy:** A device may be used within the same IoT solution and reassigned to a new customer, or customer site. This new customer may be serviced using a different IoT hub.
- **Solution change:** A device could be moved into a new or updated IoT solution. This reassignment may require the device to communicate with a new IoT hub that's connected to other back-end components.
- **Quarantine:** Similar to a solution change. A device that's malfunctioning, compromised, or out-of-date may be reassigned to an IoT hub that can only update and get back in compliance. Once the device is functioning properly, it's then migrated back to its main hub.

Reprovisioning support within the Device Provisioning Service addresses these needs. Devices can be automatically reassigned to new IoT hubs based on the reprovisioning policy that's configured on the device's enrollment entry.

Device state data

Device state data is composed of the [device twin](#) and device capabilities. This data is stored in the Device Provisioning Service instance and the IoT hub that a device is assigned to.

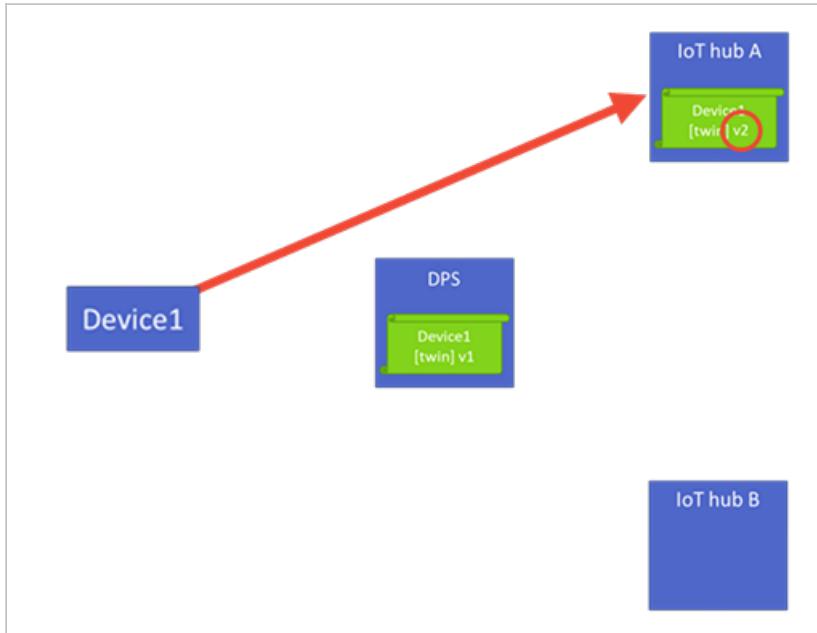


When a device is initially provisioned with a Device Provisioning Service instance, the following steps are done:

1. The device sends a provisioning request to a Device Provisioning Service instance. The service instance authenticates the device identity based on an enrollment entry, and creates the initial configuration of the device state data. The service instance assigns the device to an IoT hub based on the enrollment configuration and returns that IoT hub assignment to the device.

2. The provisioning service instance gives a copy of any initial device state data to the assigned IoT hub. The device connects to the assigned IoT hub and begins operations.

Over time, the device state data on the IoT hub may be updated by [device operations](#) and [back-end operations](#). The initial device state information stored in the Device Provisioning Service instance stays untouched. This untouched device state data is the initial configuration.

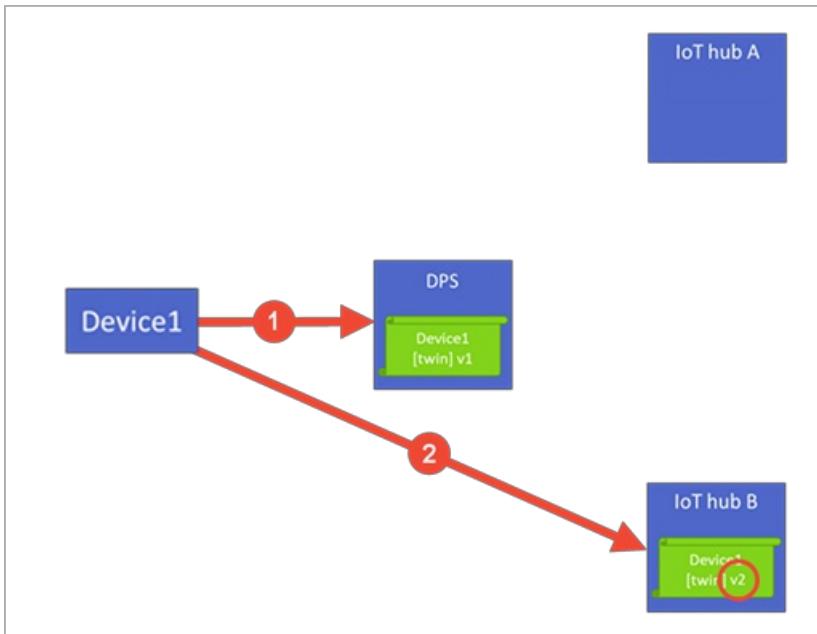


Depending on the scenario, as a device moves between IoT hubs, it may also be necessary to migrate device state updated on the previous IoT hub over to the new IoT hub. This migration is supported by reprovisioning policies in the Device Provisioning Service.

Reprovisioning policies

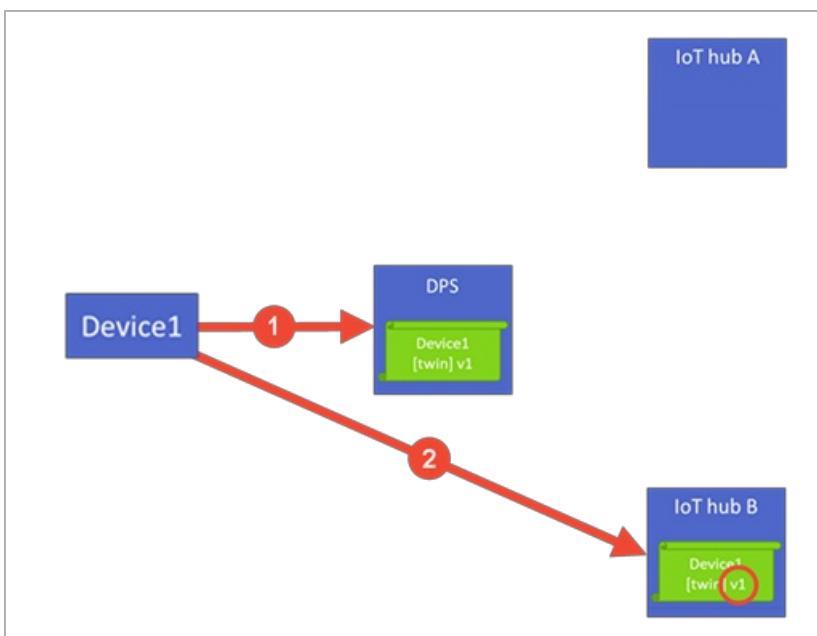
Depending on the scenario, a device could send a request to a provisioning service instance on reboot. It also supports a method to manually trigger provisioning on demand. The reprovisioning policy on an enrollment entry determines how the device provisioning service instance handles these provisioning requests. The policy also determines whether device state data should be migrated during reprovisioning. The same policies are available for individual enrollments and enrollment groups:

- **Re-provision and migrate data:** This policy is the default for new enrollment entries. This policy takes action when devices associated with the enrollment entry submit a new request (1). Depending on the enrollment entry configuration, the device may be reassigned to another IoT hub. If the device is changing IoT hubs, the device registration with the initial IoT hub will be removed. The updated device state information from that initial IoT hub will be migrated over to the new IoT hub (2). During migration, the device's status will be reported as **Assigning**.



- **Re-provision and reset to initial config:** This policy takes action when devices associated with the enrollment entry submit a new provisioning request (1). Depending on the enrollment entry configuration, the device may be reassigned to another IoT hub. If the device is changing IoT hubs, the device registration with the initial IoT hub will be removed. The initial configuration data that the provisioning service instance received when the device was provisioned is provided to the new IoT hub (2). During migration, the device's status will be reported as **Assigning**.

This policy is often used for a factory reset without changing IoT hubs.



- **Never re-provision:** The device is never reassigned to a different hub. This policy is provided for managing backwards compatibility.

NOTE

DPS will always call the custom allocation webhook regardless of re-provisioning policy in case there is new [ReturnData](#) for the device. If the re-provisioning policy is set to **never re-provision**, the webhook will be called but the device will not change its assigned hub.

When designing your solution and defining a reprovisioning logic there are a few things to consider. For example:

- How often you expect your devices to restart
- The [DPS quotas and limits](#)
- Expected deployment time for your fleet (phased rollout vs all at once)
- Retry capability implemented on your client code, as described on the [Retry general guidance](#) at the Azure Architecture Center

TIP

We recommend not provisioning on every reboot of the device, as this could cause some issues when reprovisioning several thousands or millions of devices at once. Instead you should attempt to use the [Device Registration Status Lookup API](#) and try to connect with that information to IoT Hub. If that fails, then try to reprovision as the IoT Hub information might have changed. Keep in mind that querying for the registration state will count as a new device registration, so you should consider the [Device registration limit](#). Also consider implementing an appropriate retry logic, such as exponential back-off with randomization, as described on the [Retry general guidance](#). In some cases, depending on the device capabilities, it's possible to save the IoT Hub information directly on the device to connect directly to IoT Hub after the first-time provisioning using DPS occurred. If you choose to do this, make sure you implement a fallback mechanism in case you get specific [errors from Hub occur](#), for example, consider the following scenarios:

- Retry the Hub operation if the result code is 429 (Too Many Requests) or an error in the 5xx range. Do not retry for any other errors.
- For 429 errors, only retry after the time indicated in the Retry-After header.
- For 5xx errors, use exponential back-off, with the first retry at least 5 seconds after the response.
- On errors other than 429 and 5xx, re-register through DPS
- Ideally you should also support a [method](#) to manually trigger provisioning on demand.

We also recommend taking into account the service limits when planning activities like pushing updates to your fleet. For example, updating the fleet all at once could cause all devices to re-register through DPS (which could easily be above the registration quota limit) - For such scenarios, consider planning for device updates in phases instead of updating your entire fleet at the same time.

Managing backwards compatibility

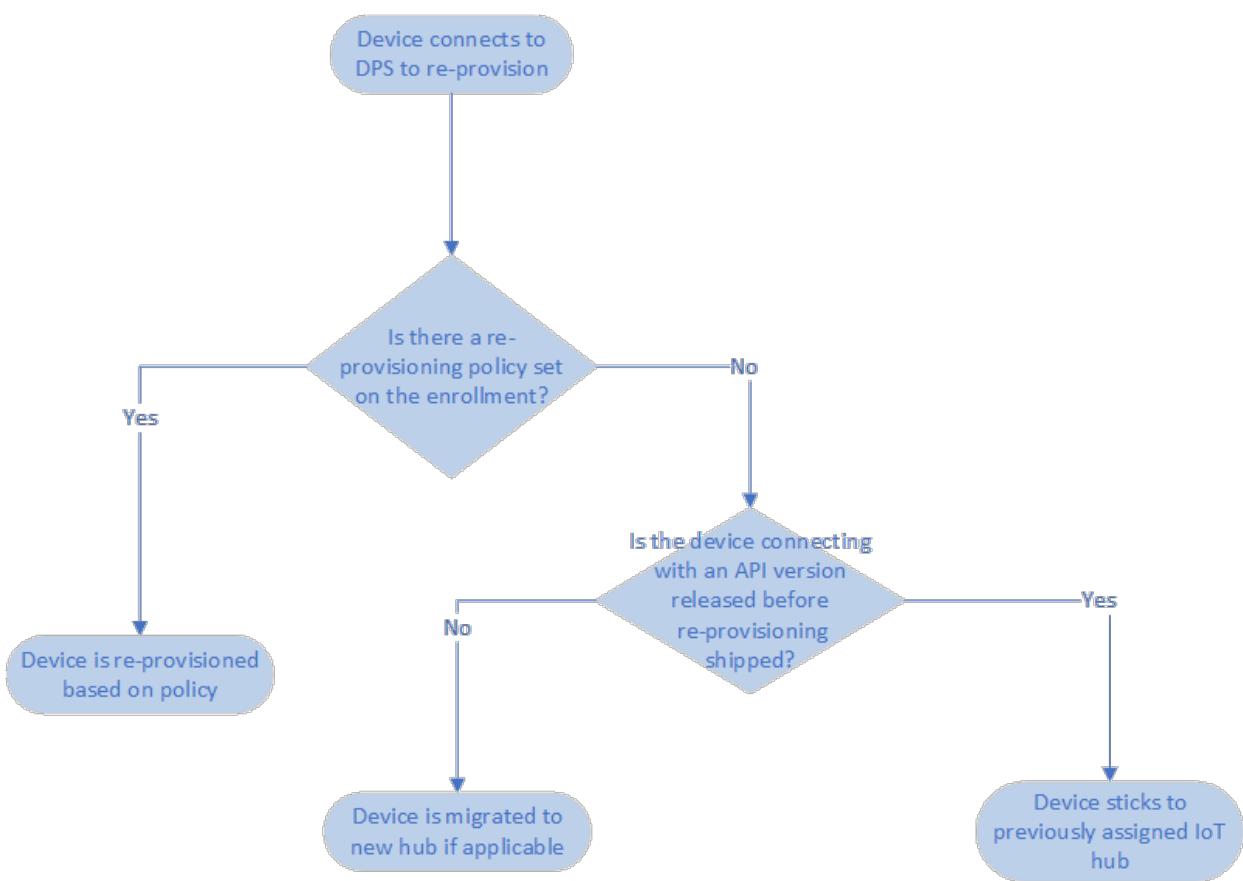
Before September 2018, device assignments to IoT hubs had a sticky behavior. When a device went back through the provisioning process, it would only be assigned back to the same IoT hub.

For solutions that have taken a dependency on this behavior, the provisioning service includes backwards compatibility. This behavior is presently maintained for devices according to the following criteria:

1. The devices connect with an API version before the availability of native reprovisioning support in the Device Provisioning Service. Refer to the API table below.
2. The enrollment entry for the devices doesn't have a reprovisioning policy set on them.

This compatibility makes sure that previously deployed devices experience the same behavior that's present during initial testing. To preserve the previous behavior, don't save a reprovisioning policy to these enrollments. If a reprovisioning policy is set, the reprovisioning policy takes precedence over the behavior. By allowing the reprovisioning policy to take precedence, customers can update device behavior without having to reimagine the device.

The following flow chart helps to show when the behavior is present:



The following table shows the API versions before the availability of native reprovisioning support in the Device Provisioning Service:

REST API	C SDK	PYTHON SDK	NODE SDK	JAVA SDK	.NET SDK
2018-04-01 and earlier	1.2.8 and earlier	1.4.2 and earlier	1.7.3 or earlier	1.13.0 or earlier	1.1.0 or earlier

NOTE

These values and links are likely to change. This is only a placeholder attempt to determine where the versions can be determined by a customer and what the expected versions will be.

Next steps

- [How to reprovision devices](#)

Best practices for large-scale IoT device deployments

8/22/2022 • 9 minutes to read • [Edit Online](#)

Scaling an IoT solution to millions of devices can be challenging. Large-scale solutions often need to be designed in accordance with service and subscription limits. When customers use Azure IoT Device Provisioning Service, they use it in combination with other Azure IoT platform services and components, such as IoT Hub and Azure IoT device SDKs. This article describes best practices, patterns, and sample code you can incorporate in your design to take advantage of these services and allow your deployments to scale out. By following these simple patterns and practices right from the design phase of the project, you can maximize the performance of your IoT devices.

First-time device provisioning

First-time provisioning is the process of onboarding a device for the first time as a part of an IoT solution. When working with large-scale deployments, it's important to schedule the provisioning process to avoid overload situations caused by all the devices attempting to connect at the same time.

Device deployment using a staggered provisioning schedule

For deployment of devices in the scale of millions, registering all the devices at once may result in the DPS instance being overwhelmed due to throttling (HTTP response code `429, Too Many Requests`) and a failure to register your devices. To prevent such throttling, you should use a staggered registration schedule for the devices. The recommended batch size should be in accordance with DPS [quotas and limits](#). For instance, if the registration rate is 200 devices per minute, the batch size for onboarding would be 200 devices per batch.

Timing logic when retrying operations

If transient faults occur due to a service being busy, a retry logic enables devices to successfully connect to the IoT cloud. However, a large number of retries could further degrade a busy service that's running close to or at its capacity. As with any Azure service, you should implement an intelligent retry mechanism with exponential backoff. More information on different retry patterns can be found in [the Retry design pattern](#) and [transient fault handling](#).

Rather than immediately retrying a deployment when throttled, you should wait until the time specified in the `retry-after` header. If there's no retry header available from the service, this algorithm can help achieve a smoother device onboarding experience:

```
min_retry_delay_msec = 1000
max_retry_delay_msec = (1.0 / <load>) * <T> * 1000
max_random_jitter_msec = max_retry_delay_msec
```

Where `<load>` is a configurable factor with values > 0 (indicates that the load will perform at an average of load time multiplied by the number of connections per second) and `<T>` is the absolute minimum time to cold boot the devices (calculated as $T = N / cps$ where `N` is the total number of devices and `cps` is the service limit for number of connections per second). In this case, devices should delay reconnecting for a random amount of time, between `min_retry_delay_msec` and `max_retry_delay_msec`.

For more information on the timing of retry operations, see [Retry timing](#).

Reprovisioning devices

Reprovisioning is the process where the device needs to be provisioned to an IoT Hub after having been successfully connected previously. There can be many reasons that result in a need for device to reconnect to an IoT Hub, such as:

- A device could reboot due to power outage, loss in network connectivity, geo-relocation, firmware updates, factory reset, or certificate key rotation.
- The IoT Hub instance could be unavailable due to an unplanned IoT Hub outage.

You shouldn't need to provision every time the device reboots. Most devices that are reprovisioned end up connected to the same IoT hub in most scenarios. Instead, the device should attempt to directly connect to its IoT hub using the information that was cached from a previous successful connection.

Devices that can store a connection string

If the devices have the ability to store the connection string to the previously provisioned and connected IoT Hub, use the same string to skip the entire reprovisioning process and directly connect to the IoT Hub. This reduces the latency in successfully connecting to the appropriate IoT Hub. There are two possible cases here:

- The IoT Hub to connect upon device reboot is the same as the previously connected IoT Hub.

The connection string retrieved from the cache should work fine and the device must attempt to reconnect to the same endpoint. No need for a fresh start for the provisioning process.

- The IoT Hub to connect upon device reboot is different from the previously connected IoT Hub.

The connection string stored in memory is inaccurate. Attempting to connect to the same endpoint won't be successful and so the retry mechanism for the IoT Hub connection is triggered. Once the threshold for the IoT Hub connection failure is reached, the retry mechanism automatically triggers a fresh start to the provisioning process.

Devices that can't store a connection string

In certain scenarios, devices don't have a large enough footprint or memory to accommodate caching of the connection string from a past successful IoT Hub connection. You can use the [Device Registration Status Lookup API](#) to retrieve the connection string from the previous time the device was provisioned and then attempt a connection to that IoT Hub. At every device reboot, that API needs to be invoked to get the device registration status. If data related to a previously connected IoT Hub was returned by the API call, you can connect to the same IoT Hub. If the API returns a null payload, then there's no previous connection available and the reprovisioning process through DPS is automatically triggered.

Reprovisioning sample

These code examples show a class for reading to and writing from the device cache, followed by code that attempts to reconnect a device to the IoT Hub if a connection string is found and reprovisioning through DPS if it isn't.

```

using Newtonsoft.Json;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;

namespace ProvisioningCache
{
    public class ProvisioningDetailsFileStorage : IProvisioningDetailCache
    {
        private string dataDirectory = null;

        public ProvisioningDetailsFileStorage()
        {
            dataDirectory = Environment.GetEnvironmentVariable("ProvisioningDetailsDataDirectory");
        }

        public ProvisioningResponse GetProvisioningDetailResponseFromCache(string registrationId)
        {
            try
            {
                var provisioningResponseFile = File.ReadAllText(Path.Combine(dataDirectory, registrationId));

                ProvisioningResponse response = JsonConvert.DeserializeObject<ProvisioningResponse>(provisioningResponseFile);

                return response;
            }
            catch (Exception ex)
            {
                return null;
            }
        }

        public void SetProvisioningDetailResponse(string registrationId, ProvisioningResponse provisioningDetails)
        {
            var provisioningDetailsJson = JsonConvert.SerializeObject(provisioningDetails);

            File.WriteAllText(Path.Combine(dataDirectory, registrationId), provisioningDetailsJson);
        }
    }
}

```

You could use code similar to the following to determine how to proceed with reconnecting a device after determining whether there's connection info in the cache:

```

IProvisioningDetailCache provisioningDetailCache = new ProvisioningDetailsFileStorage();

var provisioningDetails = provisioningDetailCache.GetProvisioningDetailResponseFromCache(registrationId);

// If no info is available in cache, go through DPS for provisioning
if(provisioningDetails == null)
{
    logger.LogInformation($"Initializing the device provisioning client...");
    using var transport = new ProvisioningTransportHandlerAmqp();
    ProvisioningDeviceClient provClient = ProvisioningDeviceClient.Create(dpsEndpoint, dpsScopeId, security,
transport);
    logger.LogInformation($"Initialized for registration Id {security.GetRegistrationID()}.");
    logger.LogInformation("Registering with the device provisioning service... ");

    // This method will attempt to retry in case of a transient fault
    DeviceRegistrationResult result = await registerDevice(provClient);
    provisioningDetails = new ProvisioningResponse() { iotHubHostName = result.AssignedHub, deviceId =
result.DeviceId };
    provisioningDetailCache.SetProvisioningDetailResponse(registrationId, provisioningDetails);
}

// If there was IoT Hub info from previous provisioning in the cache, try connecting to the IoT Hub directly
// If trying to connect to the IoT Hub returns status 429, make sure to retry operation honoring
//   the retry-after header
// If trying to connect to the IoT Hub returns a 500-series server error, have an exponential backoff with
//   at least 5 seconds of wait-time
// For all response codes 429 and 5xx, reprovision through DPS
// Ideally, you should also support a method to manually trigger provisioning on demand
if (provisioningDetails != null)
{
    logger.LogInformation($"Device {provisioningDetails.deviceId} registered to
{provisioningDetails.iotHubHostName}.");
    logger.LogInformation("Creating TPM authentication for IoT Hub...");
    IAuthenticationMethod auth = new DeviceAuthenticationWithTpm(provisioningDetails.deviceId, security);
    logger.LogInformation($"Testing the provisioned device with IoT Hub...");
    DeviceClient iotClient = DeviceClient.Create(provisioningDetails.iotHubHostName, auth,
TransportType.Amqp);
    logger.LogInformation($"Registering the Method Call back for Reprovisioning...");
    await iotClient.SetMethodHandlerAsync("Reprovision", reprovisionDirectMethodCallback, iotClient);

    // Now you should start a thread into this method and do your business while the DeviceClient is still
    connected
    await startBackgroundWork(iotClient);
    logger.LogInformation("Wait until closed...");

    // Wait until the app unloads or is cancelled
    var cts = new CancellationTokenSource();
    AssemblyLoadContext.Default.Unloading += (ctx) => cts.Cancel();
    Console.CancelKeyPress += (sender, cpe) => cts.Cancel();

    await WhenCancelled(cts.Token);
    await iotClient.CloseAsync();
    Console.WriteLine("Finished.");
}

```

IoT Hub connectivity considerations

- Any single IoT hub is limited to 1 million devices plus modules. If you plan to have more than a million devices, cap the number of devices to 1 million per hub and add hubs as needed when increasing the scale of your deployment. For more information, see [IoT Hub quotas](#).
- If you have plans for more than a million devices and you need to support them in a specific region (such as in an EU region for data residency requirements), you can [contact us](#) to ensure that the region you're deploying to has the capacity to support your current and future scale.

Recommended device logic when connecting to IoT Hub via DPS:

- On first boot, devices should go use the [DPS registration API](#) to register.
- On subsequent boots, devices should:
 - If possible, cache their provisioning details and connect using this information from this cache.
 - If they can't cache IoT hub connection information, use the [Device Registration Status Lookup API](#) to return connection information once registration has been done. This API call is a much lighter weight operation for DPS than a full device registration operation.
 - For devices in either case described above, devices should use the following logic in response to error codes when connecting:
 - When receiving any of the 500-series of server error responses, retry the connection using either cached credentials or the results of a Device Registration Status Lookup API call.
 - When receiving `401, Unauthorized` or `403, Forbidden` or `404, Not Found`, perform a full re-registration by calling the [DPS registration API](#).
- At any time, devices should be capable of responding to a user-initiated reprovisioning command.

Other IoT Hub scenarios when using DPS:

- IoT Hub failover: Devices should continue to work as connection information shouldn't change and logic is in place to retry the connection once the hub is available again.
- Change of IoT Hub: Assigning devices to a different IoT Hub should be done by using a [custom allocation policy](#).
- Retry IoT Hub connection: You shouldn't use an aggressive retry strategy, instead allowing a gap of at least a minute before a retry.
- IoT Hub partitions: If your device strategy leans heavily on telemetry, the number of device-to-cloud partitions should be increased.

Monitoring devices

An important part of the overall deployment is monitoring the solution end-to-end to make sure that the system is performing appropriately. There are several ways to monitor the health of a service for large-scale deployment of IoT devices. The following patterns have proven effective in monitoring the service:

- Create an application to query each enrollment group on a DPS instance, get the total devices registered to that group, and then aggregate the numbers from across various enrollment groups. This number provides an exact count of the devices that are currently registered via DPS and can be used to monitor the state of the service.
- Monitor device registrations over a specific period. For instance, monitor registration rates for a DPS instance over the prior five days. Note that this approach only provides an approximate figure and is also capped to a time period.

Next steps

- [Provision devices across load-balanced IoT Hubs](#)
- [Retry timing when retrying operations](#)

IoT Hub Device Provisioning Service high availability and disaster recovery

8/22/2022 • 2 minutes to read • [Edit Online](#)

Device Provisioning Service (DPS) is a helper service for IoT Hub that enables zero-touch device provisioning at-scale. DPS is an important part of your IoT solution. This article describes the High Availability (HA) and Disaster Recovery (DR) capabilities that DPS provides. To learn more about how to achieve HA-DR across your entire IoT solution, see [Disaster recovery and high availability for Azure applications](#). To learn about HA-DR in IoT Hub, see [IoT Hub high availability and disaster recovery](#).

High availability

DPS is a highly available service; for details, see the [SLA for Azure IoT Hub](#). The full [Azure SLA](#) explains the guaranteed availability of Azure as a whole.

DPS also supports [Availability Zones](#). An Availability Zone is a high-availability offering that protects your applications and data from datacenter failures. A region with Availability Zone support is comprised of a minimum of three zones supporting that region. Each zone provides one or more datacenters, each in a unique physical location with independent power, cooling, and networking. This provides replication and redundancy within the region. Availability Zone support for DPS is enabled automatically for DPS resources in the following Azure regions:

- Australia East
- Brazil South
- Canada Central
- Central US
- East US
- East US 2
- France Central
- Japan East
- North Europe
- UK South
- West Europe
- West US 2

You don't need to take any action to use availability zones in supported regions. Your DPS instances are AZ-enabled by default. It's recommended that you leverage Availability Zones by using regions where they are supported.

Disaster recovery and Microsoft-initiated failover

DPS leverages [paired regions](#) to enable automatic failover. Microsoft-initiated failover is exercised by Microsoft in rare situations when an entire region goes down to failover all the DPS instances from the affected region to its corresponding paired region. This process is a default option 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 user consent before the user's DPS instance is failed over.

The only users who are able to opt-out of this feature are those deploying to the Brazil South and Southeast Asia

(Singapore) regions.

NOTE

Azure IoT Hub Device Provisioning Service doesn't store or process customer data outside of the geography where you deploy the service instance. For more information, see [Cross-region replication in Azure](#).

Disable disaster recovery

By default, DPS provides automatic failover by replicating data to the [paired region](#) for a DPS instance. For some regions, you can avoid data replication outside of the region by disabling disaster recovery when creating a DPS instance. The following regions support this feature:

- **Brazil South**; paired region, South Central US.
- **Southeast Asia (Singapore)**; paired region, East Asia (Hong Kong).

To disable disaster recovery in supported regions, make sure that **Disaster recovery enabled** is unselected when you create your DPS instance:

The IoT Hub device provisioning service is a helper service for IoT Hub that enables zero-touch, just-in-time provisioning to the right IoT hub without requiring human intervention, allowing customers to provision millions of devices in a secure and scalable manner. [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 * DEVICEHUB_DEV1

Resource group * testdrdpsasia

Instance details

Name * testdrdpsasia

Region * Southeast Asia

Disaster Recovery
IoT Hub device provisioning services perform automatic failover in the event of a data center outage. Your data will be duplicated to East Asia. You cannot change this setting after the resource has been created. [Learn more](#)

Disaster recovery enabled (recommended)

⚠ Failover will not be available for this resource. This setting cannot be changed after you create this IoT hub device provisioning service.

You can also disable disaster recovery when you create a DPS instance using an [ARM template](#).

Failover capability will not be available if you disable disaster recovery for a DPS instance.

You can check whether disaster recovery is disabled from the **Overview** page of your DPS instance in Azure portal:

testdrdpsasia

Device Provisioning Service

Overview

Activity log

Access control (IAM)

Tags

Diagnose and solve problems

Settings

Search (Ctrl+ /)

Move

Delete

Refresh

View Cost | JSON View

Essentials

Resource group (move)	:	DEVICEHUB_DEV1
Status	:	Active
Location	:	Southeast Asia
Subscription (move)	:	DEVICEHUB_DEV1
Subscription ID	:	
Tags (edit)	:	Click here to add tags

Service endpoint : testdrdpsasia.azure-devices-provisioning.net

Global device endpoint : global.azure-devices-provisioning.net

ID Scope : 0ne004A298F

Pricing and scale tier : S1

Automatic failover enabled : No

Device Provisioning Service IP addresses

8/22/2022 • 2 minutes to read • [Edit Online](#)

The IP address prefixes for the public endpoints of an IoT Hub Device Provisioning Service (DPS) are published periodically under the [AzureIoTHub service tag](#). You may use these IP address prefixes to control connectivity between an IoT DPS instance and devices or network assets to implement a variety of network isolation goals:

GOAL	APPROACH
Ensure your devices and services communicate with DPS endpoints only	Use the AzureIoTHub service tag to discover DPS instances. Configure ALLOW rules on your devices' and services' firewall setting for those IP address prefixes accordingly. Configure rules to drop traffic to other destination IP addresses that you don't want devices or services to communicate with.
Ensure your DPS endpoint receives connections only from your devices and network assets	Use IoT DPS IP filter feature to create filter rules for the device and DPS service APIs. These filter rules can be used to allow connections only from your devices and network asset IP addresses (see limitations section).

Best practices

- When adding ALLOW rules in your devices' firewall configuration, it's best to provide specific [ports used by applicable protocols](#).
- The IP address prefixes of IoT DPS instances are subject to change. These changes are published periodically via service tags before taking effect. It's 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](#). The 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 DPS endpoints in a specific region. To account for datacenter disaster recovery, or [regional failover](#), ensure connectivity to IP prefixes of your DPS instance's geo-pair region is also enabled.
- Setting up firewall rules for a DPS instance may block off connectivity needed to run Azure CLI and PowerShell commands against it. To avoid these connectivity issues, you can add ALLOW rules for your clients' IP address prefixes to re-enable CLI or PowerShell clients to communicate with your DPS instance.

Limitations and workarounds

- The DPS IP filter feature has a limit of 100 rules.
- Your configured [IP filtering rules](#) are only applied on your DPS endpoints and not on the linked IoT Hub endpoints. IP filtering for linked IoT Hubs must be configured separately. For more information, see, [IoT Hub IP filtering rules](#).

Support for IPv6

IPv6 is currently not supported on IoT Hub or DPS.

Next steps

To learn more about IP address configurations with DPS, see:

- [Configure IP filtering](#)

TLS support in Azure IoT Hub Device Provisioning Service (DPS)

8/22/2022 • 2 minutes to read • [Edit Online](#)

DPS uses [Transport Layer Security \(TLS\)](#) to secure connections from IoT devices.

Current TLS protocol versions supported by DPS are:

- TLS 1.2

Restrict connections to TLS 1.2

For added security, it is advised to configure your DPS instances to *only* allow device client connections that use TLS version 1.2 and to enforce the use of [recommended ciphers](#).

To do this, provision a new DPS resource setting the `minTlsVersion` property to `1.2` in your Azure Resource Manager template's DPS resource specification. The following example template JSON specifies the `minTlsVersion` property for a new DPS instance.

```
{  
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",  
    "contentVersion": "1.0.0.0",  
    "resources": [  
        {  
            "type": "Microsoft.Devices/ProvisioningServices",  
            "apiVersion": "2020-01-01",  
            "name": "<provide-a-valid-DPS-resource-name>",  
            "location": "<any-region>",  
            "properties": {  
                "minTlsVersion": "1.2"  
            },  
            "sku": {  
                "name": "S1",  
                "capacity": 1  
            },  
        }  
    ]  
}
```

You can deploy the template with the following Azure CLI command.

```
az deployment group create -g <your resource group name> --template-file template.json
```

For more information on creating DPS resources with Resource Manager templates, see, [Set up DPS with an Azure Resource Manager template](#).

The DPS resource created using this configuration will refuse devices that attempt to connect using TLS versions 1.0 and 1.1. Similarly, the TLS handshake will be refused if the device client's HELLO message does not list any of the [recommended ciphers](#).

NOTE

The `minTlsVersion` property is read-only and cannot be changed once your DPS resource is created. It is therefore essential that you properly test and validate that *all* your IoT devices are compatible with TLS 1.2 and the [recommended ciphers](#) in advance.

NOTE

Upon failovers, the `minTlsVersion` property of your DPS will remain effective in the geo-paired region post-failover.

Recommended ciphers

DPS instances that are configured to accept only TLS 1.2 will also enforce the use of the following cipher suites:

RECOMMENDED TLS 1.2 CIPHER SUITES

```
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
```

Legacy cipher suites

These cipher suites are currently still supported by DPS but will be deprecated. Use the recommended cipher suites above if possible.

OPTION #1 (BETTER SECURITY)

```
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA_P384 (uses SHA-1)
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA_P256 (uses SHA-1)
TLS_RSA_WITH_AES_256_GCM_SHA384 (lack of Perfect Forward Secrecy)
TLS_RSA_WITH_AES_128_GCM_SHA256 (lack of Perfect Forward Secrecy)
TLS_RSA_WITH_AES_256_CBC_SHA256 (lack of Perfect Forward Secrecy)
TLS_RSA_WITH_AES_128_CBC_SHA256 (lack of Perfect Forward Secrecy)
TLS_RSA_WITH_AES_256_CBC_SHA (uses SHA-1, lack of Perfect Forward Secrecy)
TLS_RSA_WITH_AES_128_CBC_SHA (uses SHA-1, lack of Perfect Forward Secrecy)
```

OPTION #2 (BETTER PERFORMANCE)

```
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA_P256 (uses SHA-1)
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA_P384 (uses SHA-1)
TLS_RSA_WITH_AES_128_GCM_SHA256 (lack of Perfect Forward Secrecy)
TLS_RSA_WITH_AES_256_GCM_SHA384 (lack of Perfect Forward Secrecy)
TLS_RSA_WITH_AES_128_CBC_SHA256 (lack of Perfect Forward Secrecy)
TLS_RSA_WITH_AES_256_CBC_SHA256 (lack of Perfect Forward Secrecy)
TLS_RSA_WITH_AES_128_CBC_SHA (uses SHA-1, lack of Perfect Forward Secrecy)
TLS_RSA_WITH_AES_256_CBC_SHA (uses SHA-1, lack of Perfect Forward Secrecy)
```

Use TLS 1.2 in the IoT SDKs

Use the links below to configure TLS 1.2 and allowed ciphers in the Azure IoT 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 with IoT Hub

IoT Hub can be configured to use TLS 1.2 when communicating with devices. For more information, see [Deprecating TLS 1.0 and 1.1 for IoT Hub](#).

Use TLS 1.2 with IoT Edge

IoT Edge devices can be configured to use TLS 1.2 when communicating with IoT Hub and DPS. For more information, see the [IoT Edge documentation page](#).

Security practices for Azure IoT device manufacturers

8/22/2022 • 15 minutes to read • [Edit Online](#)

As more manufacturers release IoT devices, it's helpful to identify guidance around common practices. This article summarizes recommended security practices to consider when you manufacture devices for use with Azure IoT Device Provisioning Service (DPS).

- Selecting device authentication options
- Installing certificates on IoT devices
- Integrating a Trusted Platform Module (TPM) into the manufacturing process

Selecting device authentication options

The ultimate aim of any IoT device security measure is to create a secure IoT solution. But issues such as hardware limitations, cost, and level of security expertise all impact which options you choose. Further, your approach to security impacts how your IoT devices connect to the cloud. While there are [several elements of IoT security](#) to consider, a key element that every customer encounters is what authentication type to use.

Three widely used authentication types are X.509 certificates, Trusted Platform Modules (TPM), and symmetric keys. While other authentication types exist, most customers who build solutions on Azure IoT use one of these three types. The rest of this article surveys pros and cons of using each authentication type.

X.509 certificate

X.509 certificates are a type of digital identity you can use for authentication. The X.509 certificate standard is documented in [IETF RFC 5280](#). In Azure IoT, there are two ways to authenticate certificates:

- Thumbprint. A thumbprint algorithm is run on a certificate to generate a hexadecimal string. The generated string is a unique identifier for the certificate.
- CA authentication based on a full chain. A certificate chain is a hierarchical list of all certificates needed to authenticate an end-entity (EE) certificate. To authenticate an EE certificate, it's necessary to authenticate each certificate in the chain including a trusted root CA.

Pros for X.509:

- X.509 is the most secure authentication type supported in Azure IoT.
- X.509 allows a high level of control for purposes of certificate management.
- Many vendors are available to provide X.509 based authentication solutions.

Cons for X.509:

- Many customers may need to rely on external vendors for their certificates.
- Certificate management can be costly and adds to total solution cost.
- Certificate life-cycle management can be difficult if logistics are not well thought out.

Trusted Platform Module (TPM)

TPM, also known as [ISO/IEC 11889](#), is a standard for securely generating and storing cryptographic keys. TPM also refers to a virtual or physical I/O device that interacts with modules that implement the standard. A TPM device can exist as discrete hardware, integrated hardware, a firmware-based module, or a software-based module.

There are two key differences between TPMs and symmetric keys:

- TPM chips can also store X.509 certificates.
- TPM attestation in DPS uses the TPM endorsement key (EK), a form of asymmetric authentication. With asymmetric authentication, a public key is used for encryption, and a separate private key is used for decryption. In contrast, symmetric keys use symmetric authentication, where the private key is used for both encryption and decryption.

Pros for TPM:

- TPMs are included as standard hardware on many Windows devices, with built-in support for the operating system.
- TPM attestation is easier to secure than shared access signature (SAS) token-based symmetric key attestation.
- You can easily expire and renew, or roll, device credentials. DPS automatically rolls the IoT Hub credentials whenever a TPM device is due for reprovisioning.

Cons for TPM:

- TPMs are complex and can be difficult to use.
- Application development with TPMs is difficult unless you have a physical TPM or a quality emulator.
- You may have to redesign the board of your device to include a TPM in the hardware.
- If you roll the EK on a TPM, it destroys the identity of the TPM and creates a new one. Although the physical chip stays the same, it has a new identity in your IoT solution.

Symmetric key

With symmetric keys, the same key is used to encrypt and decrypt messages. As a result, the same key is known to both the device and the service that authenticates it. Azure IoT supports SAS token-based symmetric key connections. Symmetric key authentication requires significant owner responsibility to secure the keys and achieve an equal level of security with X.509 authentication. If you use symmetric keys, the recommended practice is to protect the keys by using a hardware security module (HSM).

Pros for symmetric key:

- Using symmetric keys is the simplest, lowest cost way to get started with authentication.
- Using symmetric keys streamlines your process because there's nothing extra to generate.

Cons for symmetric key:

- Symmetric keys take a significant degree of effort to secure the keys. The same key is shared between device and cloud, which means the key must be protected in two places. In contrast, the challenge with TPM and X.509 certificates is proving possession of the public key without revealing the private key.
- Symmetric keys make it easy to follow poor security practices. A common tendency with symmetric keys is to hard code the unencrypted keys on devices. While this practice is convenient, it leaves the keys vulnerable. You can mitigate some risk by securely storing the symmetric key on the device. However, if your priority is ultimately security rather than convenience, use X.509 certificates or TPM for authentication.

Shared symmetric key

There's a variation of symmetric key authentication known as shared symmetric key. This approach involves using the same symmetric key in all devices. The recommendation is to avoid using shared symmetric keys on your devices.

Pro for shared symmetric key:

- Simple to implement and inexpensive to produce at scale.

Cons for shared symmetric key:

- Highly vulnerable to attack. The benefit of easy implementation is far outweighed by the risk.
- Anyone can impersonate your devices if they obtain the shared key.
- If you rely on a shared symmetric key that becomes compromised, you will likely lose control of the devices.

Making the right choice for your devices

To choose an authentication method, make sure you consider the benefits and costs of each approach for your unique manufacturing process. For device authentication, usually there's an inverse relationship between how secure a given approach is, and how convenient it is.

Installing certificates on IoT devices

If you use X.509 certificates to authenticate your IoT devices, this section offers guidance on how to integrate certificates into your manufacturing process. You'll need to make several decisions. These include decisions about common certificate variables, when to generate certificates, and when to install them.

If you're used to using passwords, you might ask why you can't use the same certificate in all your devices, in the same way that you'd be able to use the same password in all your devices. First, using the same password everywhere is dangerous. The practice has exposed companies to major DDoS attacks, including the one that took down DNS on the US East Coast several years ago. Never use the same password everywhere, even with personal accounts. Second, a certificate isn't a password, it's a unique identity. A password is like a secret code that anyone can use to open a door at a secured building. It's something you know, and you could give the password to anyone to gain entrance. A certificate is like a driver's license with your photo and other details, which you can show to a guard to get into a secured building. It's tied to who you are. Provided that the guard accurately matches people with driver's licenses, only you can use your license (identity) to gain entrance.

Variables involved in certificate decisions

Consider the following variables, and how each one impacts the overall manufacturing process.

Where the certificate root of trust comes from

It can be costly and complex to manage a public key infrastructure (PKI). Especially if your company doesn't have any experience managing a PKI. Your options are:

- Use a third-party PKI. You can buy intermediate signing certificates from a third-party certificate vendor. Or you can use a private Certificate Authority (CA).
- Use a self-managed PKI. You can maintain your own PKI system and generate your own certificates.
- Use the [Azure Sphere](#) security service. This option applies only to Azure Sphere devices.

Where certificates are stored

There are a few factors that impact the decision on where certificates are stored. These factors include the type of device, expected profit margins (whether you can afford secure storage), device capabilities, and existing security technology on the device that you may be able to use. Consider the following options:

- In a hardware security module (HSM). Using an HSM is highly recommended. Check whether your device's control board already has an HSM installed. If you know you don't have an HSM, work with your hardware manufacturer to identify an HSM that meets your needs.
- In a secure place on disk such as a trusted execution environment (TEE).
- In the local file system or a certificate store. For example, the Windows certificate store.

Connectivity at the factory

Connectivity at the factory determines how and when you'll get the certificates to install on the devices.

Connectivity options are as follows:

- Connectivity. Having connectivity is optimal, it streamlines the process of generating certificates locally.

- No connectivity. In this case, you use a signed certificate from a CA to generate device certificates locally and offline.
- No connectivity. In this case, you can obtain certificates that were generated ahead of time. Or you can use an offline PKI to generate certificates locally.

Audit requirement

Depending on the type of devices you produce, you might have a regulatory requirement to create an audit trail of how device identities are installed on your devices. Auditing adds significant production cost. So in most cases, only do it if necessary. If you're unsure whether an audit is required, check with your company's legal department. Auditing options are:

- Not a sensitive industry. No auditing is required.
- Sensitive industry. Certificates should be installed in a secure room according to compliance certification requirements. If you need a secure room to install certificates, you are likely already aware of how certificates get installed in your devices. And you probably already have an audit system in place.

Length of certificate validity

Like a driver's license, certificates have an expiration date that is set when they are created. Here are the options for length of certificate validity:

- Renewal not required. This approach uses a long renewal period, so you'll never need to renew the certificate during the device's lifetime. While such an approach is convenient, it's also risky. You can reduce the risk by using secure storage like an HSM on your devices. However, the recommended practice is to avoid using long-lived certificates.
- Renewal required. You'll need to renew the certificate during the lifetime of the device. The length of the certificate validity depends on context, and you'll need a strategy for renewal. The strategy should include where you're getting certificates, and what type of over-the-air functionality your devices have to use in the renewal process.

When to generate certificates

The internet connectivity capabilities at your factory will impact your process for generating certificates. You have several options for when to generate certificates:

- Pre-loaded certificates. Some HSM vendors offer a premium service in which the HSM vendor installs certificates for the customer. First, customers give the HSM vendor access to a signing certificate. Then the HSM vendor installs certificates signed by that signing certificate onto each HSM the customer buys. All the customer has to do is install the HSM on the device. While this service adds cost, it helps to streamline your manufacturing process. And it resolves the question of when to install certificates.
- Device-generated certificates. If your devices generate certificates internally, then you must extract the public X.509 certificate from the device to enroll it in DPS.
- Connected factory. If your factory has connectivity, you can generate device certificates whenever you need them.
- Offline factory with your own PKI. If your factory does not have connectivity, and you are using your own PKI with offline support, you can generate the certificates when you need them.
- Offline factory with third-party PKI. If your factory does not have connectivity, and you are using a third-party PKI, you must generate the certificates ahead of time. And it will be necessary to generate the certificates from a location that has connectivity.

When to install certificates

After you generate certificates for your IoT devices, you can install them in the devices.

If you use pre-loaded certificates with an HSM, the process is simplified. After the HSM is installed in the device, the device code can access it. Then you'll call the HSM APIs to access the certificate that's stored in the HSM. This approach is the most convenient for your manufacturing process.

If you don't use a pre-loaded certificate, you must install the certificate as part of your production process. The simplest approach is to install the certificate in the HSM at the same time that you flash the initial firmware image. Your process must add a step to install the image on each device. After this step, you can run final quality checks and any other steps, before you package and ship the device.

There are software tools available that let you run the installation process and final quality check in a single step. You can modify these tools to generate a certificate, or to pull a certificate from a pre-generated certificate store. Then the software can install the certificate where you need to install it. Software tools of this type enable you to run production quality manufacturing at scale.

After you have certificates installed on your devices, the next step is to learn how to enroll the devices with [DPS](#).

Integrating a TPM into the manufacturing process

If you use a TPM to authenticate your IoT devices, this section offers guidance. The guidance covers the widely used TPM 2.0 devices that have hash-based message authentication code (HMAC) key support. The TPM specification for TPM chips is an ISO standard that's maintained by the Trusted Computing Group. For more on TPM, see the specifications for [TPM 2.0](#) and [ISO/IEC 11889](#).

Taking ownership of the TPM

A critical step in manufacturing a device with a TPM chip is to take ownership of the TPM. This step is required so that you can provide a key to the device owner. The first step is to extract the endorsement key (EK) from the device. The next step is to actually claim ownership. How you accomplish this depends on which TPM and operating system you use. If needed, contact the TPM manufacturer or the developer of the device operating system to determine how to claim ownership.

In your manufacturing process, you can extract the EK and claim ownership at different times, which adds flexibility. Many manufacturers take advantage of this flexibility by adding a hardware security module (HSM) to enhance the security of their devices. This section provides guidance on when to extract the EK, when to claim ownership of the TPM, and considerations for integrating these steps into a manufacturing timeline.

IMPORTANT

The following guidance assumes you use a discrete, firmware, or integrated TPM. In places where it's applicable, the guidance adds notes on using a non-discrete or software TPM. If you use a software TPM, there may be additional steps that this guidance doesn't include. Software TPMs have a variety of implementations that are beyond the scope of this article. In general, it's possible to integrate a software TPM into the following general manufacturing timeline. However, while a software emulated TPM is suitable for prototyping and testing, it can't provide the same level of security as a discrete, firmware, or integrated TPM. As a general practice, avoid using a software TPM in production.

General manufacturing timeline

The following timeline shows how a TPM goes through a production process and ends up in a device. Each manufacturing process is unique, and this timeline shows the most common patterns. The timeline offers guidance on when to take certain actions with the keys.

Step 1: TPM is manufactured

- If you buy TPMs from a manufacturer for use in your devices, see if they'll extract public endorsement keys (EK_pubs) for you. It's helpful if the manufacturer provides the list of EK_pubs with the shipped devices.

NOTE

You could give the TPM manufacturer write access to your enrollment list by using shared access policies in your provisioning service. This approach lets them add the TPMs to your enrollment list for you. But that is early in the manufacturing process, and it requires trust in the TPM manufacturer. Do so at your own risk.

- If you manufacture TPMs to sell to device manufacturers, consider giving your customers a list of EK_pubs along with their physical TPMs. Providing customers with EK_pubs saves a step in their process.
- If you manufacture TPMs to use with your own devices, identify which point in your process is the most convenient to extract the EK_pub. You can extract the EK_pub at any of the remaining points in the timeline.

Step 2: TPM is installed into a device

At this point in the production process, you should know which DPS instance the device will be used with. As a result, you can add devices to the enrollment list for automated provisioning. For more information about automatic device provisioning, see the [DPS documentation](#).

- If you haven't extracted the EK_pub, now is a good time to do so.
- Depending on the installation process of the TPM, this step is also a good time to take ownership of the TPM.

Step 3: Device has firmware and software installed

At this point in the process, install the DPS client along with the ID scope and global URL for provisioning.

- Now is the last chance to extract the EK_pub. If a third party will install the software on your device, it's a good idea to extract the EK_pub first.
- This point in the manufacturing process is ideal to take ownership of the TPM.

NOTE

If you're using a software TPM, you can install it now. Extract the EK_pub at the same time.

Step 4: Device is packaged and sent to the warehouse

A device can sometimes sit in a warehouse for up to a year before being deployed and provisioned with DPS. If a device sits in a warehouse for a long time before deployment, customers who deploy the device might need to update the firmware, software, or expired credentials.

Step 5: Device is installed into the location

After the device arrives at its final location, it goes through automated provisioning with DPS.

For more information, see [provisioning](#) and [TPM attestation](#).

Resources

In addition to the recommended security practices in this article, Azure IoT provides resources to help with selecting secure hardware and creating secure IoT deployments:

- Azure IoT [security recommendations](#) to guide the deployment process.
- The [Microsoft Defender for Cloud](#) offers a service to help create secure IoT deployments.
- For help with evaluating your hardware environment, see the whitepaper [Evaluating your IoT Security](#).
- For help with selecting secure hardware, see [The Right Secure Hardware for your IoT Deployment](#).

How to transfer payloads between devices and DPS

8/22/2022 • 2 minutes to read • [Edit Online](#)

Sometimes DPS needs more data from devices to properly provision them to the right IoT Hub, and that data needs to be provided by the device. Vice versa, DPS can return data to the device to facilitate client-side logic.

When to use it

This feature can be used as an enhancement for [custom allocation](#). For example, you want to allocate your devices based on the device model without human intervention. In this case, you can configure the device to report its model information as part of the [register device call](#). DPS will pass the device's payload to the custom allocation webhook. Then your function can decide which IoT hub the device will be provisioned to based on the device model information. If needed, the webhook can return data back to the device as a JSON object in the webhook response.

Device sends data payload to DPS

When your device is sending a [register device call](#) to DPS, The register call can be enhanced to take other fields in the body. The body looks like the following:

```
{  
    "registrationId": "mydevice",  
    "tpm": {  
        "endorsementKey": "stuff",  
        "storageRootKey": "things"  
    },  
    "payload": { A JSON object that contains your additional data }  
}
```

DPS returns data to the device

If the custom allocation policy webhook wishes to return some data to the device, it will pass the data back as a JSON object in the webhook response. The change is in the payload section below.

```
{  
    "iotHubHostName": "sample-iot-hub-1.azure-devices.net",  
    "initialTwin": {  
        "tags": {  
            "tag1": true  
        },  
        "properties": {  
            "desired": {  
                "prop1": true  
            }  
        }  
    },  
    "payload": { A JSON object that contains the data returned by the webhook }  
}
```

SDK support

This feature is available in C, C#, JAVA and Nodejs client SDKs. To learn more about the Azure IoT SDKs available

for IoT Hub and the IoT Hub Device Provisioning service, see [Microsoft Azure IoT SDKs](#).

IoT Plug and Play (PnP) devices use the payload to send their model ID when they register with DPS. You can find examples of this usage in the PnP samples in the SDK or sample repositories. For example, [C# PnP thermostat](#) or [Node.js PnP temperature controller](#).

Next steps

- To learn how to provision devices using a custom allocation policy, see [How to use custom allocation policies](#)

How to use custom allocation policies

8/22/2022 • 19 minutes to read • [Edit Online](#)

A custom allocation policy gives you more control over how devices are assigned to an IoT hub. This is accomplished by using custom code in an [Azure Function](#) to assign devices to an IoT hub. The device provisioning service calls your Azure Function code providing all relevant information about the device and the enrollment. Your function code is executed and returns the IoT hub information used to provisioning the device.

By using custom allocation policies, you define your own allocation policies when the policies provided by the Device Provisioning Service don't meet the requirements of your scenario.

For example, maybe you want to examine the certificate a device is using during provisioning and assign the device to an IoT hub based on a certificate property. Or, maybe you have information stored in a database for your devices and need to query the database to determine which IoT hub a device should be assigned to.

This article demonstrates a custom allocation policy using an Azure Function written in C#. Two new IoT hubs are created representing a *Contoso Toasters Division* and a *Contoso Heat Pumps Division*. Devices requesting provisioning must have a registration ID with one of the following suffixes to be accepted for provisioning:

- **-contoso-tstrsd-007:** Contoso Toasters Division
- **-contoso-hpsd-088:** Contoso Heat Pumps Division

The devices will be provisioned based on one of these required suffixes on the registration ID. These devices will be simulated using a provisioning sample included in the [Azure IoT C SDK](#).

You perform the following steps in this article:

- Use the Azure CLI to create two Contoso division IoT hubs ([Contoso Toasters Division](#) and [Contoso Heat Pumps Division](#))
- Create a new group enrollment using an Azure Function for the custom allocation policy
- Create device keys for two device simulations.
- Set up the development environment for the Azure IoT C SDK
- Simulate the devices and verify that they are provisioned according to the example code in the custom allocation policy

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

Prerequisites

The following prerequisites are for a Windows development environment. For Linux or macOS, see the appropriate section in [Prepare your development environment](#) in the SDK documentation.

- [Visual Studio 2019](#) with the '[Desktop development with C++](#)' workload enabled. Visual Studio 2015 and Visual Studio 2017 are also supported.
- Latest version of [Git](#) installed.
- Use the Bash environment in [Azure Cloud Shell](#). For more information, see [Azure Cloud Shell Quickstart - Bash](#).
- If you prefer to run CLI reference commands locally, [install](#) the Azure CLI. If you're running on Windows



[Launch Cloud Shell](#)

or macOS, consider running Azure CLI in a Docker container. For more information, see [How to run the Azure CLI in a Docker container](#).

- If you're using a local installation, sign in to the Azure CLI by using the `az login` command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see [Sign in with the Azure CLI](#).
- When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see [Use extensions with the Azure CLI](#).
- Run `az version` to find the version and dependent libraries that are installed. To upgrade to the latest version, run `az upgrade`.

Create the provisioning service and two divisional IoT hubs

In this section, you use the Azure Cloud Shell to create a provisioning service and two IoT hubs representing the **Contoso Toasters Division** and the **Contoso Heat Pumps division**.

TIP

The commands used in this article create the provisioning service and other resources in the West US location. We recommend that you create your resources in the region nearest you that supports Device Provisioning Service. You can view a list of available locations by running the command

```
az provider show --namespace Microsoft.Devices --query "resourceTypes[?resourceType=='ProvisioningServices'].locations | [0]" --out table
```

or by going to the [Azure Status](#) page and searching for "Device Provisioning Service". In commands, locations can be specified either in one word or multi-word format; for example: westus, West US, WEST US, etc. The value is not case sensitive. If you use multi-word format to specify location, enclose the value in quotes; for example,

```
-- location "West US".
```

1. Use the Azure Cloud Shell to create a resource group with the `az group create` command. An Azure resource group is a logical container into which Azure resources are deployed and managed.

The following example creates a resource group named *contoso-us-resource-group* in the *westus* region. It is recommended that you use this group for all resources created in this article. This approach will make clean up easier after you're finished.

```
az group create --name contoso-us-resource-group --location westus
```

2. Use the Azure Cloud Shell to create a device provisioning service (DPS) with the `az iot dps create` command. The provisioning service will be added to *contoso-us-resource-group*.

The following example creates a provisioning service named *contoso-provisioning-service-1098* in the *westus* location. You must use a unique service name. Make up your own suffix in the service name in place of **1098**.

```
az iot dps create --name contoso-provisioning-service-1098 --resource-group contoso-us-resource-group --location westus
```

This command may take a few minutes to complete.

3. Use the Azure Cloud Shell to create the **Contoso Toasters Division** IoT hub with the `az iot hub create` command. The IoT hub will be added to *contoso-us-resource-group*.

The following example creates an IoT hub named *contoso-toasters-hub-1098* in the *westus* location. You

must use a unique hub name. Make up your own suffix in the hub name in place of **1098**.

Caution

The example Azure Function code for the custom allocation policy requires the substring **-toasters-** in the hub name. Make sure to use a name containing the required **toasters** substring.

```
az iot hub create --name contoso-toasters-hub-1098 --resource-group contoso-us-resource-group --  
location westus --sku S1
```

This command may take a few minutes to complete.

4. Use the Azure Cloud Shell to create the **Contoso Heat Pumps Division** IoT hub with the [az iot hub create](#) command. This IoT hub will also be added to *contoso-us-resource-group*.

The following example creates an IoT hub named *contoso-heatpumps-hub-1098* in the *westus* location. You must use a unique hub name. Make up your own suffix in the hub name in place of **1098**.

Caution

The example Azure Function code for the custom allocation policy requires the substring **-heatpumps-** in the hub name. Make sure to use a name containing the required **heatpumps** substring.

```
az iot hub create --name contoso-heatpumps-hub-1098 --resource-group contoso-us-resource-group --  
location westus --sku S1
```

This command may take a few minutes to complete.

5. The IoT hubs must be linked to the DPS resource.

Run the following two commands to get the connection strings for the hubs you just created. Replace the hub resource names with the names you chose in each command:

```
hubToastersConnectionString=$(az iot hub connection-string show --hub-name contoso-toasters-hub-1098  
--key primary --query connectionString -o tsv)  
hubHeatpumpsConnectionString=$(az iot hub connection-string show --hub-name contoso-heatpumps-hub-  
1098 --key primary --query connectionString -o tsv)
```

Run the following commands to link the hubs to the DPS resource. Replace the DPS resource name with the name you chose in each command:

```
az iot dps linked-hub create --dps-name contoso-provisioning-service-1098 --resource-group contoso-  
us-resource-group --connection-string $hubToastersConnectionString --location westus  
az iot dps linked-hub create --dps-name contoso-provisioning-service-1098 --resource-group contoso-  
us-resource-group --connection-string $hubHeatpumpsConnectionString --location westus
```

Create the custom allocation function

In this section, you create an Azure function that implements your custom allocation policy. This function decides which divisional IoT hub a device should be registered to based on whether its registration ID contains the string **-contoso-tstrsd-007** or **-contoso-hpsd-088**. It also sets the initial state of the device twin based on whether the device is a toaster or a heat pump.

1. Sign in to the [Azure portal](#). From your home page, select **+ Create a resource**.
2. In the *Search the Marketplace* search box, type "Function App". From the drop-down list select **Function App**, and then select **Create**.
3. On **Function App** create page, under the **Basics** tab, enter the following settings for your new function

app and select **Review + create**:

Resource Group: Select the **contoso-us-resource-group** to keep all resources created in this article together.

Function App name: Enter a unique function app name. This example uses **contoso-function-app-1098**.

Publish: Verify that **Code** is selected.

Runtime Stack: Select **.NET Core** from the drop-down.

Version: Select **3.1** from the drop-down.

Region: Select the same region as your resource group. This example uses **West US**.

NOTE

By default, Application Insights is enabled. Application Insights is not necessary for this article, but it might help you understand and investigate any issues you encounter with the custom allocation. If you prefer, you can disable Application Insights by selecting the **Monitoring** tab and then selecting **No** for **Enable Application Insights**.

Home > New > Marketplace >

Create Function App

Basics Hosting Monitoring Tags Review + create

Create a function app, which lets you group functions as a logical unit for easier management, deployment and sharing of resources. Functions lets you execute your code in a serverless environment without having to first create a VM or publish a web application.

Project Details

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

Subscription * ⓘ <Choose Your Subscription> ▾

Resource Group * ⓘ contoso-us-resource-group ▾ Create new

Instance Details

Function App name * contoso-function-app-1098 .azurewebsites.net

Publish * Code Docker Container

Runtime stack * .NET Core

Version * 3.1

Region * West US

Review + create < Previous Next : Hosting >

- On the **Summary** page, select **Create** to create the function app. Deployment may take several minutes. When it completes, select **Go to resource**.

5. On the left pane of the function app **Overview** page, click **Functions** and then + **Add** to add a new function.
6. On the **Add function** page, click **HTTP Trigger**, then click the **Add** button.
7. On the next page, click **Code + Test**. This allows you to edit the code for the function named **HttpTrigger1**. The **run.csx** code file should be opened for editing.
8. Reference required NuGet packages. To create the initial device twin, the custom allocation function uses classes that are defined in two NuGet packages that must be loaded into the hosting environment. With Azure Functions, NuGet packages are referenced using a *function.proj* file. In this step, you save and upload a *function.proj* file for the required assemblies. For more information, see [Using NuGet packages with Azure Functions](#).
 - a. Copy the following lines into your favorite editor and save the file on your computer as *function.proj*.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.Azure.Devices.Provisioning.Service" Version="1.16.3" />
    <PackageReference Include="Microsoft.Azure.Devices.Shared" Version="1.27.0" />
  </ItemGroup>
</Project>
```

- b. Click the **Upload** button located above the code editor to upload your *function.proj* file. After uploading, select the file in the code editor using the drop down box to verify the contents.
9. Make sure *run.csx* for **HttpTrigger1** is selected in the code editor. Replace the code for the **HttpTrigger1** function with the following code and select **Save**:

```
#r "Newtonsoft.Json"

using System.Net;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Primitives;
using Newtonsoft.Json;

using Microsoft.Azure.Devices.Shared; // For TwinCollection
using Microsoft.Azure.Devices.Provisioning.Service; // For TwinState

public static async Task<IActionResult> Run(HttpContext req, ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a request.");

    // Get request body
    string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
    dynamic data = JsonConvert.DeserializeObject(requestBody);

    log.LogInformation("Request.Body....");
    log.LogInformation(requestBody);

    // Get registration ID of the device
    string regId = data?.deviceRuntimeContext?.registrationId;

    string message = "Uncaught error";
    bool fail = false;
    ResponseObj obj = new ResponseObj();

    if (regId == null)
```

```

    {
        message = "Registration ID not provided for the device.";
        log.LogInformation("Registration ID : NULL");
        fail = true;
    }
    else
    {
        string[] hubs = data?.linkedHubs?.ToObject<string[]>();

        // Must have hubs selected on the enrollment
        if (hubs == null)
        {
            message = "No hub group defined for the enrollment.";
            log.LogInformation("linkedHubs : NULL");
            fail = true;
        }
        else
        {
            // This is a Contoso Toaster Model 007
            if (regId.Contains("-contoso-tstrsd-007"))
            {
                //Find the "-toasters-" IoT hub configured on the enrollment
                foreach(string hubString in hubs)
                {
                    if (hubString.Contains("-toasters-"))
                        obj.iotHubHostName = hubString;
                }

                if (obj.iotHubHostName == null)
                {
                    message = "No toasters hub found for the enrollment.";
                    log.LogInformation(message);
                    fail = true;
                }
                else
                {
                    // Specify the initial tags for the device.
                    TwinCollection tags = new TwinCollection();
                    tags["deviceType"] = "toaster";

                    // Specify the initial desired properties for the device.
                    TwinCollection properties = new TwinCollection();
                    properties["state"] = "ready";
                    properties["darknessSetting"] = "medium";

                    // Add the initial twin state to the response.
                    TwinState twinState = new TwinState(tags, properties);
                    obj.initialTwin = twinState;
                }
            }
            // This is a Contoso Heat pump Model 008
            else if (regId.Contains("-contoso-hpsd-008"))
            {
                //Find the "-heatpumps-" IoT hub configured on the enrollment
                foreach(string hubString in hubs)
                {
                    if (hubString.Contains("-heatpumps-"))
                        obj.iotHubHostName = hubString;
                }

                if (obj.iotHubHostName == null)
                {
                    message = "No heat pumps hub found for the enrollment.";
                    log.LogInformation(message);
                    fail = true;
                }
                else
                {
                    // Specify the initial tags for the device.

```

```

        // Specify the initial tags for the device.
        TwinCollection tags = new TwinCollection();
        tags["deviceType"] = "heatpump";

        // Specify the initial desired properties for the device.
        TwinCollection properties = new TwinCollection();
        properties["state"] = "on";
        properties["temperatureSetting"] = "65";

        // Add the initial twin state to the response.
        TwinState twinState = new TwinState(tags, properties);
        obj.initialTwin = twinState;
    }
}

// Unrecognized device.
else
{
    fail = true;
    message = "Unrecognized device registration.";
    log.LogInformation("Unknown device registration");
}
}

log.LogInformation("\nResponse");
log.LogInformation((obj.iotHubHostName != null) ? JsonConvert.SerializeObject(obj) : message);

return (fail)
    ? new BadRequestObjectResult(message)
    : (ActionResult)new OkObjectResult(obj);
}

public class ResponseObj
{
    public string iotHubHostName {get; set;}
    public TwinState initialTwin {get; set;}
}

```

Create the enrollment

In this section, you'll create a new enrollment group that uses the custom allocation policy. For simplicity, this article uses [Symmetric key attestation](#) with the enrollment. For a more secure solution, consider using [X.509 certificate attestation](#) with a chain of trust.

1. Still on the [Azure portal](#), open your provisioning service.
2. Select **Manage enrollments** on the left pane, and then select the **Add enrollment group** button at the top of the page.
3. On **Add Enrollment Group**, enter the following information, and select the **Save** button.

Group name: Enter **contoso-custom-allocated-devices**.

Attestation Type: Select **Symmetric Key**.

Auto Generate Keys: This checkbox should already be checked.

Select how you want to assign devices to hubs: Select **Custom (Use Azure Function)**.

Subscription: Select the subscription where you created your Azure Function.

Function App: Select your function app by name. **contoso-function-app-1098** was used in this example.

Function: Select the **HttpTrigger1** function.

Add Enrollment Group

 Save

Group name *

contoso-custom-allocated-devices



Attestation Type ⓘ

Certificate Symmetric Key

Auto-generate keys ⓘ



Primary Key ⓘ

Enter your primary key

Secondary Key ⓘ

Enter your secondary key

IoT Edge device ⓘ

True False

Select how you want to assign devices to hubs ⓘ

Custom (Use Azure Function)

Select the IoT hubs this group can be assigned to: ⓘ

2 selected

[Link a new IoT hub](#)

 You can use Azure Functions to write your own custom allocation policy. Select the Functions app you want to use, and the provisioning service will trigger the app via an HTTP PUT request. The app will return the desired IoT hub and initial twin for provisioning the device. [Learn more.](#)

Select Azure Function

Subscription *

<Your Selected Subscription>

Function App *

contoso-function-app-1098

[Create a new function app](#)

Function *

HttpTrigger1

- After saving the enrollment, reopen it and make a note of the **Primary Key**. You must save the enrollment first to have the keys generated. This key will be used to generate unique device keys for simulated devices later.

Derive unique device keys

In this section, you create two unique device keys. One key will be used for a simulated toaster device. The other key will be used for a simulated heat pump device.

To generate the device key, you use the **Primary Key** you noted earlier to compute the **HMAC-SHA256** of the device registration ID for each device and convert the result into Base64 format. For more information on creating derived device keys with enrollment groups, see the group enrollments section of [Symmetric key attestation](#).

For the example in this article, use the following two device registration IDs and compute a device key for both

devices. Both registration IDs have a valid suffix to work with the example code for the custom allocation policy:

- **breakroom499-contoso-tstrsd-007**
- **mainbuilding167-contoso-hpsd-088**
- [Windows](#)
- [Linux](#)

If you're using a Windows-based workstation, you can use PowerShell to generate your derived device key as shown in the following example.

Replace the value of **KEY** with the **Primary Key** you noted earlier.

```
$KEY='oiK77Oy7rBw8YB6IS6ukRChAw+Yq6GC61RMrPLSTi00tdI+XDu0LmLuNm11p+qv2I+adqGUdZHm46zXAQdZoOA=='  
  
$REG_ID1='breakroom499-contoso-tstrsd-007'  
$REG_ID2='mainbuilding167-contoso-hpsd-088'  
  
$hmacsha256 = New-Object System.Security.Cryptography.HMACSHA256  
$hmacsha256.key = [Convert]::FromBase64String($KEY)  
$sig1 = $hmacsha256.ComputeHash([Text.Encoding]::ASCII.GetBytes($REG_ID1))  
$sig2 = $hmacsha256.ComputeHash([Text.Encoding]::ASCII.GetBytes($REG_ID2))  
$derivedkey1 = [Convert]::ToBase64String($sig1)  
$derivedkey2 = [Convert]::ToBase64String($sig2)  
  
echo ``n`n$REG_ID1 : $derivedkey1`n$REG_ID2 : $derivedkey2`n`n"
```

```
breakroom499-contoso-tstrsd-007 : JC8F96eayuQwwz+PkE7IzjH2lIAjCUnAa61tDigBnSs=  
mainbuilding167-contoso-hpsd-088 : 6uejA9PfkQgmYylj8Zerp3kcbeVrGZ172YLa7VSnJzg=
```

The simulated devices will use the derived device keys with each registration ID to perform symmetric key attestation.

Prepare an Azure IoT C SDK development environment

In this section, you prepare the development environment used to build the [Azure IoT C SDK](#). The SDK includes the sample code for the simulated device. This simulated device will attempt provisioning during the device's boot sequence.

This section is oriented toward a Windows-based workstation. For a Linux example, see the set-up of the VMs in [How to provision for multitenancy](#).

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, which builds a version of the SDK specific to your development client platform. A Visual Studio solution for the simulated device will be generated in the `cmake` directory.

```
cmake -Dhsm_type_symm_key:BOOL=ON -Duse_prov_client:BOOL=ON ..
```

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

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

```
$ cmake -Dhsm_type_symm_key:BOOL=ON -Duse_prov_client:BOOL=ON ..  
-- 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
```

Simulate the devices

In this section, you update a provisioning sample named `prov_dev_client_sample` located in the Azure IoT C SDK you set up previously.

This sample code simulates a device boot sequence that sends the provisioning request to your Device Provisioning Service instance. The boot sequence will cause the toaster device to be recognized and assigned to the IoT hub using the custom allocation policy.

1. In the Azure portal, select the **Overview** tab for your Device Provisioning Service and note down the *ID Scope* value.

2. In Visual Studio, open the `azure_iot_sdks.sln` solution file that was generated by running CMake earlier. The solution file should be in the following location:

```
azure-iot-sdk-c\cmake\azure_iot_sdks.sln
```

3. In Visual Studio's *Solution Explorer* window, navigate to the **Provision_Samples** folder. Expand the sample project named **prov_dev_client_sample**. Expand **Source Files**, and open **prov_dev_client_sample.c**.
4. Find the `id_scope` constant, and replace the value with your **ID Scope** value that you copied earlier.

```
static const char* id_scope = "0ne00002193";
```

5. Find the definition for the `main()` function in the same file. Make sure the `hsm_type` variable is set to `SECURE_DEVICE_TYPE_SYMMETRIC_KEY` as shown below:

```
SECURE_DEVICE_TYPE hsm_type;
//hsm_type = SECURE_DEVICE_TYPE TPM;
//hsm_type = SECURE_DEVICE_TYPE X509;
hsm_type = SECURE_DEVICE_TYPE SYMMETRIC KEY;
```

6. Right-click the **prov_dev_client_sample** project and select **Set as Startup Project**.

Simulate the Contoso toaster device

1. To simulate the toaster device, find the call to `prov_dev_set_symmetric_key_info()` in `prov_dev_client_sample.c` which is commented out.

```
// Set the symmetric key if using they auth type  
//prov_dev_set_symmetric_key_info("<symm_registration_id>", "<symmetric_Key>");
```

Uncomment the function call and replace the placeholder values (including the angle brackets) with the toaster registration ID and derived device key you generated previously. The key value **JC8F96eayuQwwz+PkE7IzjH2lIAjCUnAa61tDigBnSs=** shown below is only given as an example.

```
// Set the symmetric key if using they auth type  
prov_dev_set_symmetric_key_info("breakroom499-contoso-tstrsd-007",  
"JC8F96eayuQwwz+PkE7IzjH2lIAjCUnAa61tDigBnSs=");
```

Save the file.

2. On the Visual Studio menu, select **Debug > Start without debugging** to run the solution. In the prompt to rebuild the project, select **Yes**, to rebuild the project before running.

The following output is an example of the simulated toaster device successfully booting up and connecting to the provisioning service instance to be assigned to the toasters IoT hub by the custom allocation policy:

```
Provisioning API Version: 1.3.6  
  
Registering Device  
  
Provisioning Status: PROV_DEVICE_REG_STATUS_CONNECTED  
Provisioning Status: PROV_DEVICE_REG_STATUS_ASSIGNING  
Provisioning Status: PROV_DEVICE_REG_STATUS_ASSIGNING  
  
Registration Information received from service: contoso-toasters-hub-1098.azure-devices.net,  
deviceId: breakroom499-contoso-tstrsd-007  
  
Press enter key to exit:
```

Simulate the Contoso heat pump device

1. To simulate the heat pump device, update the call to `prov_dev_set_symmetric_key_info()` in `prov_dev_client_sample.c` again with the heat pump registration ID and derived device key you generated earlier. The key value **6uejA9PfkQgmYylj8Zerp3kcbeVrGZ172YLa7VSnJzg=** shown below is also only given as an example.

```
// Set the symmetric key if using they auth type  
prov_dev_set_symmetric_key_info("mainbuilding167-contoso-hpsd-088",  
"6uejA9PfkQgmYylj8Zerp3kcbeVrGZ172YLa7VSnJzg=");
```

Save the file.

2. On the Visual Studio menu, select **Debug > Start without debugging** to run the solution. In the prompt to rebuild the project, select **Yes** to rebuild the project before running.

The following output is an example of the simulated heat pump device successfully booting up and connecting to the provisioning service instance to be assigned to the Contoso heat pumps IoT hub by the custom allocation policy:

```

Provisioning API Version: 1.3.6

Registering Device

Provisioning Status: PROV_DEVICE_REG_STATUS_CONNECTED
Provisioning Status: PROV_DEVICE_REG_STATUS_ASSIGNING
Provisioning Status: PROV_DEVICE_REG_STATUS_ASSIGNING

Registration Information received from service: contoso-heatpumps-hub-1098.azure-devices.net,
deviceId: mainbuilding167-contoso-hpsd-088

Press enter key to exit:

```

Troubleshooting custom allocation policies

The following table shows expected scenarios and the results error codes you might receive. Use this table to help troubleshoot custom allocation policy failures with your Azure Functions.

SCENARIO	REGISTRATION RESULT FROM PROVISIONING SERVICE	PROVISIONING SDK RESULTS
The webhook returns 200 OK with 'iotHubHostName' set to a valid IoT hub host name	Result status: Assigned	SDK returns PROV_DEVICE_RESULT_OK along with hub information
The webhook returns 200 OK with 'iotHubHostName' present in the response, but set to an empty string or null	Result status: Failed Error code: CustomAllocationIoTHubNotSpecified (400208)	SDK returns PROV_DEVICE_RESULT_HUB_NOT_SPECIFIED
The webhook returns 401 Unauthorized	Result status: Failed Error code: CustomAllocationUnauthorizedAccess (400209)	SDK returns PROV_DEVICE_RESULT_UNAUTHORIZED
An Individual Enrollment was created to disable the device	Result status: Disabled	SDK returns PROV_DEVICE_RESULT_DISABLED
The webhook returns error code >= 429	DPS' orchestration will retry a number of times. The retry policy is currently: - Retry count: 10 - Initial interval: 1s - Increment: 9s	SDK will ignore error and submit another get status message in the specified time
The webhook returns any other status code	Result status: Failed Error code: CustomAllocationFailed (400207)	SDK returns PROV_DEVICE_RESULT_DEV_AUTH_ERROR

Clean up resources

If you plan to continue working with the resources created in this article, you can leave them. If you don't plan to continue using the resources, use the following steps to delete all of the resources created in this article to avoid unnecessary charges.

The steps here assume you created all resources in this article as instructed in the same resource group named **contoso-us-resource-group**.

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 you want to keep, only delete the IoT Hub resource itself instead of deleting the resource group.

To delete the 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 resources, **contoso-us-resource-group**.
3. To the right of your resource group in the result list, select ... then **Delete resource group**.
4. You'll 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

- To learn more Reprovisioning, see [IoT Hub Device reprovisioning concepts](#)
- To learn more Deprovisioning, see [How to deprovision devices that were previously autoprovioned](#)

Programmatically create a Device Provisioning Service enrollment group for X.509 certificate attestation

8/22/2022 • 13 minutes to read • [Edit Online](#)

This article shows you how to programmatically create an [enrollment group](#) that uses intermediate or root CA X.509 certificates. The enrollment group is created by using the [Azure IoT Hub DPS service SDK](#) and a sample application. An enrollment group controls access to the provisioning service for devices that share a common signing certificate in their certificate chain. To learn more, see [Controlling device access to the provisioning service with X.509 certificates](#). For more information about using X.509 certificate-based Public Key Infrastructure (PKI) with Azure IoT Hub and Device Provisioning Service, see [X.509 CA certificate security overview](#).

Prerequisites

- If you don't have an Azure subscription, create a [free account](#) before you begin.
- Complete the steps in [Set up IoT Hub Device Provisioning Service with the Azure portal](#).
- Install [.NET 6.0 SDK or later](#) or later on your Windows-based machine. You can use the following command to check your version.

```
dotnet --info
```

- Install [Node.js v4.0 or above](#) or later on your machine.
- [Java SE Development Kit 8](#). This article installs the [Java Service SDK](#) below. It works on both Windows and Linux. This article uses Windows.
- [Maven 3](#).
- Install the latest version of [Git](#). Make sure that Git is added to the environment variables accessible to the command window. See [Software Freedom Conservancy's Git client tools](#) for the latest version of `git` tools to install, which includes *Git Bash*, the command-line app that you can use to interact with your local Git repository.

NOTE

Although the steps in this article work on both Windows and Linux computers, this article uses a Windows development computer.

Create test certificates

Enrollment groups that use X.509 certificate attestation can be configured to use a root CA certificate or an intermediate certificate. The more usual case is to configure the enrollment group with an intermediate certificate. This provides more flexibility as multiple intermediate certificates can be generated or revoked by the same root CA certificate.

For this article, you'll need either a root CA certificate file, an intermediate CA certificate file, or both in `.pem` or

.cer format. One file contains the public portion of the root CA X.509 certificate and the other contains the public portion of the intermediate CA X.509 certificate.

If you already have a root CA file and/or an intermediate CA file, you can continue to [Add and verify your root or intermediate CA certificate](#).

If you don't have a root CA file and/or an intermediate CA file, follow the steps in [Create an X.509 certificate chain](#) to create them. You can stop after you complete the steps in [Create the intermediate CA certificate](#) as you won't need device certificates to complete the steps in this article. When you're finished, you'll have two X.509 certificate files: `./certs/azure-iot-test-only.root.ca.cert.pem` and `./certs/azure-iot-test-only.intermediate.cert.pem`.

Add and verify your root or intermediate CA certificate

Devices that provision through an enrollment group using X.509 certificates, present the entire certificate chain when they authenticate with DPS. For DPS to be able to validate the certificate chain, the root or intermediate certificate configured in an enrollment group must either be a verified certificate or must roll up to a verified certificate in the certificate chain a device presents when it authenticates with the service.

For this article, assuming you have both a root CA certificate and an intermediate CA certificate signed by the root CA:

- If you plan on creating the enrollment group with the root CA certificate, you'll need to upload and verify the root CA certificate.
- If you plan on creating the enrollment group with the intermediate CA certificate, you can upload and verify either the root CA certificate or the intermediate CA certificate. (If you have multiple intermediate CA certificates in the certificate chain, you could, alternatively, upload and verify any intermediate certificate that sits between the root CA certificate and the intermediate certificate that you create the enrollment group with.)

To add and verify your root or intermediate CA certificate to the Device Provisioning Service:

1. Sign in to the [Azure portal](#).
2. On the left-hand menu or on the portal page, select **All resources**.
3. Select your Device Provisioning Service.
4. In the **Settings** menu, select **Certificates**.
5. On the top menu, select **+ Add**:
6. Enter a name for your root or intermediate CA certificate, and upload the .pem or .cer file.
7. Select **Set certificate status to verified on upload**.

The screenshot shows the DPS Certificates blade on the left and the 'Add certificate' dialog on the right. In the Certificates blade, 'Certificates' is selected in the left menu. The 'Add' button in the dialog is highlighted with a red box. In the dialog, the 'Certificate name' field contains 'azure-iot-test-only-root'. The 'Certificate .pem or .cer file' field contains the path '"/azuresdk/testonly/root.ca.cert.pem"'. The 'Set certificate status to verified on upload' checkbox is checked. A note says 'We'll verify this certificate automatically, with no manual verification steps required.' A 'Save' button is at the bottom.

8. Select **Save**.

Get the connection string for your provisioning service

For the sample in this article, you'll need to copy the connection string for your provisioning service.

1. Sign in to the [Azure portal](#).
2. On the left-hand menu or on the portal page, select **All resources**.
3. Select your Device Provisioning Service.
4. In the **Settings** menu, select **Shared access policies**.
5. Select the access policy that you want to use.
6. In the **Access Policy** panel, copy and save the primary key connection string.

The screenshot shows the DPS Shared access policies blade on the left and the 'Access Policy' dialog on the right. In the Shared access policies blade, 'Shared access policies' is selected in the left menu. The 'provisioningserviceowner' policy is selected and highlighted with a red box. The 'Access Policy' dialog shows the 'provisioningserviceowner' policy details. The 'Name' is 'provisioningserviceowner'. Under 'Permissions', all checkboxes are checked: 'Service configuration', 'Enrollment read', 'Enrollment write', 'Registration status read', and 'Registration status write'. Under 'Shared Access keys', there are two fields: 'Primary key' and 'Secondary key', each with a red box around its respective input field. Below these are the 'Primary key connection string' and 'Secondary key connection string', both with red boxes around their input fields. A 'Save' button is at the bottom.

Create the enrollment group sample

This section shows you how to create a .NET Core console application that adds an enrollment group to your provisioning service.

1. Open a Windows command prompt and navigate to a folder where you want to create your app.
2. To create a console project, run the following command:

```
dotnet new console --framework net6.0 --use-program-main
```

3. To add a reference to the DPS service SDK, run the following command:

```
dotnet add package Microsoft.Azure.Devices.Provisioning.Service
```

This step downloads, installs, and adds a reference to the [Azure IoT DPS service client NuGet package](#) and its dependencies. This package includes the binaries for the .NET service SDK.

4. Open *Program.cs* file in an editor.
5. Replace the namespace statement at the top of the file with the following:

```
namespace CreateEnrollmentGroup;
```

6. Add the following `using` statements at the top of the file **above** the `namespace` statement:

```
using System.Security.Cryptography.X509Certificates;
using System.Threading.Tasks;
using Microsoft.Azure.Devices.Provisioning.Service;
```

7. Add the following fields to the `Program` class, and make the indicated changes.

```
private static string ProvisioningConnectionString = "{ProvisioningServiceConnectionString}";
private static string EnrollmentGroupId = "enrollmentgroup1";
private static string X509RootCertPath = @"{Path to a .cer or .pem file for a verified root CA or
intermediate CA X.509 certificate}";
```

- Replace the `ProvisioningServiceConnectionString` placeholder value with the connection string of the provisioning service that you copied in the previous section.
- Replace the `X509RootCertPath` placeholder value with the path to a .pem or .cer file. This file represents the public part of either a root CA X.509 certificate that has been previously uploaded and verified with your provisioning service, or an intermediate certificate that has itself been uploaded and verified or had a certificate in its signing chain uploaded and verified.
- You may optionally change the `EnrollmentGroupId` value. The string can contain only lower case characters and hyphens.

IMPORTANT

In production code, be aware of the following security considerations:

- Hard-coding the connection string for the provisioning service administrator is against security best practices. Instead, the connection string should be held in a secure manner, such as in a secure configuration file or in the registry.
- Be sure to upload only the public part of the signing certificate. Never upload .pfx (PKCS12) or .pem files containing private keys to the provisioning service.

8. Add the following method to the `Program` class. This code creates an `EnrollmentGroup` entry and then calls the `ProvisioningServiceClient.CreateOrUpdateEnrollmentGroupAsync` method to add the enrollment group to the provisioning service.

```
public static async Task RunSample()
{
    Console.WriteLine("Starting sample...");

    using (ProvisioningServiceClient provisioningServiceClient =
        ProvisioningServiceClient.CreateFromConnectionString(ProvisioningConnectionString))
    {
        #region Create a new enrollmentGroup config
        Console.WriteLine("\nCreating a new enrollmentGroup...");
        var certificate = new X509Certificate2(X509RootCertPath);
        Attestation attestation = X509Attestation.CreateFromRootCertificates(certificate);
        EnrollmentGroup enrollmentGroup =
            new EnrollmentGroup(
                EnrollmentGroupId,
                attestation)
        {
            ProvisioningStatus = ProvisioningStatus.Enabled
        };
        Console.WriteLine(enrollmentGroup);
        #endregion

        #region Create the enrollmentGroup
        Console.WriteLine("\nAdding new enrollmentGroup...");
        EnrollmentGroup enrollmentGroupResult =
            await
provisioningServiceClient.CreateOrUpdateEnrollmentGroupAsync(enrollmentGroup).ConfigureAwait(false);
        Console.WriteLine("\nEnrollmentGroup created with success.");
        Console.WriteLine(enrollmentGroupResult);
        #endregion

    }
}
```

9. Finally, replace the `Main` method with the following lines:

```
static async Task Main(string[] args)
{
    await RunSample();
    Console.WriteLine("\nHit <Enter> to exit ...");
    Console.ReadLine();
}
```

10. Save your changes.

This section shows you how to create a Node.js script that adds an enrollment group to your provisioning service.

1. From a command window in your working folder, run:

```
npm install azure-iot-provisioning-service
```

This step downloads, installs, and adds a reference to the [Azure IoT DPS service client package](#) and its dependencies. This package includes the binaries for the Node.js service SDK.

2. Using a text editor, create a `create_enrollment_group.js` file in your working folder. Add the following code to the file and save:

```
'use strict';
var fs = require('fs');

var provisioningServiceClient = require('azure-iot-provisioning-
service').ProvisioningServiceClient;

var serviceClient = provisioningServiceClient.fromConnectionString(process.argv[2]);

var enrollment = {
    enrollmentGroupId: 'first',
    attestation: {
        type: 'x509',
        x509: {
            signingCertificates: {
                primary: {
                    certificate: fs.readFileSync(process.argv[3], 'utf-8').toString()
                }
            }
        }
    },
    provisioningStatus: 'disabled'
};

serviceClient.createOrUpdateEnrollmentGroup(enrollment, function(err, enrollmentResponse) {
    if (err) {
        console.log('error creating the group enrollment: ' + err);
    } else {
        console.log("enrollment record returned: " + JSON.stringify(enrollmentResponse, null, 2));
        enrollmentResponse.provisioningStatus = 'enabled';
        serviceClient.createOrUpdateEnrollmentGroup(enrollmentResponse, function(err,
enrollmentResponse) {
            if (err) {
                console.log('error updating the group enrollment: ' + err);
            } else {
                console.log("updated enrollment record returned: " + JSON.stringify(enrollmentResponse,
null, 2));
            }
        });
    }
});
```

1. Open a Windows command prompt.

2. Clone the GitHub repo for device enrollment code sample using the [Java Service SDK](#):

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

3. From the location where you downloaded the repo, go to the sample folder:

```
cd azure-iot-sdk-java\provisioning\provisioning-samples\service-enrollment-group-sample
```

4. Open the file `/src/main/java/samples/com/microsoft/azure/sdk/iot/ServiceEnrollmentGroupSample.java` in an editor of your choice.
5. Replace `[Provisioning Connection String]` with the connection string that you copied in [Get the connection string for your provisioning service](#).
6. Replace the `PUBLIC_KEY_CERTIFICATE_STRING` constant string with the value of your root or intermediate CA certificate `.pem` file. This file represents the public part of either a root CA X.509 certificate that has been previously uploaded and verified with your provisioning service, or an intermediate certificate that has itself been uploaded and verified or had a certificate in its signing chain uploaded and verified.

The syntax of certificate text must follow the pattern below with no extra spaces or characters.

```
private static final String PUBLIC_KEY_CERTIFICATE_STRING =
    "-----BEGIN CERTIFICATE-----\n" +
    "MIIFOjCCAYKgAwIBAgIJAPzMa6s7mj7+MA0GCSqGSIb3DQEBCwUAMCoxKDAmBgNV\n" +
    ...
    "MDMwWhcNMjAxMTIyMjEzMDMwWjAqMSgwJgYDVQQDDB9BenVyZSBjb1QgSHViIENB\n" +
"-----END CERTIFICATE-----";
```

Updating this string value manually can be prone to error. To generate the proper syntax, you can copy and paste the following command into a **Git Bash** prompt, replace `your-cert.pem` with the location of your certificate file, and press **ENTER**. This command will generate the syntax for the `PUBLIC_KEY_CERTIFICATE_STRING` string constant value and write it to the output.

```
sed 's/^"/;$ !s/$/\n" +;$ s$/"/' your-cert.pem
```

Copy and paste the output certificate text for the constant value.

IMPORTANT

In production code, be aware of the following security considerations:

- Hard-coding the connection string for the provisioning service administrator is against security best practices. Instead, the connection string should be held in a secure manner, such as in a secure configuration file or in the registry.
- Be sure to upload only the public part of the signing certificate. Never upload .pfx (PKCS12) or .pem files containing private keys to the provisioning service.

7. The sample allows you to set an IoT hub in the enrollment group to provision the device to. This must be an IoT hub that has been previously linked to the provisioning service. For this article, we'll let DPS choose from the linked hubs according to the default allocation policy, evenly-weighted distribution. Comment out the following statement in the file:

```
enrollmentGroup.setIoTHubHostName(IOTHUB_HOST_NAME); // Optional parameter.
```

8. The sample code creates, updates, queries, and deletes an enrollment group for X.509 devices. To verify successful creation of the enrollment group in Azure portal, comment out the following lines of code near the end of the file:

```
// **** Delete info of enrollmentGroup  
*****  
System.out.println("\nDelete the enrollmentGroup...");  
provisioningServiceClient.deleteEnrollmentGroup(enrollmentGroupId);
```

9. Save the *ServiceEnrollmentGroupSample.java* file.

Run the enrollment group sample

1. Run the sample:

```
dotnet run
```

2. Upon successful creation, the command window displays the properties of the new enrollment group.

1. Run the following command in your command prompt. Include the quotes around the command arguments and replace `<connection string>` with the connection string you copied in the previous section, and `<certificate .pem file>` with the path to your certificate `.pem` file. This file represents the public part of either a root CA X.509 certificate that has been previously uploaded and verified with your provisioning service, or an intermediate certificate that has itself been uploaded and verified or had a certificate in its signing chain uploaded and verified.

```
node create_enrollment_group.js "<connection string>" "<certificate .pem file>"
```

2. Upon successful creation, the command window displays the properties of the new enrollment group.

1. From the `azure-iot-sdk-java/provisioning/provisioning-samples/service-enrollment-group-sample` folder in your command prompt, run the following command to build the sample:

```
mvn install -DskipTests
```

This command downloads the [Azure IoT DPS service client Maven package](#) to your machine and builds the sample. This package includes the binaries for the Java service SDK.

2. Switch to the `target` folder and run the sample. Be aware that the build in the previous step outputs `.jar` file in the `target` folder with the following file format: `provisioning-x509-sample-{version}-with-deps.jar`; for example: `provisioning-x509-sample-1.8.1-with-deps.jar`. You may need to replace the version in the command below.

```
cd target  
java -jar ./service-enrollment-group-sample-1.8.1-with-deps.jar
```

3. Upon successful creation, the command window displays the properties of the new enrollment group.

To verify that the enrollment group has been created:

1. In the Azure portal, select your Device Provisioning Service.
2. In the **Settings** menu, select **Manage enrollments**.
3. Select **Enrollment Groups**. You should see a new enrollment entry that corresponds to the enrollment group ID that you used in the sample.

contoso-dps-2 | Manage enrollments

Azure IoT Hub Device Provisioning Service (DPS)

Search (Ctrl+ /) Add enrollment group Add individual enrollment Refresh Delete

Overview Activity log Access control (IAM) Tags Diagnose and solve problems

Settings Quick Start Shared access policies Linked IoT hubs Certificates

Manage enrollments **Manage allocations policy**

You can add or remove individual device enrollments and/or enrollment groups from this page. [Learn more](#)

Enrollment Groups Individual Enrollments

Search group enrollment by group name (name has to be exact match)

GROUP NAME

enrollmentgroupstest

contoso-dps-2 | Manage enrollments

Azure IoT Hub Device Provisioning Service (DPS)

Search (Ctrl+ /) Add enrollment group Add individual enrollment Refresh Delete

Overview Activity log Access control (IAM) Tags Diagnose and solve problems

Settings Quick Start Shared access policies Linked IoT hubs Certificates

Manage enrollments **Manage allocations policy**

You can add or remove individual device enrollments and/or enrollment groups from this page. [Learn more](#)

Enrollment Groups Individual Enrollments

Search group enrollment by group name (name has to be exact match)

GROUP NAME

first

contoso-dps-2 | Manage enrollments

Azure IoT Hub Device Provisioning Service (DPS)

Search (Ctrl+ /) Add enrollment group Add individual enrollment Refresh Delete

Overview Activity log Access control (IAM) Tags Diagnose and solve problems

Settings Quick Start Shared access policies Linked IoT hubs Certificates

Manage enrollments **Manage allocations policy**

You can add or remove individual device enrollments and/or enrollment groups from this page. [Learn more](#)

Enrollment Groups Individual Enrollments

Search group enrollment by group name (name has to be exact match)

GROUP NAME

enrollmentgroupid-acbb8f43

Clean up resources

If you plan to explore the Azure IoT Hub Device Provisioning Service tutorials, don't clean up the resources created in this article. Otherwise, use the following steps to delete all resources created by this article.

1. Close the sample output window on your computer.

2. From the left-hand menu in the Azure portal, select **All resources**.
3. Select your Device Provisioning Service.
4. In the left-hand menu under **Settings**, select **Manage enrollments**.
5. Select the **Enrollment Groups** tab.
6. Select the check box next to the *GROUP NAME* of the enrollment group you created in this article.
7. At the top of the page, select **Delete**.
8. From your Device Provisioning Service in the Azure portal, select **Certificates** under **Settings** on the left-hand menu.
9. Select the certificate you uploaded for this article.
10. At the top of **Certificate Details**, select **Delete**.

Certificate tooling

The [Azure IoT C SDK](#) has scripts that can help you create root CA, intermediate CA, and device certificates, and do proof-of-possession with the service to verify root and intermediate CA certificates. To learn more, see [Managing test CA certificates for samples and tutorials](#).

The [Group certificate verification sample](#) in the [Azure IoT Samples for C# \(.NET\)](#) shows how to do proof-of-possession in C# with an existing X.509 intermediate or root CA certificate.

The [Azure IoT Node.js SDK](#) has scripts that can help you create root CA, intermediate CA, and device certificates, and do proof-of-possession with the service to verify root and intermediate CA certificates. To learn more, see [Tools for the Azure IoT Device Provisioning Device SDK for Node.js](#).

You can also use tools available in the [Azure IoT C SDK](#). To learn more, see [Managing test CA certificates for samples and tutorials](#).

The [Azure IoT Java SDK](#) contains test tooling that can help you create an X.509 certificate chain, upload a root or intermediate certificate from that chain, and do proof-of-possession with the service to verify root and intermediate CA certificates. To learn more, see [X509 certificate generator using DICE emulator](#).

Next steps

In this article, you created an enrollment group for an X.509 intermediate or root CA certificate using the Azure IoT Hub Device Provisioning Service. To explore further, check out the following links:

- For more information about X.509 certificate attestation with DPS, see [X.509 certificate attestation](#).
- For an end-to-end example of provisioning devices through an enrollment group using X.509 certificates, see the [Provision multiple X.509 devices using enrollment groups](#) tutorial.
- To learn about managing individual enrollments and enrollment groups using Azure portal, see [How to manage device enrollments with Azure portal](#).

Programmatically create a Device Provisioning Service individual enrollment for TPM attestation

8/22/2022 • 12 minutes to read • [Edit Online](#)

This article shows you how to programmatically create an individual enrollment for a TPM device in the Azure IoT Hub Device Provisioning Service by using the [Azure IoT Hub DPS service SDK](#) and a sample application. After you've created the individual enrollment, you can optionally enroll a simulated TPM device to the provisioning service through this enrollment entry.

Although these steps work on both Windows and Linux computers, this article uses a Windows development computer.

Prerequisites

- If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.
- Complete the steps in [Set up IoT Hub Device Provisioning Service with the Azure portal](#).
- Install [.NET 6.0 SDK or later](#) or later on your Windows-based machine. You can use the following command to check your version.

```
dotnet --info
```

- (Optional) If you want to enroll a simulated device at the end of this article, follow the procedure in [Create and provision a simulated TPM device](#) up to the step where you get an endorsement key for the device. Save the **Endorsement key**, you'll use it later in this article.

NOTE

Don't follow the steps to create an individual enrollment by using the Azure portal.

- Install [Node.js v4.0+](#).
- (Optional) If you want to enroll a simulated device at the end of this article, follow the procedure in [Create and provision a simulated TPM device](#) up to the step where you get an endorsement key and registration ID for the device. Save the **Endorsement key** and **Registration ID**, you'll use them later in this article.

NOTE

Don't follow the steps to create an individual enrollment by using the Azure portal.

- Install the [Java SE Development Kit 8](#). This article installs the [Java Service SDK](#) below. It works on both Windows and Linux. This article uses Windows.
- Install [Maven 3](#).
- Install [Git](#) and make sure the the path is added to the environment variable `PATH`.
- (Optional) If you want to enroll a simulated device at the end of this article, follow the procedure in

Create and provision a simulated TPM device up to the step where you get an endorsement key for the device. Note the **Endorsement key** and the **Registration ID**, you'll use them later in this article.

NOTE

Don't follow the steps to create an individual enrollment by using the Azure portal.

Get TPM endorsement key (Optional)

You can follow the steps in this article to create a sample individual enrollment. In this, case, you'll be able to view the enrollment entry in DPS, but you won't be able to use it to provision a device.

You can also choose to follow the steps in this article to create an individual enrollment and enroll a simulated TPM device. If you want to enroll a simulated device at the end of this article, follow the procedure in [Create and provision a simulated TPM device](#) up to the step where you get an endorsement key for the device. Save the **Endorsement key**, you'll use it later in this article.

NOTE

Don't follow the steps to create an individual enrollment by using the Azure portal.

You can also choose to follow the steps in this article to create an individual enrollment and enroll a simulated TPM device. If you want to enroll a simulated device at the end of this article, follow the procedure in [Create and provision a simulated TPM device](#) up to the step where you get an endorsement key and registration ID for the device. Save the **Endorsement key** and **Registration ID**, you'll use them later in this article.

NOTE

Don't follow the steps to create an individual enrollment by using the Azure portal.

You can also choose to follow the steps in this article to create an individual enrollment and enroll a simulated TPM device. If you want to enroll a simulated device at the end of this article, follow the procedure in [Create and provision a simulated TPM device](#) up to the step where you get an endorsement key for the device. Note the **Endorsement key** and the **Registration ID**, you'll use them later in this article.

NOTE

Don't follow the steps to create an individual enrollment by using the Azure portal.

Get the connection string for your provisioning service

For the sample in this article, you'll need to copy the connection string for your provisioning service.

1. Sign in to the [Azure portal](#).
2. On the left-hand menu or on the portal page, select **All resources**.
3. Select your Device Provisioning Service.
4. In the **Settings** menu, select **Shared access policies**.
5. Select the access policy that you want to use.
6. In the **Access Policy** panel, copy and save the primary key connection string.

The screenshot shows the Azure portal interface for managing Device Provisioning Services. On the left, the navigation menu includes options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings (Quick Start, Shared access policies, Linked IoT hubs, Certificates, Manage enrollments, Manage allocation policy, Networking, Properties, Locks), Monitoring (Alerts, Metrics, Diagnostic settings), and Home > Contoso-DPS. The main content area is titled 'Access Policy' for 'Contoso-DPS | Shared access policies'. It displays a table with one row for 'provisioningserviceowner'. A tooltip explains that Device Provisioning Services use permissions to grant access to each Device Provisioning Service configuration. The 'provisioningserviceowner' row has a checked checkbox under 'Policy ↑↓'. The 'Permissions' section lists five checked items: Service configuration, Enrollment read, Enrollment write, Registration status read, and Registration status write. Below this, the 'Shared Access keys' section shows two key fields: 'Primary key' and 'Secondary key', each with a copy icon. Under 'Primary key connection string', there is a redacted string followed by a copy icon. A 'Save' button is at the bottom.

Create the individual enrollment sample

This section shows you how to create a .NET Core console app that adds an individual enrollment for a TPM device to your provisioning service.

1. Open a Windows command prompt and navigate to a folder where you want to create your app.
2. To create a console project, run the following command:

```
dotnet new console --framework net6.0 --use-program-main
```

3. To add a reference to the DPS service SDK, run the following command:

```
dotnet add package Microsoft.Azure.Devices.Provisioning.Service
```

This step downloads, installs, and adds a reference to the [Azure IoT DPS service client NuGet package](#) and its dependencies. This package includes the binaries for the .NET service SDK.

4. Open *Program.cs* file in an editor.
5. Replace the namespace statement at the top of the file with the following:

```
namespace CreateIndividualEnrollment;
```

6. Add the following `using` statements at the top of the file **above** the `namespace` statement:

```
using System.Threading.Tasks;
using Microsoft.Azure.Devices.Provisioning.Service;
```

7. Add the following fields to the `Program` class, and make the listed changes.

```

private static string ProvisioningConnectionString = "{ProvisioningServiceConnectionString}";
private const string RegistrationId = "sample-registrationid-csharp";
private const string TpmEndorsementKey =
    "AToAAQALAMAsgAgg3GXZ0SEs/gakMyNRqXXJP1S124GUgtk8qHaGzMuaaoABgCAAEMAEAgAAAAAAEAxsj2gUS" +
    "cTk1UjuioeTlfGYZrrimExB+bScH75adUMRIi2UOMxG1kw4y+9RW/IVoMl4e620VxZad0ARX2guqvjY07KPvt3d" +
    "yKhZS3dkcvfBisBhP1XH9B33VqHG9SHnbnQXdBuacgKAfxome8UmBKfe+naTsE5fkvjb/do3/dD614sGBwFCnKR" +
    "dln4XpM03zLpoHFa08z0wt81/uP3qUIxmCYv9A7m69Ms+5/pCkTu/rK4mRDsfhZ0QLfbzVI6zQFOKF/rwsfBtFe" +
    "WlWtcuJMK1XdD8TXWE1Tzgh7JS4qhFzreL0c1mI0GCj+Aws0usZh7dLIVPnlgZcBhgy1SSDQM==";

// Optional parameters
private const string OptionalDeviceId = "myCSharpDevice";
private const ProvisioningStatus OptionalProvisioningStatus = ProvisioningStatus.Enabled;

```

- Replace the `ProvisioningServiceConnectionString` placeholder value with the connection string of the provisioning service that you copied in the previous section.
- If you're using this article together with the [Create and provision a simulated TPM device quickstart](#) to provision a simulated device, replace the endorsement key with the value that you noted in that quickstart. You can replace the device ID and registration ID with the values suggested in that quickstart, use your own values, or use the default values in this sample.

8. Add the following method to the `Program` class. This code creates an individual enrollment entry and then calls the `CreateOrUpdateIndividualEnrollmentAsync` method on the `ProvisioningServiceClient` to add the individual enrollment to the provisioning service.

```

public static async Task RunSample()
{
    Console.WriteLine("Starting sample...");

    using (ProvisioningServiceClient provisioningServiceClient =
        ProvisioningServiceClient.CreateFromConnectionString(ProvisioningConnectionString))
    {
        #region Create a new individualEnrollment config
        Console.WriteLine("\nCreating a new individualEnrollment object...");
        Attestation attestation = new TpmAttestation(TpmEndorsementKey);
        IndividualEnrollment individualEnrollment =
            new IndividualEnrollment(
                RegistrationId,
                attestation);

        // The following parameters are optional. Remove them if you don't need them.
        individualEnrollment.DeviceId = OptionalDeviceId;
        individualEnrollment.ProvisioningStatus = OptionalProvisioningStatus;
        #endregion

        #region Create the individualEnrollment
        Console.WriteLine("\nAdding the individualEnrollment to the provisioning service...");
        IndividualEnrollment individualEnrollmentResult =
            await
provisioningServiceClient.CreateOrUpdateIndividualEnrollmentAsync(individualEnrollment).ConfigureAwait(false);
        Console.WriteLine("\nIndividualEnrollment created with success.");
        Console.WriteLine(individualEnrollmentResult);
        #endregion

    }
}

```

9. Finally, replace the `Main` method with the following lines:

```
static async Task Main(string[] args)
{
    await RunSample();
    Console.WriteLine("\nHit <Enter> to exit ...");
    Console.ReadLine();
}
```

10. Save your changes.

1. From a command window in your working folder, run:

```
npm install azure-iot-provisioning-service
```

This step downloads, installs, and adds a reference to the [Azure IoT DPS service client package](#) and its dependencies. This package includes the binaries for the Node.js service SDK.

2. Using a text editor, create a *create_individual_enrollment.js* file in your working folder. Add the following code to the file:

```
'use strict';

var provisioningServiceClient = require('azure-iot-provisioning-service').ProvisioningServiceClient;

var serviceClient = provisioningServiceClient.fromConnectionString(process.argv[2]);
var endorsementKey = process.argv[3];

var enrollment = {
    registrationId: 'first',
    attestation: {
        type: 'tpm',
        tpm: {
            endorsementKey: endorsementKey
        }
    }
};

serviceClient.createOrUpdateIndividualEnrollment(enrollment, function(err, enrollmentResponse) {
    if (err) {
        console.log('error creating the individual enrollment: ' + err);
    } else {
        console.log("enrollment record returned: " + JSON.stringify(enrollmentResponse, null, 2));
    }
});
```

3. Save the file.

1. Open a Windows command prompt.

2. Clone the [Microsoft Azure IoT SDKs for Java GitHub repo](#):

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

3. Go to the sample folder:

```
cd azure-iot-sdk-java\provisioning\provisioning-samples\service-enrollment-sample
```

4. Open the file *|src|main|java|samples|com|microsoft|azure|sdk|iot|ServiceEnrollmentSample.java* in an editor.

5. Replace [Provisioning Connection String] with the connection string that you copied in [Get the connection string for your provisioning service](#).

```
private static final String PROVISIONING_CONNECTION_STRING = "[Provisioning Connection String]";
```

6. Add the TPM device details. Replace the [RegistrationId] and [TPM Endorsement Key] in the following statements with your endorsement key and registration ID.

```
private static final String REGISTRATION_ID = "[RegistrationId]";
private static final String TPM_ENDORSEMENT_KEY = "[TPM Endorsement Key]";
```

- If you're using this article together with the [Create and provision a simulated TPM device](#) quickstart to provision a simulated device, use the **Registration ID** and **Endorsement key** values that you noted from that quickstart.
- If you're using this article to just create a sample individual enrollment and don't intend to use it to enroll a device, you can use the following value for an endorsement key:

```
private static final String TPM_ENDORSEMENT_KEY =
"AToAAQALAAASgAgg3GXZ0SEs/gakMyNRqXXJP1S124GUgtk8qHaGzMUaoABgCAAEMEAgAAAAAAEAsj2gUScTk1Uj
uioeTlfGYZrrimExB+bSch75adUMRIi2UOMxG1kw4y+9RW/IVoM14e620VxZad0ARX2gUqvjY07KPVt3dyKhZS3dkcvfBi
sBhP1XH9B33VqHG9SHnbnQXdBuAcgKAfxome8UmBKfe+naTsE5fkvjb/do3/dD614sGBwFCnKRdln4XpM03zLpoHFao8zO
wt81/uP3qUIxmCYv9A7m69Ms+5/pCkTu/rK4mRDsfhZ0QLfbzVI6zQFOKF/rwsfBtFeWlWtcuJMK1XdD8TXWE1Tzgh7JS4
qhFzreL0c1mI0GCj+Aws0usZh7dLIVPnefZcBhgy1SSDQM==";
```

Enter your own value for the registration ID, for example, "myJavaDevice".

7. For individual enrollments, you can choose to set a device ID that DPS will assign to the device when it provisions it to IoT Hub. If you don't assign a device ID, DPS will use the registration ID as the device ID. By default, this sample assigns "myJavaDevice" as the device ID. If you want to change the device ID, modify the following statement:

```
private static final String DEVICE_ID = "myJavaDevice";
```

If you don't want to assign a specific device ID, comment out the following statement:

```
individualEnrollment.setDeviceId(DEVICE_ID);
```

8. The sample allows you to set an IoT hub in the individual enrollment to provision the device to. This must be an IoT hub that has been previously linked to the provisioning service. For this article, we'll let DPS choose from the linked hubs according to the default allocation policy, evenly-weighted distribution. Comment out the following statement in the file:

```
individualEnrollment.setIoTHubHostName(IOTHUB_HOST_NAME);
```

9. The sample creates, updates, queries, and deletes an individual TPM device enrollment. To verify successful enrollment in portal, temporarily comment out the following lines of code at the end of the file:

```
// **** Delete info of individualEnrollment
*****
System.out.println("\nDelete the individualEnrollment...");
provisioningServiceClient.deleteIndividualEnrollment(REGISTRATION_ID);
```

10. Save your changes.

Run the individual enrollment sample

1. Run the sample:

```
dotnet run
```

2. Upon successful creation, the command window displays the properties of the new enrollment.

To run the sample, you'll need the connection string for your provisioning service that you copied in the previous section, as well as the endorsement key for the device. If you've followed the [Create and provision a simulated device](#) quickstart to create a simulated TPM device, use the endorsement key created for that device. Otherwise, to create a sample individual enrollment, you can use the following endorsement key supplied with the [Node.js Service SDK](#):

```
AToAAQALAAASgAgg3GXZ0SEs/gakMyNRqXXJP1S124GUgtk8qHaGzMuaaoABgCAAEMAEAgAAAAAAEAsj2gUScTk1UjuioeT1fGYZrrimE
xB+bScH75adUMRIi2UOMxG1kw4y+9RW/IVoM14e620VxZad0ARX2gUqVjY07KPvt3dyKhZS3dkcvfBisBhP1XH9B33VqHG9SHnbnQXdBuA
CgKAfxome8UmBKfe+naTsE5fkvb/bo3/dD614sGBwFCnKRdln4XpM03zLpoHFaoz0wt8l/uP3qUIxmCYv9A7m69Ms+5/pCkTu/rK4mRDsfhZ
0QLfbzVI6zQFOKF/rwsfBtFeWlWtcuJMK1XdD8TXWE1Tzgh7JS4qhFzreL0c1mI0GCj+Aws0usZh7dLIVPnlgZcBhgy1SSDQM==
```

1. To create an individual enrollment for your TPM device, run the following command (include the quotes around the command arguments):

```
node create_individual_enrollment.js "<the connection string for your provisioning service>" "
<endorsement key>"
```

2. Upon successful creation, the command window displays the properties of the new enrollment.

1. From the `azure-iot-sdk-java/provisioning/provisioning-samples/service-enrollment-sample` folder in your command prompt, run the following command to build the sample:

```
mvn install -DskipTests
```

This command downloads the [Azure IoT DPS service client Maven package](#) to your machine and builds the sample. This package includes the binaries for the Java service SDK.

2. Switch to the `target` folder and run the sample. Be aware that the build in the previous step outputs `.jar` file in the `target` folder with the following file format: `service-enrollment-sample-{version}-with-deps.jar`; for example: `service-enrollment-sample-1.8.1-with-deps.jar`. You may need to replace the version in the command below.

```
cd target
java -jar ./service-enrollment-sample-1.8.1-with-deps.jar
```

3. Upon successful creation, the command window displays the properties of the new enrollment.

To verify that the enrollment group has been created:

1. In the Azure portal, select your Device Provisioning Service.
2. In the Settings menu, select **Manage enrollments**.
3. Select **Individual Enrollments**. You should see a new enrollment entry that corresponds to the registration ID that you used in the sample.

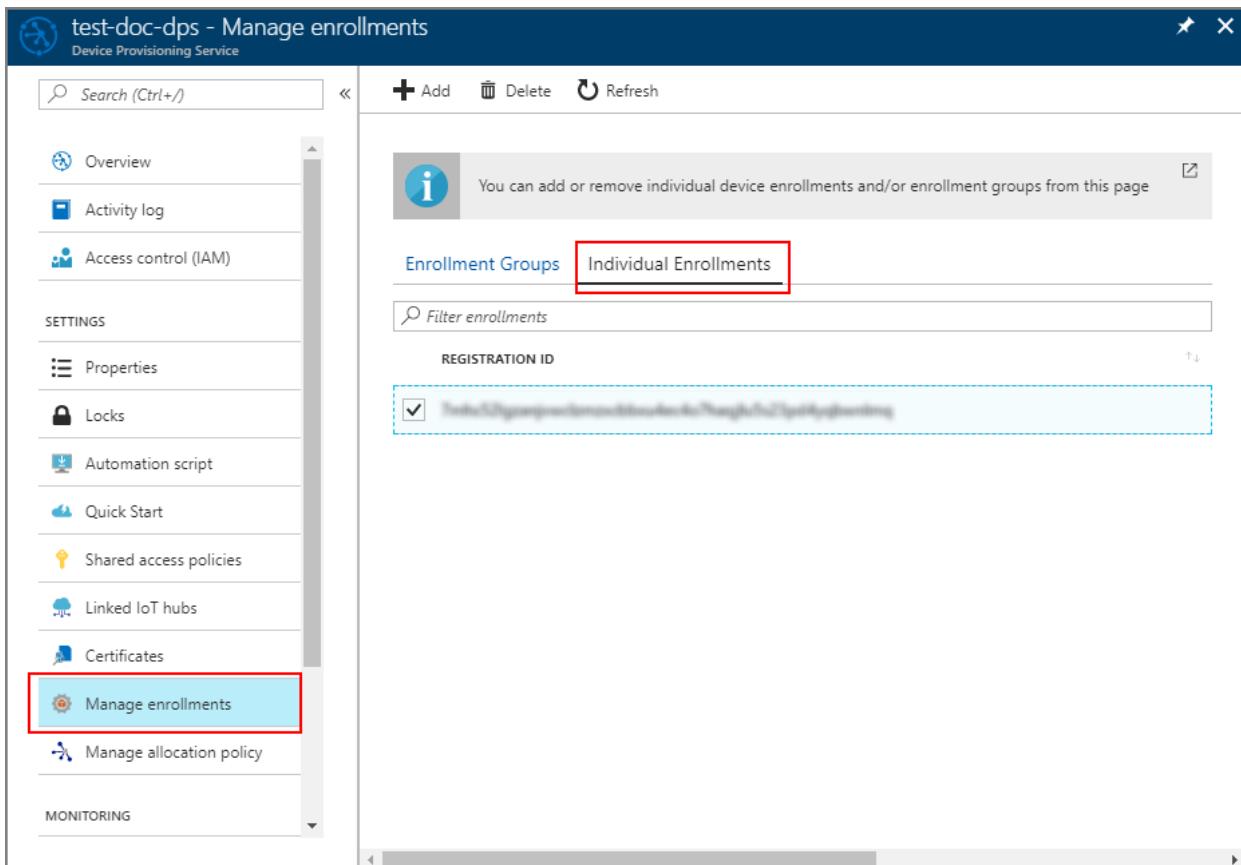
The screenshot shows the Azure Device Provisioning Service interface. On the left, there's a navigation sidebar with various options like Overview, Activity log, Access control (IAM), Tags, Quick Start, Shared access policies, Linked IoT hubs, Certificates, and two main management sections: **Manage enrollments** and **Manage allocation policy**. The **Manage enrollments** section is highlighted with a red box. On the right, the main content area is titled "sample-provisioning-service - Manage enrollments". It has tabs for "Enrollment Groups" and "Individual Enrollments", with "Individual Enrollments" being the active tab and also highlighted with a red box. Below this, there's a search bar labeled "Filter enrollments" and a table header "REGISTRATION ID". A row in the table is selected and highlighted with a red box, showing the value "sample-registrationid-csharp". Above the table, a message box says: "You can add or remove individual device enrollments and/or enrollment groups from this page".

This screenshot shows a detailed view of an individual enrollment entry. The left sidebar has the same structure as the previous screenshot, with the **Manage enrollments** section highlighted. The main content area shows an enrollment entry named "first". The entry is displayed in a modal window with the following details:

- Registration status:** Status: Unassigned, Assigned hub: -, Device ID: -, Last assigned: -
- Identity attestation information:** Mechanism: TPM, Endorsement key: ****
- IoT Hub:** Assign automatically
- IoT Hub device ID:** (empty input field)
- Initial device twin state:** A JSON object with placeholder values:


```
{
        "tags": {},
        "desiredproperties": {}
      }
```

A "Save" button is located at the bottom right of the modal.



Enroll a simulated device (Optional)

If you've been following steps in the [Create and provision a simulated TPM device](#) quickstart to provision a simulated device, resume the quickstart at [Register the device](#).

If you've been following steps in the [Create and provision a simulated TPM device](#) quickstart to provision a simulated device, resume the quickstart at [Register the device](#).

If you've been following steps in the [Create and provision a simulated TPM device](#) quickstart to provision a simulated device, resume the quickstart at [Register the device](#).

Clean up resources

If you plan to explore the DPS tutorials, don't clean up the resources created in this article. Otherwise, use the following steps to delete all resources created by this article.

1. Close the sample output window on your computer.
2. From the left-hand menu in the Azure portal, select **All resources**.
3. Select your Device Provisioning Service.
4. In the left-hand menu under **Settings**, select **Manage enrollments**.
5. Select the **Individual Enrollments** tab.
6. Select the check box next to the *REGISTRATION ID* of the enrollment entry you created in this article.
7. At the top of the page, select **Delete**.
8. If you followed the steps in [Create and provision a simulated TPM device](#) to create a simulated TPM device, do the following steps:
 - a. In the Azure portal, navigate to the IoT Hub where your device was provisioned.

- b. In the left-hand menu under **Device management**, select **Devices**.
 - c. Select the check box next to the *Device ID* of the device you registered in this article.
 - d. At the top of the pane, select **Delete**.
8. If you followed the steps in [Create and provision a simulated TPM device](#) to create a simulated TPM device, do the following steps:
 - a. Close the TPM simulator window and the sample output window for the simulated device.
 - b. In the Azure portal, navigate to the IoT Hub where your device was provisioned.
 - c. In the left-hand menu under **Device management**, select **Devices**.
 - d. Select the check box next to the *Device ID* of the device you registered in this article.
 - e. At the top of the pane, select **Delete**.
8. If you followed the steps in [Create and provision a simulated TPM device](#) to create a simulated TPM device, do the following steps:
 - a. Close the TPM simulator window and the sample output window for the simulated device.
 - b. In the Azure portal, navigate to the IoT Hub where your device was provisioned.
 - c. In the left-hand menu under **Device management**, select **Devices**.
 - d. Select the check box next to the *Device ID* of the device you registered in this article.
 - e. At the top of the pane, select **Delete**.

Next steps

In this article, you've programmatically created an individual enrollment entry for a TPM device. Optionally, you created a TPM simulated device on your computer and provisioned it to your IoT hub using the Azure IoT Hub Device Provisioning Service. To explore further, check out the following links:

- For more information about TPM attestation with DPS, see [TPM attestation](#).
- For an end-to-end example of provisioning a device through an individual enrollment using TPM attestation, see the [Provision a simulated TPM device quickstart](#).
- To learn about managing individual enrollments and enrollment groups using Azure portal, see [How to manage device enrollments with Azure portal](#).

Communicate with your DPS using the MQTT protocol

8/22/2022 • 3 minutes to read • [Edit Online](#)

DPS enables devices to communicate with the DPS device endpoint using:

- [MQTT v3.1.1](#) on port 8883
- [MQTT v3.1.1](#) over WebSocket on port 443.

DPS 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 DPS.

All device communication with DPS must be secured using TLS/SSL. Therefore, DPS doesn't support non-secure connections over port 1883.

NOTE

DPS does not currently support devices using TPM [attestation mechanism](#) over the MQTT protocol.

Connecting to DPS

A device can use the MQTT protocol to connect to a DPS instance using any of the following options.

- Libraries in the [Azure IoT Provisioning SDKs](#).
- The MQTT protocol directly.

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 **registrationId**.
- For the **Username** field, use `{idScope}/registrations/{registration_id}/api-version=2019-03-31`, where `{idScope}` is the [ID scope](#) of the DPS and `{registration_id}` is the [Registration ID](#) for your device.

NOTE

If you use X.509 certificate authentication, the registration ID is provided by the subject common name (CN) of your device leaf (end-entity) certificate. `{registration_id}` in the **Username** field must match the common name.

- 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 sr={URL-encoded-resourceURI}&sig={signature-string}&se={expiry}&skn=registration`

The resourceURI should be in the format `{idScope}/registrations/{registration_id}`. The policy name (`skn`) should be set to `registration`.

NOTE

If you use X.509 certificate authentication, SAS token passwords are not required.

For more information about how to generate SAS tokens, see the security tokens section of [Control access to DPS](#).

The following is a list of DPS implementation-specific behaviors:

- DPS doesn't support persistent sessions. It treats every session as non-persistent, regardless of the value of the **CleanSession** flag. We recommend setting **CleanSession** to true.
- When a device app subscribes to a topic with **QoS 2**, DPS grants maximum QoS level 1 in the **SUBACK** packet. After that, DPS delivers messages to the device using QoS 1.

TLS/SSL configuration

To use the MQTT protocol directly, your client *must* connect over TLS 1.2. Attempts to skip this step fail with connection errors.

Registering a device

To register a device through DPS, a device should subscribe using `$dps/registrations/res/#` 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. DPS does not allow the usage of the `#` or `?` wildcards for filtering of subtopics. Since DPS is not a general-purpose pub-sub messaging broker, it only supports the documented topic names and topic filters.

The device should publish a register message to DPS using

`$dps/registrations/PUT/iotdps-register/?$rid={request_id}` as a **Topic Name**. The payload should contain the [Device Registration](#) object in JSON format. In a successful scenario, the device will receive a response on the `$dps/registrations/res/202/?$rid={request_id}&retry-after=x` topic name where x is the retry-after value in seconds. The payload of the response will contain the [RegistrationOperationStatus](#) object in JSON format.

Polling for registration operation status

The device must poll the service periodically to receive the result of the device registration operation. Assuming that the device has already subscribed to the `$dps/registrations/res/#` topic as indicated above, it can publish a get operationstatus message to the

`$dps/registrations/GET/iotdps-get-operationstatus/?$rid={request_id}&operationId={operationId}` topic name.

The operation ID in this message should be the value received in the [RegistrationOperationStatus](#) response message in the previous step. In the successful case, the service will respond on the

`$dps/registrations/res/200/?$rid={request_id}` topic. The payload of the response will contain the

[RegistrationOperationStatus](#) object. The device should keep polling the service if the response code is 202 after a delay equal to the retry-after period. The device registration operation is successful if the service returns a 200 status code.

Connecting over Websocket

When connecting over Websocket, specify the subprotocol as `mqtt`. Follow [RFC 6455](#).

Next steps

To learn more about the MQTT protocol, see the [MQTT documentation](#).

To further explore the capabilities of DPS, see:

- [About IoT DPS](#)

How to manage device enrollments with Azure portal

8/22/2022 • 7 minutes to read • [Edit Online](#)

A *device enrollment* creates a record of a single device or a group of devices that may at some point register with the Azure IoT Hub Device Provisioning Service (DPS). The enrollment record contains the initial configuration for the device(s) as part of that enrollment. Included in the configuration is either the IoT hub to which a device will be assigned, or an allocation policy that configures the hub from a set of hubs. This article shows you how to manage device enrollments for your provisioning service.

The Azure IoT Device Provisioning Service supports two types of enrollments:

- **Enrollment groups:** Used to enroll multiple related devices.
- **Individual enrollments:** Used to enroll a single device.

IMPORTANT

If you have trouble accessing enrollments from the Azure portal, it may be because you have public network access disabled or IP filtering rules configured that block access for the Azure portal. To learn more, see [Disable public network access limitations](#) and [IP filter rules limitations](#).

Create an enrollment group

An enrollment group is an entry for a group of devices that share a common attestation mechanism. We recommend that you use an enrollment group for a large number of devices that share an initial configuration, or for devices that go to the same tenant. Devices that use either [symmetric key](#) or [X.509 certificates](#) attestation are supported.

Create a symmetric key enrollment group

To create and use enrollment groups with symmetric keys, see the [Provision devices with symmetric keys](#) tutorial.

To create a symmetric key enrollment group:

1. Sign in to the [Azure portal](#).
2. On the left-hand menu or on the portal page, select **All resources**.
3. Select the Device Provisioning Service to which you want to enroll your device.
4. In the **Settings** menu, select **Manage enrollments**.
5. At the top of the page, select **+ Add enrollment group**.
6. In the **Add Enrollment Group** page, enter the following information:

FIELD	DESCRIPTION
-------	-------------

FIELD	DESCRIPTION
Group name	The name of the group of devices. The enrollment group name is a case-insensitive string (up to 128 characters long) of alphanumeric characters plus the special characters: '-' , '.' , '_' , ':' . The last character must be alphanumeric or dash ('-').
Attestation Type	Select Symmetric Key .
Auto Generate Keys	Check this box.
Select how you want to assign devices to hubs	Select <i>Static configuration</i> so that you can assign to a specific hub
Select the IoT hubs this group can be assigned to	Select one of your hubs.

Leave the rest of the fields at their default values.

Add Enrollment Group

Save

Group name *

Attestation Type ⓘ

Certificate Symmetric Key

Auto-generate keys ⓘ

Primary Key ⓘ

Secondary Key ⓘ

IoT Edge device ⓘ

True False

Select how you want to assign devices to hubs ⓘ

Static configuration

Select the IoT hubs this group can be assigned to: * ⓘ

Contoso-IotHub-2.azure-devices.net

[Link a new IoT hub](#)

Select how you want device data to be handled on re-provisioning * ⓘ

Re-provision and migrate data

7. Select **Save**.

Create a X.509 certificate enrollment group

To create a X.509 certificate enrollment group:

1. Sign in to the [Azure portal](#).
2. On the left-hand menu or on the portal page, select **All resources**.
3. Select the Device Provisioning Service to which you want to enroll your device.
4. In the **Settings** menu, select **Manage enrollments**.
5. At the top of the page, select **+ Add enrollment group**.
6. In the **Add Enrollment Group** page, enter the following information:

FIELD	DESCRIPTION
Group name	The name of the group of devices.
Attestation Type	Select Certificate .
Certificate Type	Select CA Certificate or Intermediate based on which certificate signed your device certificates.
Primary Certificate	If you're signing your device certificates with a CA certificate, then the root CA certificate must have proof of possession completed. If you're signing your device certificates with an intermediate certificate, an upload button will be available to allow you to upload your intermediate certificate. The certificate that signed the intermediate must also have proof of possession completed.

Leave the rest of the fields at their default values.

Add Enrollment Group

X

 Save

Group name *

MySymmetricGroup



Attestation Type ⓘ

Certificate Symmetric Key

IoT Edge device ⓘ

True False

Certificate Type ⓘ

CA Certificate Intermediate Certificate

Primary Certificate ⓘ

RootCATest



Secondary Certificate ⓘ

No certificate selected



Select how you want to assign devices to hubs ⓘ

Static configuration



Select the IoT hubs this group can be assigned to: * ⓘ

Contoso-IoTHub-2.azure-devices.net



Link a new IoT hub

7. Select **Save**.

Create an individual enrollment

An individual enrollment is an entry for a single device that may be assigned to an IoT hub. Devices using [symmetric key](#), [X.509 certificates](#), and [TPM attestation](#) are supported.

Create a symmetric key individual enrollment

TIP

For more detailed instructions on how to create and use individual enrollments with symmetric keys, see [Quickstart: Provision a simulated symmetric key device](#).

To create a symmetric key individual enrollment:

1. Sign in to the [Azure portal](#).
2. On the left-hand menu or on the portal page, select **All resources**.
3. Select the Device Provisioning Service to which you want to enroll your device.

4. In the **Settings** menu, select **Manage enrollments**.
5. At the top of the page, select **+ Add individual enrollment**.
6. In the **Add Enrollment** page, enter the following information.

FIELD	DESCRIPTION
Mechanism	Select <i>Symmetric Key</i>
Auto Generate Keys	Check this box.
Registration ID	Type in a unique registration ID.
IoT Hub Device ID	This ID will represent your device. It must follow the rules for a device ID. For more information, see Device identity properties . If the device ID is left unspecified, then the registration ID will be used.
Select how you want to assign devices to hubs	Select <i>Static configuration</i> so that you can assign to a specific hub
Select the IoT hubs this group can be assigned to	Select one of your hubs.



Add Enrollment

X

Save

Mechanism * ⓘ

Symmetric Key



Auto-generate keys ⓘ



Primary Key ⓘ

Enter your primary key

Secondary Key ⓘ

Enter your secondary key

Registration ID *

regid1234



IoT Hub Device ID ⓘ

contoso-us-devices

IoT Edge device ⓘ

True

False

Select how you want to assign devices to hubs ⓘ

Static configuration



Select the IoT hubs this device can be assigned to: * ⓘ

Contoso-lotHub-2.azure-devices.net



Link a new IoT hub

7. Select **Save**.

Create a X.509 certificate individual enrollment

TIP

For more detailed instructions on how to create and use individual enrollments with X.509 certificates, see [Quickstart: Provision a X.509 certificate device](#).

To create a X.509 certificate individual enrollment:

1. Sign in to the [Azure portal](#).
2. On the left-hand menu or on the portal page, select **All resources**.
3. Select the Device Provisioning Service to which you want to enroll your device.
4. In the **Settings** menu, select **Manage enrollments**.
5. At the top of the page, select **+ Add individual enrollment**.

6. In the Add Enrollment page, enter the following information.

FIELD	DESCRIPTION
Mechanism	Select <i>X.509</i>
Primary Certificate .pem or .cer file	Upload a certificate from which you may generate leaf certificates. If choosing .cer file, only base-64 encoded certificate is accepted.
IoT Hub Device ID	This ID will represent your device. It must follow the rules for a device ID. For more information, see [Device identity properties](./iot-hub/iot-hub-devguide-identity-registry). The device ID must be the subject name on the device certificate that you upload for the enrollment. That subject name must conform to the rules for a device ID.
Select how you want to assign devices to hubs	Select <i>Static configuration</i> so that you can assign to a specific hub
Select the IoT hubs this group can be assigned to	Select one of your hubs.



Add Enrollment



Save

Mechanism * ⓘ

X.509



Primary Certificate .pem or .cer file ⓘ

Select a file



Clear Selection

Secondary Certificate .pem or .cer file ⓘ

Select a file



Clear Selection

IoT Hub Device ID ⓘ

contoso-us-devices

IoT Edge device ⓘ

True

False

Select how you want to assign devices to hubs ⓘ

Static configuration



Select the IoT hubs this device can be assigned to: * ⓘ

Contoso-lotHub-2.azure-devices.net



Link a new IoT hub

7. Select **Save**.

Create a TPM individual enrollment

TIP

For more detailed instructions on how to create and use individual enrollments using TPM attestation, see one of the [Provision a simulated TPM device samples](#).

To create a TPM individual enrollment:

1. Sign in to the [Azure portal](#).
2. On the left-hand menu or on the portal page, select **All resources**.
3. Select the Device Provisioning Service to which you want to enroll your device.
4. In the **Settings** menu, select **Manage enrollments**.
5. At the top of the page, select **+ Add individual enrollment**.
6. In the **Add Enrollment** page, enter the following information.

FIELD	DESCRIPTION
Mechanism	Select <i>TPM</i>
Endorsement Key	The unique endorsement key of the TPM device.
Registration ID	Type in a unique registration ID.
IoT Hub Device ID	This ID will represent your device. It must follow the rules for a device ID. For more information, see [Device identity properties](./iot-hub/iot-hub-devguide-identity-registry). If the device ID is left unspecified, then the registration ID will be used.
Select how you want to assign devices to hubs	Select <i>Static configuration</i> so that you can assign to a specific hub
Select the IoT hubs this group can be assigned to	Select one of your hubs.



Add Enrollment



Save

Mechanism * ⓘ

TPM

Endorsement key *

Endorsement key

Registration ID *

Individual enrollment registration id

IoT Hub Device ID ⓘ

contoso-us-devices

IoT Edge device ⓘ

True

False

Select how you want to assign devices to hubs ⓘ

Static configuration

Select the IoT hubs this device can be assigned to: * ⓘ

Contoso-IotHub-2.azure-devices.net

Link a new IoT hub

Select how you want device data to be handled on re-provisioning * ⓘ

Re-provision and migrate data

7. Select **Save**.

Update an enrollment entry

To update an existing enrollment entry:

1. Sign in to the [Azure portal](#).
2. On the left-hand menu or on the portal page, select **All resources**.
3. Select the Device Provisioning Service to which you want to enroll your device.
4. In the **Settings** menu, select **Manage enrollments**.
5. Select the enrollment entry that you wish to modify.
6. On the enrollment entry details page, you can update all items, except the security type and credentials.
7. Once completed, select **Save**.

Remove a device enrollment

To remove an enrollment entry:

1. Sign in to the [Azure portal](#).
2. On the left-hand menu or on the portal page, select **All resources**.
3. Select the Device Provisioning Service to which you want to enroll your device.
4. In the **Settings** menu, select **Manage enrollments**.
5. Select the enrollment entry you want to remove.
6. At the top of the page, select **Delete**.
7. When prompted to confirm, select **Yes**.
8. Once the action is completed, you'll see that your entry has been removed from the list of device enrollments.

NOTE

Deleting an enrollment group doesn't delete the registration records for devices in the group. DPS uses the registration records to determine whether the maximum number of registrations has been reached for the DPS instance. Orphaned registration records still count against this quota. For the current maximum number of registrations supported for a DPS instance, see [Quotas and limits](#).

You may want to delete the registration records for the enrollment group before deleting the enrollment group itself. You can see and manage the registration records for an enrollment group manually on the **Registration Records** tab for the group in Azure portal. You can retrieve and manage the registration records programmatically using the [Device Registration State REST APIs](#) or equivalent APIs in the [DPS service SDKs](#), or using the [az iot dps enrollment-group registration Azure CLI commands](#).

How to do proof-of-possession for X.509 CA certificates with your Device Provisioning Service

8/22/2022 • 4 minutes to read • [Edit Online](#)

A verified X.509 Certificate Authority (CA) certificate is a CA certificate that has been uploaded and registered to your provisioning service and has gone through proof-of-possession with the service.

Verified certificates play an important role when using enrollment groups. Verifying certificate ownership provides an additional security layer by ensuring that the uploader of the certificate is in possession of the certificate's private key. Verification prevents a malicious actor sniffing your traffic from extracting an intermediate certificate and using that certificate to create an enrollment group in their own provisioning service, effectively hijacking your devices. By proving ownership of the root or an intermediate certificate in a certificate chain, you're proving that you have permission to generate leaf certificates for the devices that will be registering as a part of that enrollment group. For this reason, the root or intermediate certificate configured in an enrollment group must either be a verified certificate or must roll up to a verified certificate in the certificate chain a device presents when it authenticates with the service. To learn more about X.509 certificate attestation, see [X.509 certificates](#) and [Controlling device access to the provisioning service with X.509 certificates](#).

Automatic verification of intermediate or root CA through self-attestation

If you are using an intermediate or root CA that you trust and know you have full ownership of the certificate, you can self-attest that you have verified the certificate.

To add an auto-verified certificate, follow these steps:

1. In the Azure portal, navigate to your provisioning service and open **Certificates** from the left-hand menu.
2. Click **Add** to add a new certificate.
3. Enter a friendly display name for your certificate. Browse to the .cer or .pem file that represents the public part of your X.509 certificate. Click **Upload**.
4. Check the box next to **Set certificate status to verified on upload**.

The screenshot shows the 'Add certificate' dialog box. It includes fields for 'Certificate name' (containing 'myCert'), 'Certificate .pem or .cer file' (containing 'fullchain.pem'), and a checked checkbox for 'Set certificate status to verified on upload'. A note at the bottom states: 'We'll verify this certificate automatically, with no manual verification steps required.' A red box highlights the 'Set certificate status to verified on upload' checkbox.

Add certificate

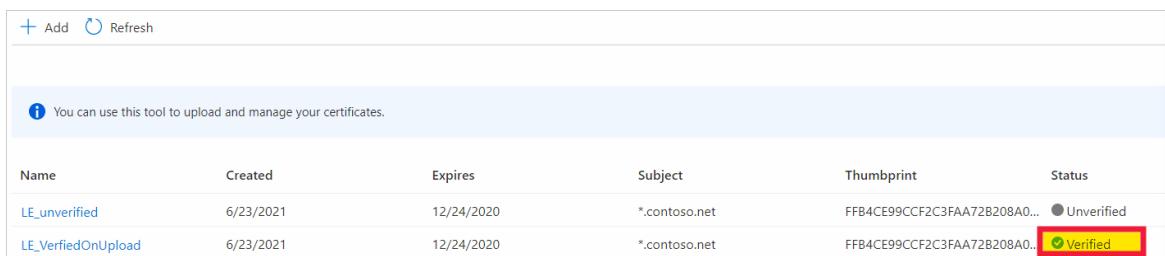
Certificate name * ⓘ
myCert ✓

Certificate .pem or .cer file. ⓘ
"fullchain.pem" ⌂

Set certificate status to verified on upload ⓘ

ⓘ We'll verify this certificate automatically, with no manual verification steps required. [Learn more](#)

5. Click **Save**.
6. Your certificate is show in the certificate tab with a status *Verified*.



Name	Created	Expires	Subject	Thumbprint	Status
LE_unverified	6/23/2021	12/24/2020	*.contoso.net	FFB4CE99CCF2C3FAA72B208A0...	<input type="radio"/> Unverified
LE_VerifiedOnUpload	6/23/2021	12/24/2020	*.contoso.net	FFB4CE99CCF2C3FAA72B208A0...	<input checked="" type="radio"/> Verified

Manual verification of intermediate or root CA

Proof-of-possession involves the following steps:

1. Get a unique verification code generated by the provisioning service for your X.509 CA certificate. You can do this from the Azure portal.
2. Create an X.509 verification certificate with the verification code as its subject and sign the certificate with the private key associated with your X.509 CA certificate.
3. Upload the signed verification certificate to the service. The service validates the verification certificate using the public portion of the CA certificate to be verified, thus proving that you are in possession of the CA certificate's private key.

Register the public part of an X.509 certificate and get a verification code

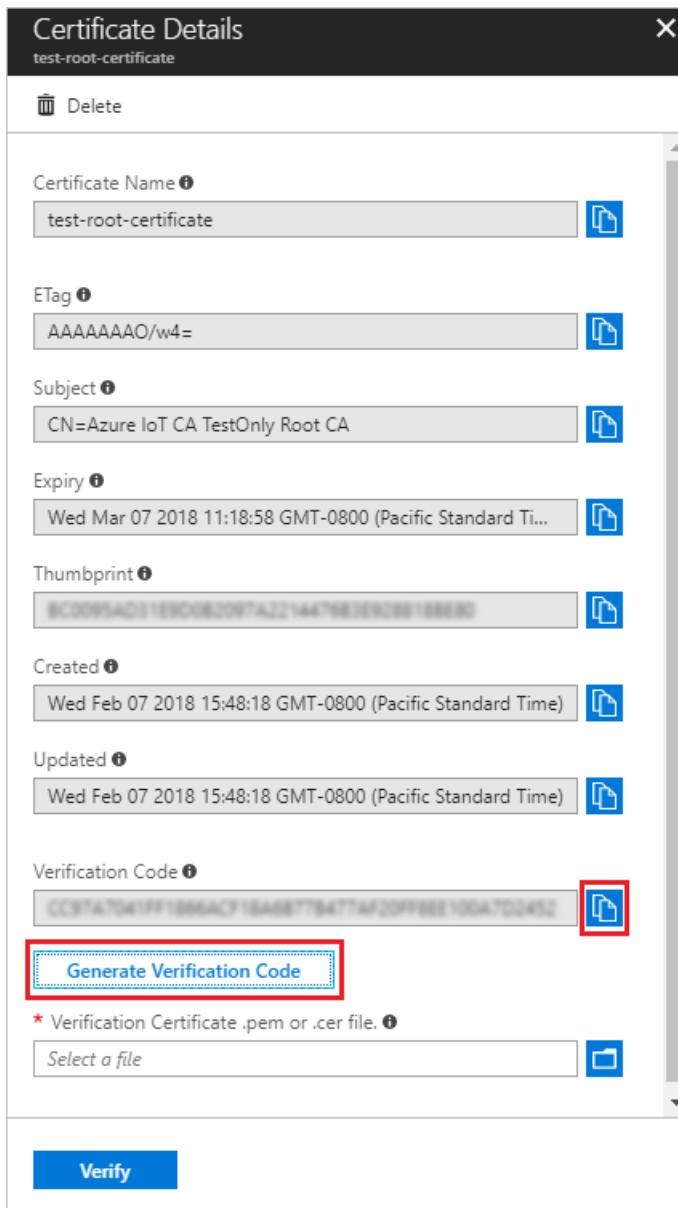
To register a CA certificate with your provisioning service and get a verification code that you can use during proof-of-possession, follow these steps.

1. In the Azure portal, navigate to your provisioning service and open **Certificates** from the left-hand menu.
2. Click **Add** to add a new certificate.
3. Enter a friendly display name for your certificate. Browse to the .cer or .pem file that represents the public part of your X.509 certificate. Click **Upload**.
4. Once you get a notification that your certificate is successfully uploaded, click **Save**.

The screenshot shows the Microsoft Azure portal interface. On the left, there's a sidebar with various service icons. The main area is titled "sample-provisioning-service - Certificates". In the center, there's a table with columns: NAME, STATUS, EXPIRY, and SUBJECT. Below the table, it says "No results". At the top right, there's a "Save" button.

Your certificate will show in the **Certificate Explorer** list. Note that the **STATUS** of this certificate is *Unverified*.

5. Click on the certificate that you added in the previous step.
6. In **Certificate Details**, click **Generate Verification Code**.
7. The provisioning service creates a **Verification Code** that you can use to validate the certificate ownership. Copy the code to your clipboard.



Digitally sign the verification code to create a verification certificate

Now, you need to sign the *Verification Code* with the private key associated with your X.509 CA certificate, which generates a signature. This is known as [Proof of possession](#) and results in a signed verification certificate.

Microsoft provides tools and samples that can help you create a signed verification certificate:

- The **Azure IoT Hub C SDK** provides PowerShell (Windows) and Bash (Linux) scripts to help you create CA and leaf certificates for development and to perform proof-of-possession using a verification code. You can download the [files](#) relevant to your system to a working folder and follow the instructions in the [Managing CA certificates readme](#) to perform proof-of-possession on a CA certificate.
- The **Azure IoT Hub C# SDK** contains the [Group Certificate Verification Sample](#), which you can use to do proof-of-possession.

IMPORTANT

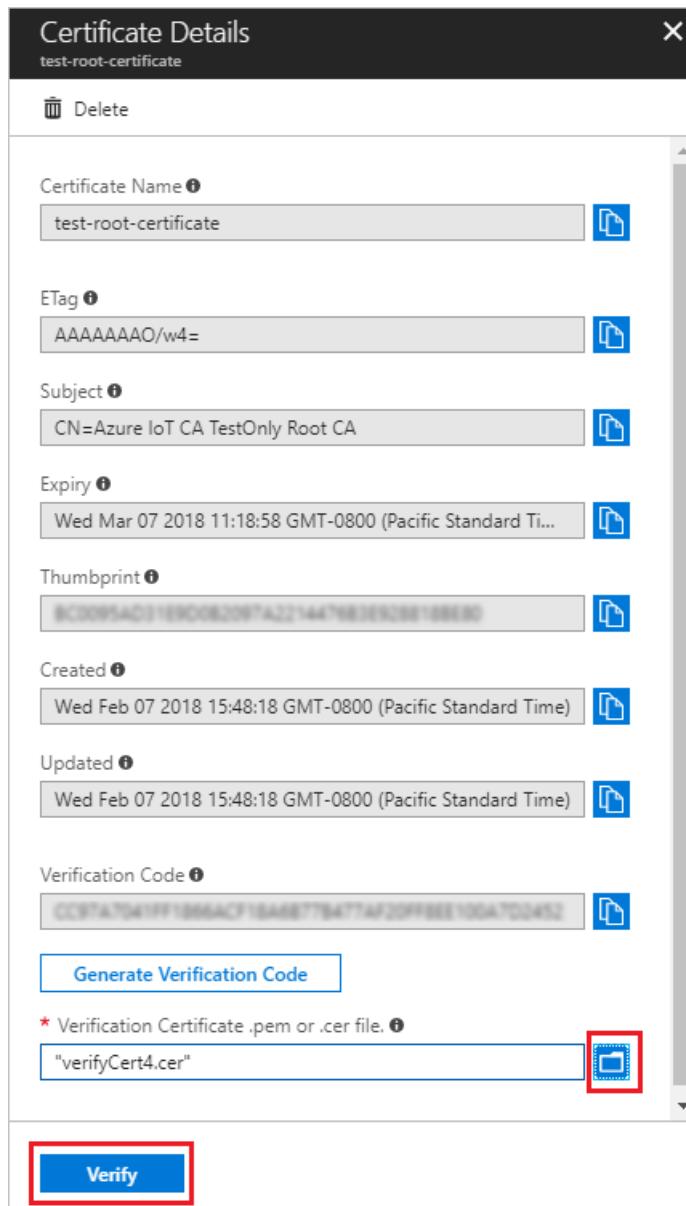
In addition to performing proof-of-possession, the PowerShell and Bash scripts cited previously also allow you to create root certificates, intermediate certificates, and leaf certificates that can be used to authenticate and provision devices. These certificates should be used for development only. They should never be used in a production environment.

The PowerShell and Bash scripts provided in the documentation and SDKs rely on [OpenSSL](#). You may also use OpenSSL or other third-party tools to help you do proof-of-possession. For an example using tooling provided

with the SDKs, see [Create an X.509 certificate chain](#).

Upload the signed verification certificate

1. Upload the resulting signature as a verification certificate to your provisioning service in the portal. In **Certificate Details** on the Azure portal, use the *File Explorer* icon next to the **Verification Certificate .pem or .cer file** field to upload the signed verification certificate from your system.
2. Once the certificate is successfully uploaded, click **Verify**. The **STATUS** of your certificate changes to **Verified** in the **Certificate Explorer** list. Click **Refresh** if it does not update automatically.



Next steps

- To learn about how to use the portal to create an enrollment group, see [Managing device enrollments with Azure portal](#).
- To learn about how to use the service SDKs to create an enrollment group, see [Managing device enrollments with service SDKs](#).

How to roll X.509 device certificates

8/22/2022 • 11 minutes to read • [Edit Online](#)

During the lifecycle of your IoT solution, you'll need to roll certificates. Two of the main reasons for rolling certificates would be a security breach, and certificate expirations.

Rolling certificates is a security best practice to help secure your system in the event of a breach. As part of [Assume Breach Methodology](#), Microsoft advocates the need for having reactive security processes in place along with preventative measures. Rolling your device certificates should be included as part of these security processes. The frequency in which you roll your certificates will depend on the security needs of your solution. Customers with solutions involving highly sensitive data may roll certificate daily, while others roll their certificates every couple years.

Rolling device certificates will involve updating the certificate stored on the device and the IoT hub. Afterwards, the device can reprovision itself with the IoT hub using normal [provisioning](#) with the Device Provisioning Service (DPS).

Obtain new certificates

There are many ways to obtain new certificates for your IoT devices. These include obtaining certificates from the device factory, generating your own certificates, and having a third party manage certificate creation for you.

Certificates are signed by each other to form a chain of trust from a root CA certificate to a [leaf certificate](#). A signing certificate is the certificate used to sign the leaf certificate at the end of the chain of trust. A signing certificate can be a root CA certificate, or an intermediate certificate in chain of trust. For more information, see [X.509 certificates](#).

There are two different ways to obtain a signing certificate. The first way, which is recommended for production systems, is to purchase a signing certificate from a root certificate authority (CA). This way chains security down to a trusted source.

The second way is to create your own X.509 certificates using a tool like OpenSSL. This approach is great for testing X.509 certificates but provides few guarantees around security. We recommend you only use this approach for testing unless you prepared to act as your own CA provider.

Roll the certificate on the device

Certificates on a device should always be stored in a safe place like a [hardware security module \(HSM\)](#). The way you roll device certificates will depend on how they were created and installed in the devices in the first place.

If you got your certificates from a third party, you must look into how they roll their certificates. The process may be included in your arrangement with them, or it may be a separate service they offer.

If you're managing your own device certificates, you'll have to build your own pipeline for updating certificates. Make sure both old and new leaf certificates have the same common name (CN). By having the same CN, the device can reprovision itself without creating a duplicate registration record.

The mechanics of installing a new certificate on a device will often involve one of the following approaches:

- You can trigger affected devices to send a new certificate signing request (CSR) to your PKI Certificate Authority (CA). In this case, each device will likely be able to download its new device certificate directly from the CA.

- You can retain a CSR from each device and use that to get a new device certificate from the PKI CA. In this case, you'll need to push the new certificate to each device in a firmware update using a secure OTA update service like [Device Update for IoT Hub](#).

Roll the certificate in the IoT hub

The device certificate can be manually added to an IoT hub. The certificate can also be automated using a Device Provisioning Service instance. In this article, we'll assume a Device Provisioning Service instance is being used to support auto-provisioning.

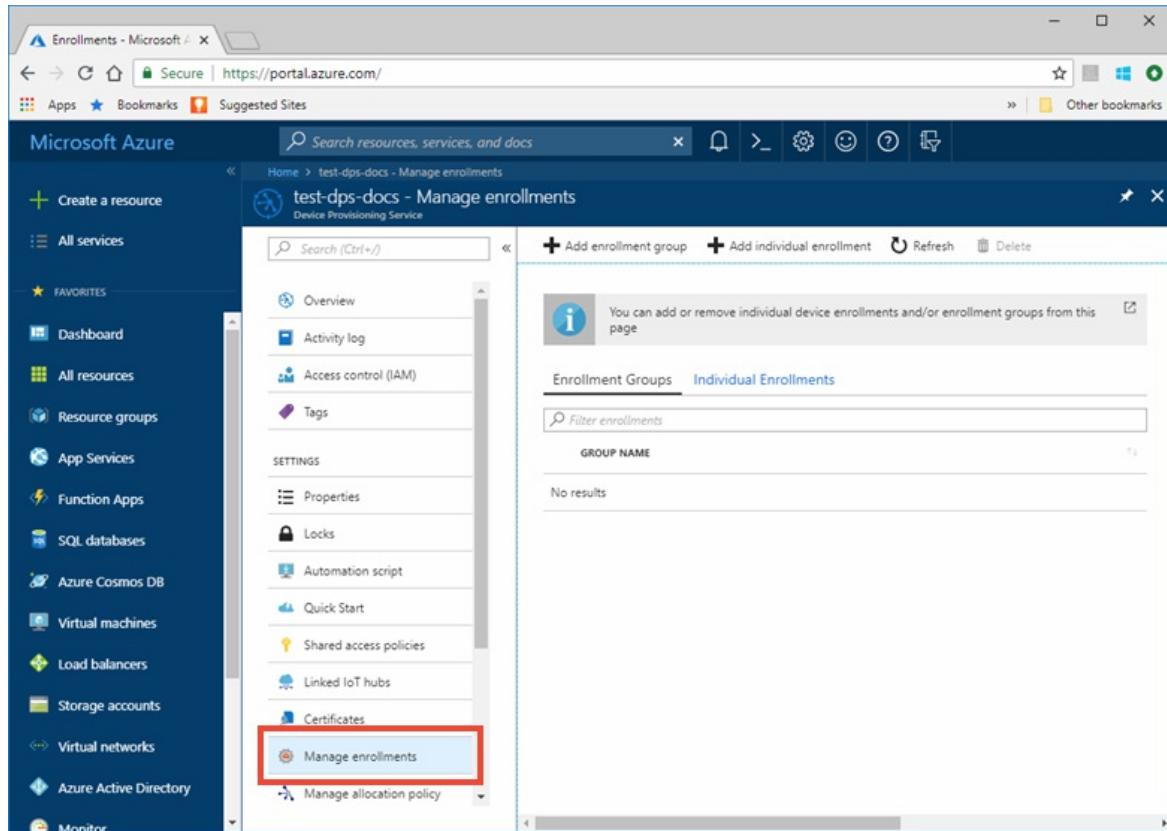
When a device is initially provisioned through auto-provisioning, it boots-up, and contacts the provisioning service. The provisioning service responds by performing an identity check before creating a device identity in an IoT hub using the device's leaf certificate as the credential. The provisioning service then tells the device which IoT hub it's assigned to, and the device then uses its leaf certificate to authenticate and connect to the IoT hub.

Once a new leaf certificate has been rolled to the device, it can no longer connect to the IoT hub because it's using a new certificate to connect. The IoT hub only recognizes the device with the old certificate. The result of the device's connection attempt will be an "unauthorized" connection error. To resolve this error, you must update the enrollment entry for the device to account for the device's new leaf certificate. Then the provisioning service can update the IoT Hub device registry information as needed when the device is reprovisioned.

One possible exception to this connection failure would be a scenario where you've created an [Enrollment Group](#) for your device in the provisioning service. In this case, if you aren't rolling the root or intermediate certificates in the device's certificate chain of trust, then the device will be recognized if the new certificate is part of the chain of trust defined in the enrollment group. If this scenario arises as a reaction to a security breach, you should at least disallow the specific device certificates in the group that are considered to be breached. For more information, see [Disallow specific devices in an enrollment group](#).

Updating enrollment entries for rolled certificates is accomplished on the [Manage enrollments](#) page. To access that page, follow these steps:

1. Sign in to the [Azure portal](#) and navigate to the IoT Hub Device Provisioning Service instance that has the enrollment entry for your device.
2. Click [Manage enrollments](#).



How you handle updating the enrollment entry will depend on whether you're using individual enrollments, or group enrollments. Also the recommended procedures differ depending on whether you're rolling certificates because of a security breach, or certificate expiration. The following sections describe how to handle these updates.

Individual enrollments and security breaches

If you're rolling certificates in response to a security breach, you should use the following approach that deletes the current certificate immediately:

1. Click **Individual Enrollments**, and click the registration ID entry in the list.
2. Click the **Delete current certificate** button and then, click the folder icon to select the new certificate to be uploaded for the enrollment entry. Click **Save** when finished.

These steps should be completed for the primary and secondary certificate, if both are compromised.

- Once the compromised certificate has been removed from the provisioning service, the certificate can still be used to make device connections to the IoT hub as long as a device registration for it exists there. You can address this two ways:

The first way would be to manually navigate to your IoT hub and immediately remove the device registration associated with the compromised certificate. Then when the device provisions again with an updated certificate, a new device registration will be created.

The screenshot shows the 'ExampleIoTHub - IoT devices' blade in the Azure portal. The left sidebar contains navigation links: Overview, Activity log, Access control (IAM), Tags, Events, SETTINGS (Shared access policies, Pricing and scale, Operations monitoring, IP Filter, Certificates, Properties, Locks, Automation script), EXPLORERS (Query explorer, IoT devices). The 'IoT devices' link is highlighted with a red box. The main area has a search bar, 'Add', 'Refresh', and 'Delete' buttons (the 'Delete' button is also highlighted with a red box). A help message says: 'You can use this tool to view, create, update, and delete devices on your IoT Hub.' Below is a query editor with a 'Query' section containing 'SELECT * FROM devices WHERE optional (e.g. tags.location='US')' and an 'Execute' button. A table lists devices with columns: DEVICE ID, STATUS, LAST ACTIVITY, LAST STATUS ..., AUTHENTICA..., CLOUD TO DE... The row for 'AZ3166' is selected and highlighted with a dashed blue border; its status is 'Enabled', last activity is 'Wed Jul 18 ...', authentication is 'SelfSigned', and cloud-to-device messages are '0'. The 'DEVICE ID' column for AZ3166 has a checked checkbox.

The second way would be to use reprovisioning support to reprovision the device to the same IoT hub. This approach can be used to replace the certificate for the device registration on the IoT hub. For more information, see [How to reprovision devices](#).

Individual enrollments and certificate expiration

If you're rolling certificates to handle certificate expirations, you should use the secondary certificate configuration as follows to reduce downtime for devices attempting to provision.

Later when the secondary certificate also nears expiration, and needs to be rolled, you can rotate to using the primary configuration. Rotating between the primary and secondary certificates in this way reduces downtime for devices attempting to provision.

1. Click **Individual Enrollments**, and click the registration ID entry in the list.
2. Click **Secondary Certificate** and then, click the folder icon to select the new certificate to be uploaded for the enrollment entry. Click **Save**.

The screenshot shows the Azure Device Provisioning Service interface. The left sidebar has a navigation menu with items like Overview, Activity log, Access control (IAM), Tags, Properties, Locks, Automation script, Quick Start, Shared access policies, Linked IoT hubs, Certificates, Manage enrollments (which is selected), Manage allocation policy, Metrics (preview), Diagnostics settings, and New support request. The main content area has tabs for Enrollment Groups and Individual Enrollments, with Individual Enrollments selected. It shows a registration ID 'abcdefg'. On the right, there's a 'Secondary Certificate' section with a 'Select a file' input field, which is highlighted with a red box. At the bottom right of the page, there is a large blue 'Save' button.

3. Later when the primary certificate has expired, come back and delete that primary certificate by clicking the **Delete current certificate** button.

Enrollment groups and security breaches

To update a group enrollment in response to a security breach, you should use one of the following approaches that will delete the current root CA, or intermediate certificate immediately.

Update compromised root CA certificates

1. Click the **Certificates** tab for your Device Provisioning Service instance.
2. Click the compromised certificate in the list, and then click the **Delete** button. Confirm the delete by entering the certificate name and click **OK**. Repeat this process for all compromised certificates.

The screenshot shows the 'Certificates' blade for the 'test-dps-docs' instance. On the left, a navigation menu includes 'Certificates' (which is highlighted with a red box), 'Manage enrollments', and other options like 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Properties', 'Locks', 'Automation script', 'Quick Start', 'Shared access policies', and 'Linked IoT hubs'. The main area displays a table with columns 'NAME' and 'STATUS'. A row for 'MyRootCert' is selected and highlighted with a red box. To the right is a detailed view of the certificate named 'MyRootCert', showing fields for Certificate Name (MyRootCert), ETag (ABCDEF...), Subject (O=MSR_TEST, C=US, CN=riot-devic...), Expiry (Wed Dec 31 2036 19:00:00 GMT-0500), Thumbprint (ABCDEFABCDEFABCDEFABC...), Created (Mon Aug 06 2018 04:48:06 GMT-0400), and Updated (Mon Aug 06 2018 04:48:06 GMT-0400). A 'Delete' button is at the top right of this panel.

3. Follow steps outlined in [Configure verified CA certificates](#) to add and verify new root CA certificates.
4. Click the **Manage enrollments** tab for your Device Provisioning Service instance, and click the **Enrollment Groups** list. Click your enrollment group name in the list.
5. Click **CA Certificate**, and select your new root CA certificate. Then click **Save**.

The screenshot shows the 'Enrollment Group Details' blade for the 'example' group. At the top, there are 'Save' and 'Refresh' buttons, with 'Save' highlighted with a red box. Below are tabs for 'Settings' (selected) and 'Registration Records'. A large information icon provides general guidance. Under 'Identity Attestation Information', there's an 'x509' section. In the 'Certificate Type' section, 'CA Certificate' is selected (highlighted with a red box). The 'Primary Certificate' dropdown is open, showing 'NewRootCA' (highlighted with a red box), 'No certificate selected', and 'NewRootCA' again. Further down are sections for 'Desired IoT Hub' (set to 'Assign automatically') and 'Enable entry' (with 'Enable' and 'Disable' buttons).

6. Once the compromised certificate has been removed from the provisioning service, the certificate can still be used to make device connections to the IoT hub as long as device registrations for it exists there. You can address this two ways:

The first way would be to manually navigate to your IoT hub and immediately remove the device registration associated with the compromised certificate. Then when your devices provision again with updated certificates, a new device registration will be created for each one.

The screenshot shows the Azure IoT Hub - IoT devices blade. The left sidebar contains navigation links: Overview, Activity log, Access control (IAM), Tags, Events, SETTINGS (Shared access policies, Pricing and scale, Operations monitoring, IP Filter, Certificates, Properties, Locks, Automation script), EXPLORERS (Query explorer, IoT devices). The IoT devices link is highlighted with a red box. The main area has a search bar, Add, Refresh, and Delete buttons (the Delete button is also highlighted with a red box). A message box says: "You can use this tool to view, create, update, and delete devices on your IoT Hub." Below is a query editor with a SELECT statement and an Execute button. A table lists device details: DEVICE ID (AZ3166), STATUS (Enabled), LAST ACTIVITY (Wed Jul 18 ...), AUTHENTICA... (SelfSigned), CLOUD TO DE... (0). The AZ3166 row is highlighted with a dashed blue border and a red box.

The second way would be to use reprovisioning support to reprovision your devices to the same IoT hub. This approach can be used to replace certificates for device registrations on the IoT hub. For more information, see [How to reprovision devices](#).

Update compromised intermediate certificates

1. Click **Enrollment Groups**, and then click the group name in the list.
2. Click **Intermediate Certificate**, and **Delete current certificate**. Click the folder icon to navigate to the new intermediate certificate to be uploaded for the enrollment group. Click **Save** when you're finished. These steps should be completed for both the primary and secondary certificate, if both are compromised.

This new intermediate certificate should be signed by a verified root CA certificate that has already been added into provisioning service. For more information, see [X.509 certificates](#).

The screenshot shows the 'Enrollment Group Details' page in the Azure portal. At the top, there's a navigation bar with a back arrow, a search bar, and a user profile. Below it, there are two tabs: 'Settings' (selected) and 'Registration Records'. A large information card with a blue 'i' icon says: 'You can view and update attestation information, set desired IoT Hub, and set the initial twin state of provisioning devices'. Under 'Identity Attestation Information', it says 'x509'. A section for 'Certificate Type' has two options: 'CA Certificate' and 'Intermediate Certificate', with 'Intermediate Certificate' selected. Below that, a 'Primary Certificate' section is expanded, showing a 'Delete current certificate' button. To the right of the certificate details is a 'Select a file' input field with a file icon, which is highlighted with a red box. A 'Secondary Certificate' section is also present. At the bottom, there are 'Save' and 'Refresh' buttons.

3. Once the compromised certificate has been removed from the provisioning service, the certificate can still be used to make device connections to the IoT hub as long as device registrations for it exists there. You can address this two ways:

The first way would be to manually navigate to your IoT hub and immediately remove the device registration associated with the compromised certificate. Then when your devices provision again with updated certificates, a new device registration will be created for each one.

The screenshot shows the 'ExampleIoTHub - IoT devices' blade in the Azure portal. On the left, a navigation menu includes 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Events', 'SETTINGS' (with sub-options like 'Shared access policies', 'Pricing and scale', 'Operations monitoring', 'IP Filter', 'Certificates', 'Properties', 'Locks', and 'Automation script'), 'EXPLORERS' (with 'Query explorer' and 'IoT devices' selected), and 'Cloud to Device'. At the top right, there are 'Add', 'Refresh', and 'Delete' buttons, with 'Delete' highlighted by a red box. Below these are informational cards: one about device management and another for a database query. A table lists devices with columns: DEVICE ID, STATUS, LAST ACTIVITY, LAST STATUS ..., AUTHENTICA..., and CLOUD TO DE...; the first device, AZ3166, is selected and highlighted with a dashed blue border.

The second way would be to use reprovisioning support to reprovision your devices to the same IoT hub. This approach can be used to replace certificates for device registrations on the IoT hub. For more information, see [How to reprovision devices](#).

Enrollment groups and certificate expiration

If you are rolling certificates to handle certificate expirations, you should use the secondary certificate configuration as follows to ensure no downtime for devices attempting to provision.

Later when the secondary certificate also nears expiration, and needs to be rolled, you can rotate to using the primary configuration. Rotating between the primary and secondary certificates in this way ensures no downtime for devices attempting to provision.

Update expiring root CA certificates

1. Follow steps outlined in [Configure verified CA certificates](#) to add and verify new root CA certificates.
2. Click the **Manage enrollments** tab for your Device Provisioning Service instance, and click the **Enrollment Groups** list. Click your enrollment group name in the list.
3. Click **CA Certificate**, and select your new root CA certificate under the **Secondary Certificate** configuration. Then click **Save**.

Save Refresh

Settings Registration Records

You can view and update attestation information, set desired IoT Hub, and set the initial twin state of provisioning devices

Identity Attestation Information
x509

Certificate Type CA Certificate Intermediate Certificate

Primary Certificate ExpiringCert

Secondary Certificate No certificate selected
No certificate selected
ExpiringCert
NewRootCA
Assign automatically

Enable entry Enable Disable

- Later when the primary certificate has expired, click the **Certificates** tab for your Device Provisioning Service instance. Click the expired certificate in the list, and then click the **Delete** button. Confirm the delete by entering the certificate name, and click **OK**.

Home > test-dps-docs - Certificates

test-dps-docs - Certificates

Device Provisioning Service

Search (Ctrl+ /)

Add Columns Refresh

Overview Activity log Access control (IAM) Tags

SETTINGS

Properties Locks Automation script Quick Start Shared access policies Linked IoT hubs Certificates Manage enrollments

Certificates

Delete

Certificate Details

MyRootCert

NAME STATUS

MyRootCert Unverified

ETag ABCDEFGHIJK=

Subject O=MSR_TEST, C=US, CN=riot-device...
Expiry Wed Dec 31 2036 19:00:00 GMT-0500 (Eastern Standard Time)
Thumbprint ABCDEFABCDEFABCDEFABC...
Created Mon Aug 06 2018 04:48:06 GMT-0400 (Eastern Daylight Time)
Updated Mon Aug 06 2018 04:48:06 GMT-0400 (Eastern Daylight Time)

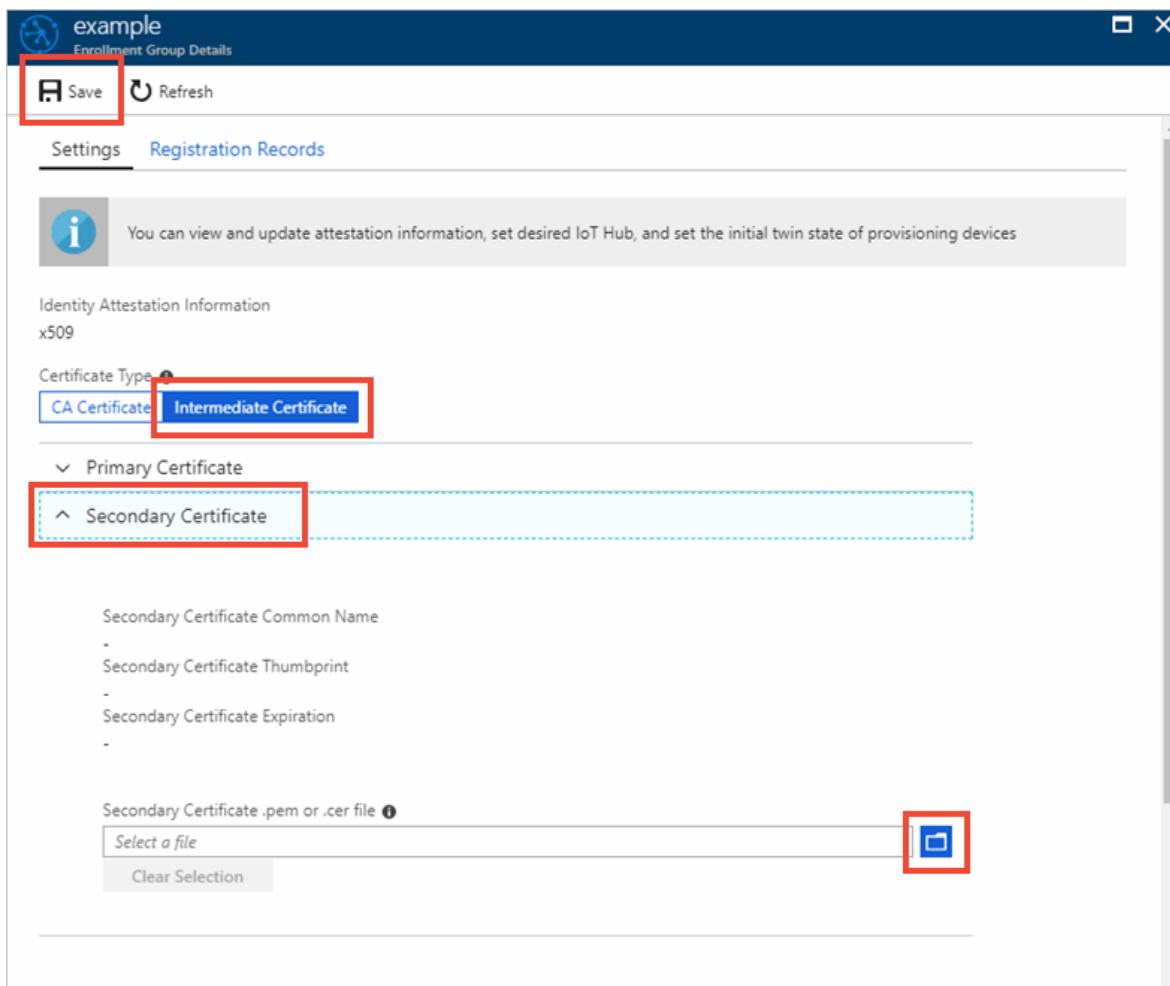
Verify

Update expiring intermediate certificates

- Click **Enrollment Groups**, and click the group name in the list.

2. Click **Secondary Certificate** and then, click the folder icon to select the new certificate to be uploaded for the enrollment entry. Click **Save**.

This new intermediate certificate should be signed by a verified root CA certificate that has already been added into provisioning service. For more information, see [X.509 certificates](#).



3. Later when the primary certificate has expired, come back and delete that primary certificate by clicking the **Delete current certificate** button.

Reprovision the device

Once the certificate is rolled on both the device and the Device Provisioning Service, the device can reprovision itself by contacting the Device Provisioning Service.

One easy way of programming devices to reprovision is to program the device to contact the provisioning service to go through the provisioning flow if the device receives an "unauthorized" error from attempting to connect to the IoT hub.

Another way is for both the old and the new certificates to be valid for a short overlap, and use the IoT hub to send a command to devices to have them re-register via the provisioning service to update their IoT Hub connection information. Because each device can process commands differently, you will have to program your device to know what to do when the command is invoked. There are several ways you can command your device via IoT Hub, and we recommend using [direct methods](#) or [jobs](#) to initiate the process.

Once reprovisioning is complete, devices will be able to connect to IoT Hub using their new certificates.

Disallow certificates

In response to a security breach, you may need to disallow a device certificate. To disallow a device certificate,

disable the enrollment entry for the target device/certificate. For more information, see disallowing devices in the [Manage disenrollment](#) article.

Once a certificate is included as part of a disabled enrollment entry, any attempts to register with an IoT hub using that certificates will fail even if it is enabled as part of another enrollment entry.

Next steps

- To learn more about X.509 certificates in the Device Provisioning Service, see [X.509 certificate attestation](#)
- To learn about how to do proof-of-possession for X.509 CA certificates with the Azure IoT Hub Device Provisioning Service, see [How to verify certificates](#)
- To learn about how to use the portal to create an enrollment group, see [Managing device enrollments with Azure portal](#).

How to reprovision devices

8/22/2022 • 5 minutes to read • [Edit Online](#)

During the lifecycle of an IoT solution, it is common to move devices between IoT hubs. This topic is written to assist solution operators configuring reprovisioning policies.

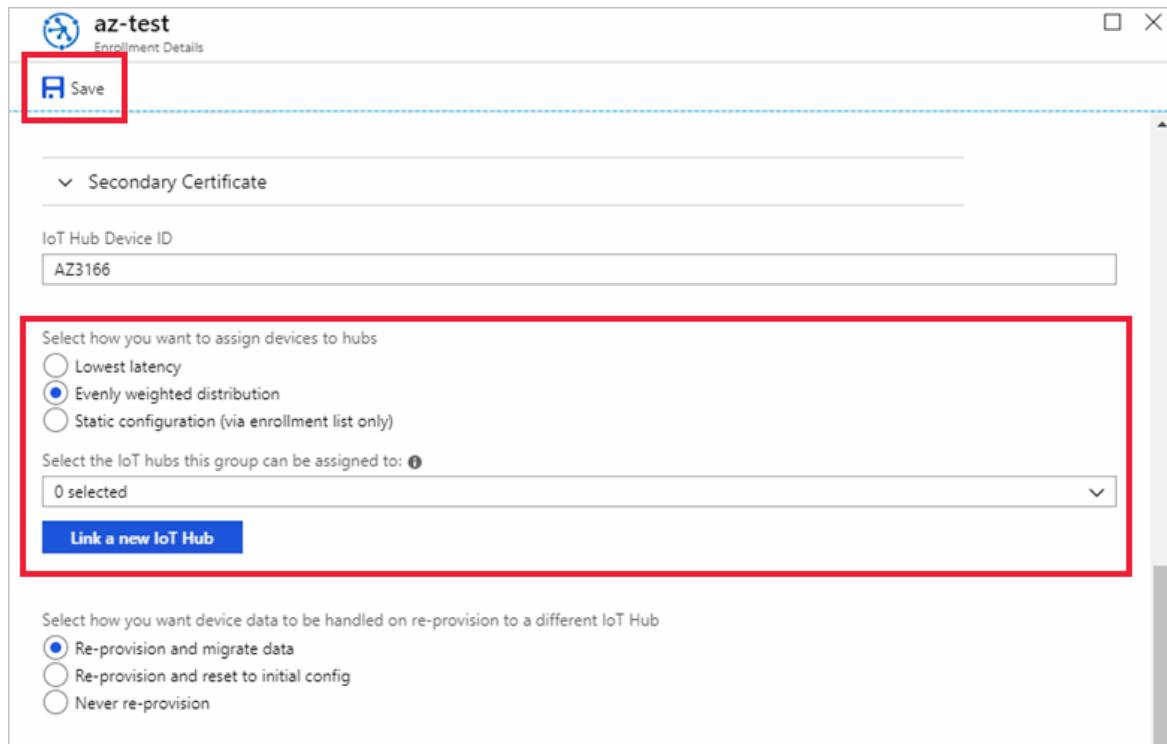
For more a more detailed overview of reprovisioning scenarios, see [IoT Hub Device reprovisioning concepts](#).

Configure the enrollment allocation policy

The allocation policy determines how the devices associated with the enrollment will be allocated, or assigned, to an IoT hub once reprovisioned.

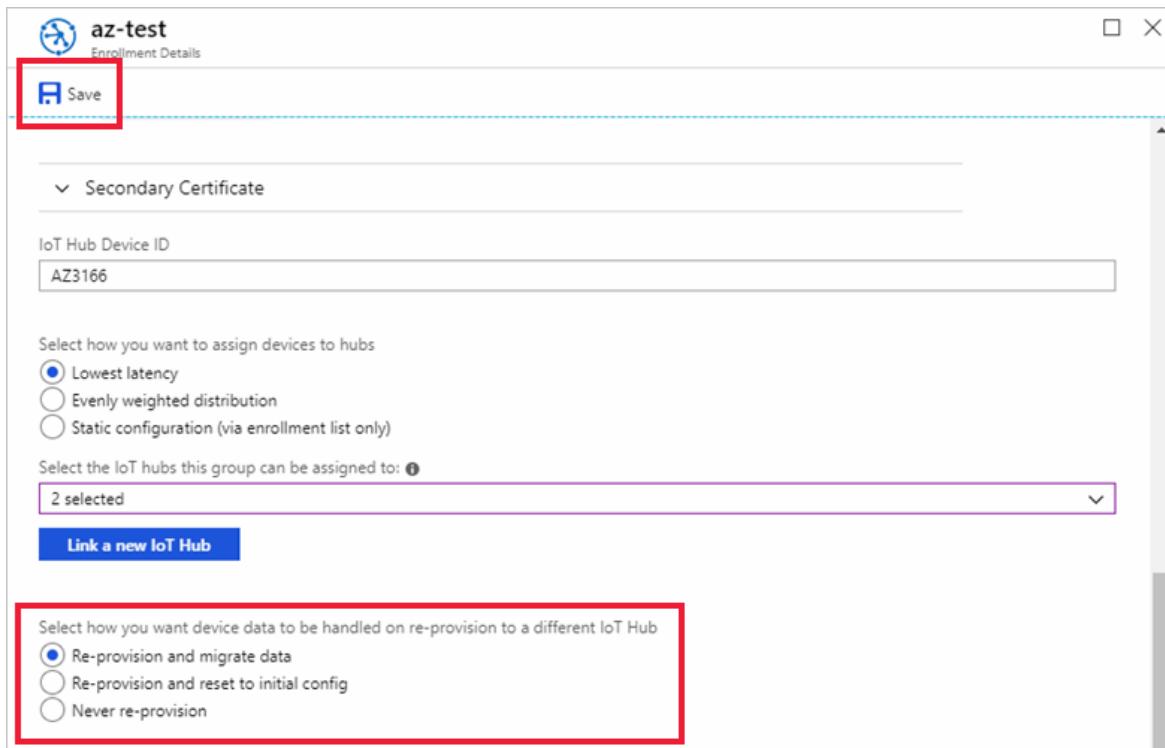
The following steps configure the allocation policy for a device's enrollment:

1. Sign in to the [Azure portal](#) and navigate to your Device Provisioning Service instance.
 2. Click **Manage enrollments**, and click the enrollment group or individual enrollment that you want to configure for reprovisioning.
 3. Under **Select how you want to assign devices to hubs**, select one of the following allocation policies:
 - **Lowest latency**: This policy assigns devices to the linked IoT Hub that will result in the lowest latency communications between device and IoT Hub. This option enables the device to communicate with the closest IoT hub based on location.
 - **Evenly weighted distribution**: This policy distributes devices across the linked IoT Hubs based on the allocation weight assigned to each linked IoT hub. This policy allows you to load balance devices across a group of linked hubs based on the allocation weights set on those hubs. If you are provisioning devices to only one IoT Hub, we recommend this setting. This setting is the default.
 - **Static configuration**: This policy requires a desired IoT Hub be listed in the enrollment entry for a device to be provisioned. This policy allows you to designate a single specific IoT hub that you want to assign devices to.
 4. Under **Select the IoT hubs this group can be assigned to**, select the linked IoT hubs that you want included with your allocation policy. Optionally, add a new linked IoT hub using the **Link a new IoT Hub** button.
- With the **Lowest latency** allocation policy, the hubs you select will be included in the latency evaluation to determine the closest hub for device assignment.
- With the **Evenly weighted distribution** allocation policy, devices will be load balanced across the hubs you select based on their configured allocation weights and their current device load.
- With the **Static configuration** allocation policy, select the IoT hub you want devices assigned to.
5. Click **Save**, or proceed to the next section to set the reprovisioning policy.



Set the reprovisioning policy

1. Sign in to the [Azure portal](#) and navigate to your Device Provisioning Service instance.
2. Click **Manage enrollments**, and click the enrollment group or individual enrollment that you want to configure for reprovisioning.
3. Under **Select how you want device data to be handled on re-provision to a different IoT hub**, choose one of the following reprovisioning policies:
 - **Re-provision and migrate data:** This policy takes action when devices associated with the enrollment entry submit a new provisioning request. Depending on the enrollment entry configuration, the device may be reassigned to another IoT hub. If the device is changing IoT hubs, the device registration with the initial IoT hub will be removed. All device state information from that initial IoT hub will be migrated over to the new IoT hub. During migration, the device's status will be reported as **Assigning**.
 - **Re-provision and reset to initial config:** This policy takes action when devices associated with the enrollment entry submit a new provisioning request. Depending on the enrollment entry configuration, the device may be reassigned to another IoT hub. If the device is changing IoT hubs, the device registration with the initial IoT hub will be removed. The initial configuration data that the provisioning service instance received when the device was provisioned is provided to the new IoT hub. During migration, the device's status will be reported as **Assigning**.
4. Click **Save** to enable the reprovisioning of the device based on your changes.



Send a provisioning request from the device

In order for devices to be reprovisioned based on the configuration changes made in the preceding sections, these devices must request reprovisioning.

How often a device submits a provisioning request depends on the scenario. When designing your solution and defining a reprovisioning logic there are a few things to consider. For example:

- How often you expect your devices to restart
- The [DPS quotas and limits](#)
- Expected deployment time for your fleet (phased rollout vs all at once)
- Retry capability implemented on your client code, as described on the [Retry general guidance](#) at the Azure Architecture Center

TIP

We recommend not provisioning on every reboot of the device, as this could hit the service throttling limits especially when reprovisioning several thousands or millions of devices at once. Instead you should attempt to use the [Device Registration Status Lookup API](#) and try to connect with that information to IoT Hub. If that fails, then try to reprovision as the IoT Hub information might have changed. Keep in mind that querying for the registration state will count as a new device registration, so you should consider the [Device registration limit](#). Also consider implementing an appropriate retry logic, such as exponential back-off with randomization, as described on the [Retry general guidance](#). In some cases, depending on the device capabilities, it's possible to save the IoT Hub information directly on the device to connect directly to IoT Hub after the first-time provisioning using DPS occurred. If you choose to do this, make sure you implement a fallback mechanism in case you get specific [errors from Hub occur](#), for example, consider the following scenarios:

- Retry the Hub operation if the result code is 429 (Too Many Requests) or an error in the 5xx range. Do not retry for any other errors.
- For 429 errors, only retry after the time indicated in the `Retry-After` header.
- For 5xx errors, use exponential back-off, with the first retry at least 5 seconds after the response.
- On errors other than 429 and 5xx, re-register through DPS
- Ideally you should also support a `method` to manually trigger provisioning on demand.

We also recommend taking into account the service limits when planning activities like pushing updates to your fleet. For example, updating the fleet all at once could cause all devices to re-register through DPS (which could easily be above the registration quota limit) - For such scenarios, consider planning for device updates in phases instead of updating your entire fleet at the same time.

Next steps

- To learn more Reprovisioning, see [IoT Hub Device reprovisioning concepts](#)
- To learn more Deprovisioning, see [How to deprovision devices that were previously auto-provisioned](#)

How to disenroll a device from Azure IoT Hub Device Provisioning Service

8/22/2022 • 7 minutes to read • [Edit Online](#)

Proper management of device credentials is crucial for high-profile systems like IoT solutions. A best practice for such systems is to have a clear plan of how to revoke access for devices when their credentials, whether a shared access signatures (SAS) token or an X.509 certificate, might be compromised.

Enrollment in the Device Provisioning Service enables a device to be [provisioned](#). A provisioned device is one that has been registered with IoT Hub, allowing it to receive its initial [device twin](#) state and begin reporting telemetry data. This article describes how to disenroll a device from your provisioning service instance, preventing it from being provisioned again in the future. To learn how to deprovision a device that has already been provisioned to an IoT hub, see [Manage deprovisioning](#).

NOTE

Be aware of the retry policy of devices that you revoke access for. For example, a device that has an infinite retry policy might continuously try to register with the provisioning service. That situation consumes service resources and possibly affects performance.

Disallow devices by using an individual enrollment entry

Individual enrollments apply to a single device and can use X.509 certificates, TPM endorsement keys (in a real or virtual TPM), or SAS tokens as the attestation mechanism. To disallow a device that has an individual enrollment, you can either disable or delete its enrollment entry.

To temporarily disallow the device by disabling its enrollment entry:

1. Sign in to the Azure portal and select **All resources** from the left menu.
2. In the list of resources, select the provisioning service that you want to disallow your device from.
3. In your provisioning service, select **Manage enrollments**, and then select the **Individual Enrollments** tab.
4. Select the enrollment entry for the device that you want to disallow.

The screenshot shows the Microsoft Azure portal with the URL <https://portal.azure.com/>. The page title is "Enrollments - Microsoft". The left sidebar shows a navigation tree for "test-dps-docs - Manage enrollments", with "Manage enrollments" highlighted with a red box. The main content area displays the "Individual Enrollments" tab, which is also highlighted with a red box. A message box states: "You can add or remove individual device enrollments and/or enrollment groups from this page". Below this, there is a table with one row, where the "REGISTRATION ID" column contains the value "riot-device-cert", which is also highlighted with a red box.

5. On your enrollment page, scroll to the bottom, and select **Disable** for the **Enable entry** switch, and then select **Save**.

The screenshot shows the "Enrollment Details" page for "riot-device-cert". The "Save" button at the top left is highlighted with a red box. Below it, there are sections for assigning devices to IoT hubs and handling device data re-provisioning. At the bottom, there is an "Initial Device Twin State" editor with JSON code, and an "Enable entry" switch with two options: "Enable" and "Disable", where "Disable" is highlighted with a red box.

To permanently disallow the device by deleting its enrollment entry:

1. Sign in to the Azure portal and select **All resources** from the left menu.
2. In the list of resources, select the provisioning service that you want to disallow your device from.
3. In your provisioning service, select **Manage enrollments**, and then select the **Individual Enrollments** tab.
4. Select the check box next to the enrollment entry for the device that you want to disallow.
5. Select **Delete** at the top of the window, and then select **Yes** to confirm that you want to remove the enrollment.

The screenshot shows the 'test-dps-docs - Manage enrollments' page. On the left sidebar, under 'Settings', the 'Manage enrollments' option is highlighted with a red box. The main area shows a list of individual enrollments. One entry, 'riot-device-cert', has a checked checkbox. A red box highlights this checkbox. At the top right of the main area, there is a 'Delete' button, which is also highlighted with a red box.

After you finish the procedure, you should see your entry removed from the list of individual enrollments.

Disallow an X.509 intermediate or root CA certificate by using an enrollment group

X.509 certificates are typically arranged in a certificate chain of trust. If a certificate at any stage in a chain becomes compromised, trust is broken. The certificate must be disallowed to prevent Device Provisioning Service from provisioning devices downstream in any chain that contains that certificate. To learn more about X.509 certificates and how they are used with the provisioning service, see [X.509 certificates](#).

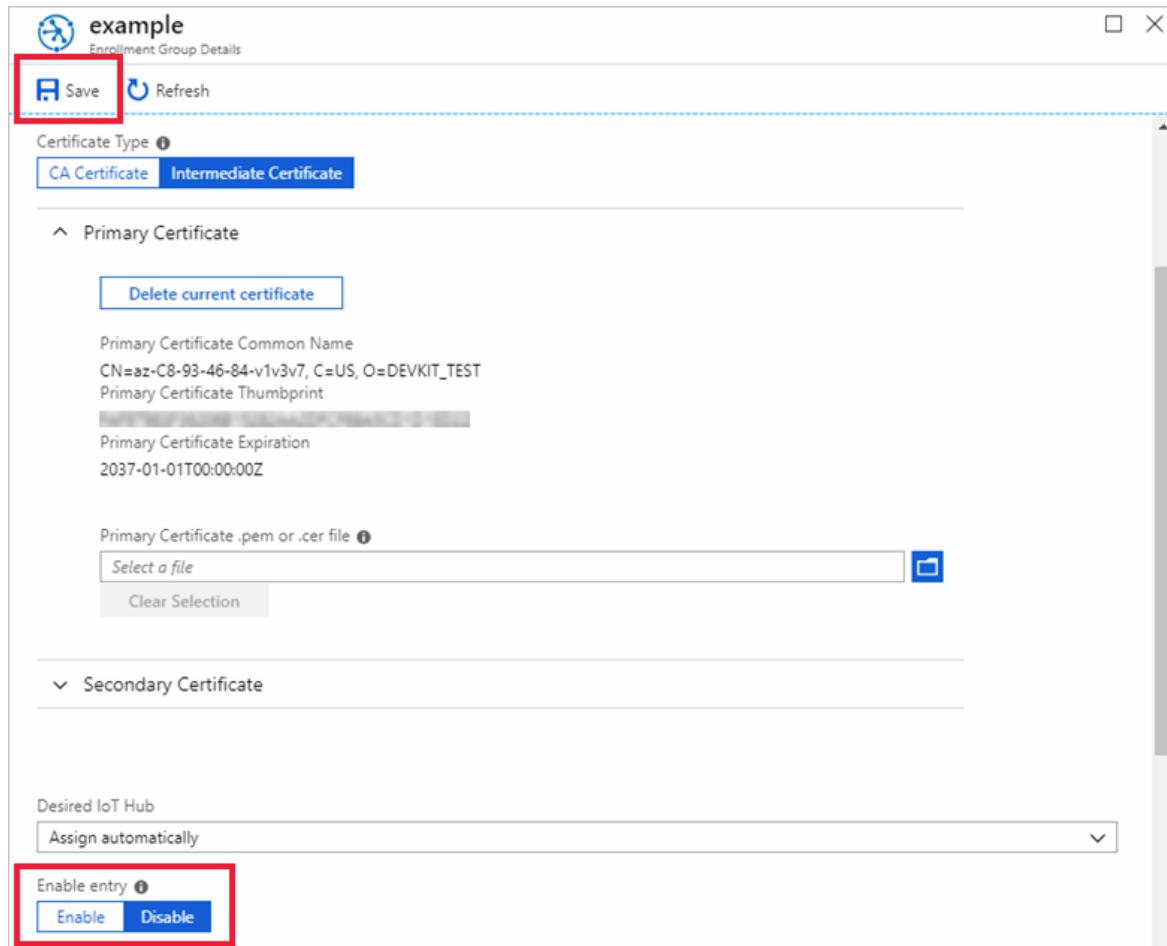
An enrollment group is an entry for devices that share a common attestation mechanism of X.509 certificates signed by the same intermediate or root CA. The enrollment group entry is configured with the X.509 certificate associated with the intermediate or root CA. The entry is also configured with any configuration values, such as twin state and IoT hub connection, that are shared by devices with that certificate in their certificate chain. To disallow the certificate, you can either disable or delete its enrollment group.

To temporarily disallow the certificate by disabling its enrollment group:

1. Sign in to the Azure portal and select **All resources** from the left menu.
2. In the list of resources, select the provisioning service that you want to disallow the signing certificate

from.

3. In your provisioning service, select **Manage enrollments**, and then select the **Enrollment Groups** tab.
4. Select the enrollment group using the certificate that you want to disallow.
5. Select **Disable** on the **Enable entry** switch, and then select **Save**.



To permanently disallow the certificate by deleting its enrollment group:

1. Sign in to the Azure portal and select **All resources** from the left menu.
2. In the list of resources, select the provisioning service that you want to disallow your device from.
3. In your provisioning service, select **Manage enrollments**, and then select the **Enrollment Groups** tab.
4. Select the check box next to the enrollment group for the certificate that you want to disallow.
5. Select **Delete** at the top of the window, and then select **Yes** to confirm that you want to remove the enrollment group.

The screenshot shows the 'test-dps-docs - Manage enrollments' page in the Azure portal. On the left, a sidebar lists various service management options like Overview, Activity log, Access control (IAM), Tags, Properties, Locks, Automation script, Quick Start, Shared access policies, Linked IoT hubs, Certificates, and Manage enrollments. The 'Manage enrollments' option is highlighted with a red box. The main content area has a search bar at the top. Below it, there are two tabs: 'Enrollment Groups' (which is underlined and highlighted with a red box) and 'Individual Enrollments'. A large blue button labeled 'Add enrollment group' is visible. At the top right, there are buttons for 'Add individual enrollment', 'Refresh', and 'Delete', with the 'Delete' button also highlighted with a red box. A message box in the center says: 'You can add or remove individual device enrollments and/or enrollment groups from this page'. Below the message, there's a table header 'GROUP NAME' with a row containing a checked checkbox and the name 'example'.

After you finish the procedure, you should see your entry removed from the list of enrollment groups.

NOTE

If you delete an enrollment group for a certificate, devices that have the certificate in their certificate chain might still be able to enroll if an enabled enrollment group for the root certificate or another intermediate certificate higher up in their certificate chain exists.

NOTE

Deleting an enrollment group doesn't delete the registration records for devices in the group. DPS uses the registration records to determine whether the maximum number of registrations has been reached for the DPS instance. Orphaned registration records still count against this quota. For the current maximum number of registrations supported for a DPS instance, see [Quotas and limits](#).

You may want to delete the registration records for the enrollment group before deleting the enrollment group itself. You can see and manage the registration records for an enrollment group manually on the **Registration Records** tab for the group in Azure portal. You can retrieve and manage the registration records programmatically using the [Device Registration State REST APIs](#) or equivalent APIs in the [DPS service SDKs](#), or using the [az iot dps enrollment-group registration](#) Azure CLI commands.

Disallow specific devices in an enrollment group

Devices that implement the X.509 attestation mechanism use the device's certificate chain and private key to authenticate. When a device connects and authenticates with Device Provisioning Service, the service first looks for an individual enrollment with a registration ID that matches the common name (CN) of the device (end-entity) certificate. The service then searches enrollment groups to determine whether the device can be provisioned. If the service finds a disabled individual enrollment for the device, it prevents the device from connecting. The service prevents the connection even if an enabled enrollment group for an intermediate or root CA in the device's certificate chain exists.

To disallow an individual device in an enrollment group, follow these steps:

1. Sign in to the Azure portal and select **All resources** from the left menu.
2. From the list of resources, select the provisioning service that contains the enrollment group for the device that you want to disallow.
3. In your provisioning service, select **Manage enrollments**, and then select the **Individual Enrollments** tab.
4. Select the **Add individual enrollment** button at the top.
5. Follow the appropriate step depending on whether you have the device (end-entity) certificate.

- If you have the device certificate, on the **Add Enrollment** page select:

Mechanism: X.509

Primary .pem or .cer file: Upload the device certificate. For the certificate, use the signed end-entity certificate installed on the device. The device uses the signed end-entity certificate for authentication.

IoT Hub Device ID: Leave this blank. For devices provisioned through X.509 enrollment groups, the device ID is set by the device certificate CN and is the same as the registration ID.

The screenshot shows the 'Add Enrollment' form in the Azure portal. The 'Mechanism' dropdown is set to 'X.509'. The 'Primary Certificate .pem or .cer file' input field contains 'X509testcert.pem' and has a 'Clear Selection' button below it. The 'Secondary Certificate .pem or .cer file' input field has a 'Select a file' placeholder and a 'Clear Selection' button. The 'IoT Hub Device ID' input field is empty and labeled 'Device ID'. The 'IoT Edge device' toggle switch is set to 'False'.

- If you don't have the device certificate, on the **Add Enrollment** page select:

Mechanism: Symmetric Key

Auto-generate keys: Make sure this is selected. The keys don't matter for this scenario.

Registration ID: If the device has already been provisioned, use its IoT Hub device ID. You can find this in the registration records of the enrollment group, or in the IoT hub that the device was provisioned to. If the device has not yet been provisioned, enter the device certificate CN. (In this latter case, you don't need the device certificate, but you will need to know the CN.)

IoT Hub Device ID: Leave this blank. For devices provisioned through X.509 enrollment groups, the device ID is set by the device certificate CN and is the same as the registration ID.

Add Enrollment

Save

Mechanism * ⓘ
Symmetric Key

Auto-generate keys ⓘ

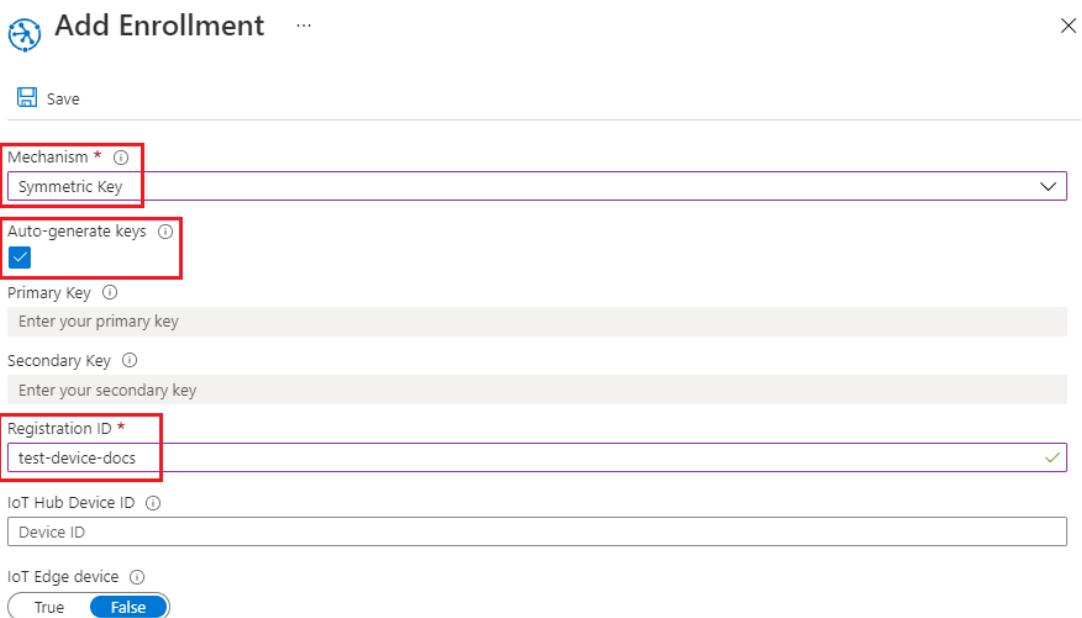
Primary Key ⓘ
Enter your primary key

Secondary Key ⓘ
Enter your secondary key

Registration ID * ⓘ
test-device-docs

IoT Hub Device ID ⓘ
Device ID

IoT Edge device ⓘ
 True False



6. Scroll to the bottom of the **Add Enrollment** page and select **Disable** on the **Enable entry** switch, and then select **Save**.

Add Enrollment

Save

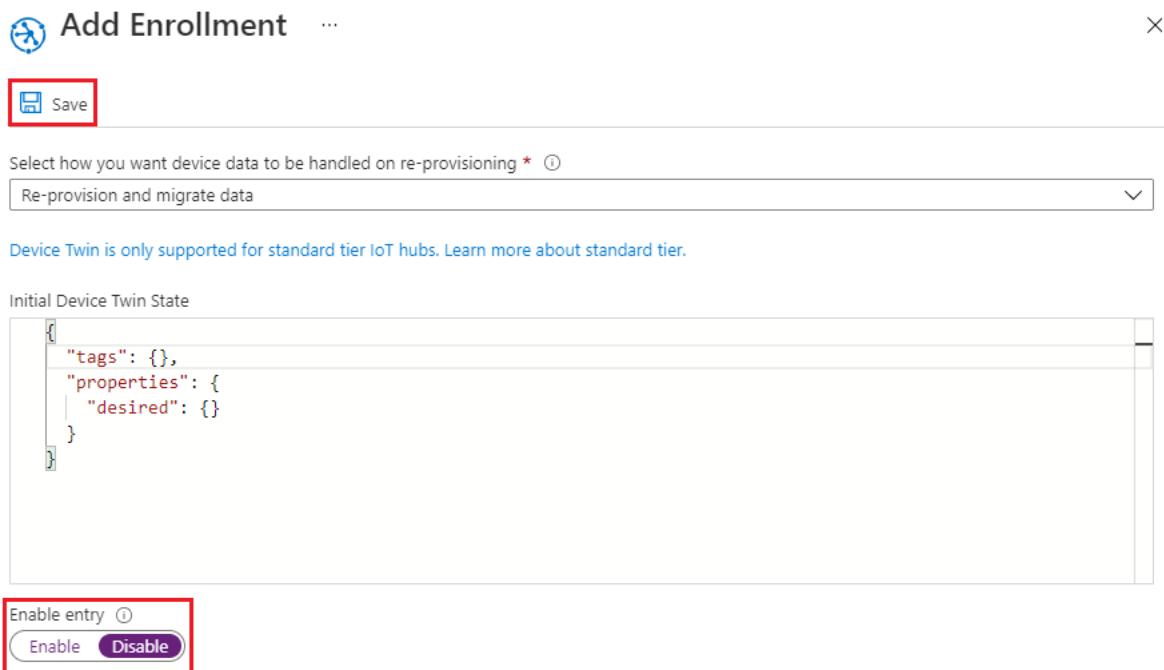
Select how you want device data to be handled on re-provisioning * ⓘ
Re-provision and migrate data

Device Twin is only supported for standard tier IoT hubs. Learn more about standard tier.

Initial Device Twin State

```
{  
  "tags": {},  
  "properties": {  
    "desired": {}  
  }  
}
```

Enable entry ⓘ
 Enable Disable



When you successfully create your enrollment, you should see your disabled device enrollment listed on the **Individual Enrollments** tab.

Next steps

Disenrollment is also part of the larger deprovisioning process. Deprovisioning a device includes both disenrollment from the provisioning service, and deregistering from IoT hub. To learn about the full process, see [How to deprovision devices that were previously auto-provisioned](#)

How to deprovision devices that were previously auto-provisioned

8/22/2022 • 4 minutes to read • [Edit Online](#)

You may find it necessary to deprovision devices that were previously auto-provisioned through the Device Provisioning Service. For example, a device may be sold or moved to a different IoT hub, or it may be lost, stolen, or otherwise compromised.

In general, deprovisioning a device involves two steps:

1. Disenroll the device from your provisioning service, to prevent future auto-provisioning. Depending on whether you want to revoke access temporarily or permanently, you may want to either disable or delete an enrollment entry. For devices that use X.509 attestation, you may want to disable/delete an entry in the hierarchy of your existing enrollment groups.
 - To learn how to disenroll a device, see [How to disenroll a device from Azure IoT Hub Device Provisioning Service](#).
 - To learn how to disenroll a device programmatically using one of the provisioning service SDKs, see [Manage device enrollments with service SDKs](#).
2. Deregister the device from your IoT Hub, to prevent future communications and data transfer. Again, you can temporarily disable or permanently delete the device's entry in the identity registry for the IoT Hub where it was provisioned. See [Disable devices](#) to learn more about disablement. See "Device Management / IoT Devices" for your IoT Hub resource, in the [Azure portal](#).

The exact steps you take to deprovision a device depend on its attestation mechanism and its applicable enrollment entry with your provisioning service. The following sections provide an overview of the process, based on the enrollment and attestation type.

Individual enrollments

Devices that use TPM attestation or X.509 attestation with a leaf certificate are provisioned through an individual enrollment entry.

To deprovision a device that has an individual enrollment:

1. Disenroll the device from your provisioning service:
 - For devices that use TPM attestation, delete the individual enrollment entry to permanently revoke the device's access to the provisioning service, or disable the entry to temporarily revoke its access.
 - For devices that use X.509 attestation, you can either delete or disable the entry. Be aware, though, if you delete an individual enrollment for a device that uses X.509 and an enabled enrollment group exists for a signing certificate in that device's certificate chain, the device can re-enroll. For such devices, it may be safer to disable the enrollment entry. Doing so prevents the device from re-enrolling, regardless of whether an enabled enrollment group exists for one of its signing certificates.
2. Disable or delete the device in the identity registry of the IoT hub that it was provisioned to.

Enrollment groups

With X.509 attestation, devices can also be provisioned through an enrollment group. Enrollment groups are configured with a signing certificate, either an intermediate or root CA certificate, and control access to the

provisioning service for devices with that certificate in their certificate chain. To learn more about enrollment groups and X.509 certificates with the provisioning service, see [X.509 certificate attestation](#).

To see a list of devices that have been provisioned through an enrollment group, you can view the enrollment group's details. This is an easy way to understand which IoT hub each device has been provisioned to. To view the device list:

1. Log in to the Azure portal and select **All resources** on the left-hand menu.
2. Select your provisioning service in the list of resources.
3. In your provisioning service, select **Manage enrollments**, then select the **Enrollment Groups** tab.
4. Select the enrollment group to open it.
5. Select the **Registration Records** tab to view the registration records for the enrollment group.

The screenshot shows the 'test-enrollment-group' page in the Azure portal. At the top, there are buttons for 'Refresh' and 'Delete Registrations'. Below these are two tabs: 'Settings' and 'Registration Records', with 'Registration Records' being the active tab and highlighted with a red box. A tooltip message above the table says: 'You can view devices that have provisioned via this enrollment group and remove the registration records for previously provisioned devices'. Below the table, there is a search bar labeled 'Search devices in this enrollment group'.

Device Id	Assigned IoT Hub	Registration Date
custom-hsm-device-01	MyExampleHub.azure-devices.net	2021-10-06T03:51:16.6849749Z

With enrollment groups, there are two scenarios to consider:

- To deprovision all of the devices that have been provisioned through an enrollment group:
 1. Disable the enrollment group to disallow its signing certificate.
 2. Use the list of provisioned devices for that enrollment group to disable or delete each device from the identity registry of its respective IoT hub.
 3. After disabling or deleting all devices from their respective IoT hubs, you can optionally delete the enrollment group. Be aware, though, that if you delete the enrollment group and there is an enabled enrollment group for a signing certificate higher up in the certificate chain of one or more of the devices, those devices can re-enroll.

NOTE

Deleting an enrollment group doesn't delete the registration records for devices in the group. DPS uses the registration records to determine whether the maximum number of registrations has been reached for the DPS instance. Orphaned registration records still count against this quota. For the current maximum number of registrations supported for a DPS instance, see [Quotas and limits](#).

You may want to delete the registration records for the enrollment group before deleting the enrollment group itself. You can see and manage the registration records for an enrollment group manually on the **Registration Records** tab for the group in Azure portal. You can retrieve and manage the registration records programmatically using the [Device Registration State REST APIs](#) or equivalent APIs in the [DPS service SDKs](#), or using the [az iot dps enrollment-group registration Azure CLI commands](#).

- To deprovision a single device from an enrollment group:
 1. Create a disabled individual enrollment for the device.

- If you have the device (end-entity) certificate, you can create a disabled X.509 individual enrollment.
- If you don't have the device certificate, you can create a disabled symmetric key individual enrollment based on the device ID in the registration record for that device.

To learn more, see [Disallow specific devices in an enrollment group](#).

The presence of a disabled individual enrollment for a device revokes access to the provisioning service for that device while still permitting access for other devices that have the enrollment group's signing certificate in their chain. Do not delete the disabled individual enrollment for the device. Doing so will allow the device to re-enroll through the enrollment group.

2. Use the list of provisioned devices for that enrollment group to find the IoT hub that the device was provisioned to and disable or delete it from that hub's identity registry.

Use Azure IoT DPS IP connection filters

8/22/2022 • 5 minutes to read • [Edit Online](#)

Security is an important aspect of any IoT solution. Sometimes you need to explicitly specify the IP addresses from which devices can connect as part of your security configuration. The *IP filter* feature for an Azure IoT Hub Device Provisioning Service (DPS) enables you to configure rules for rejecting or accepting traffic from specific IPv4 addresses.

When to use

There are two specific use-cases where it is useful to block connections to a DPS endpoint from certain IP addresses:

- Your DPS should receive traffic only from a specified range of IP addresses and reject everything else. For example, you are using your DPS with [Azure Express Route](#) to create private connections between a DPS instance and your devices.
- You need to reject traffic from IP addresses that have been identified as suspicious by the DPS administrator.

IP filter rules limitations

Note the following limitations if IP filtering is enabled:

- You might not be able to use the Azure portal to manage enrollments. If this occurs, you can add the IP address of one or more machines to the `ipFilterRules` and manage enrollments in the DPS instance from those machines with Azure CLI, PowerShell, or service APIs.

This scenario is most likely to happen when you want to use IP filtering to allow access only to selected IP addresses. In this case, you configure rules to enable certain addresses or address ranges and a default rule that blocks all other addresses (0.0.0.0/0). This default rule will block Azure portal from performing operations like managing enrollments on the DPS instance. For more information, see [IP filter rule evaluation](#) later in this article.

How filter rules are applied

The IP filter rules are applied at the DPS instance level. Therefore the IP filter rules apply to all connections from devices and back-end apps using any supported protocol.

Any connection attempt from an IP address that matches a rejecting IP rule in your DPS instance receives an unauthorized 401 status code and description. The response message does not mention the IP rule.

IMPORTANT

Rejecting IP addresses can prevent other Azure Services from interacting with the DPS instance.

Default setting

By default, IP filtering is disabled and **Public network access** is set to *All networks*. This default setting means that your DPS accepts connections from any IP address, or conforms to a rule that accepts the 0.0.0.0/0 IP address range.

The screenshot shows the Azure Device Provisioning Service (DPS) Networking page for the 'Contoso-DPS' resource. The left sidebar lists various service management options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, and Settings. Under Settings, 'Networking' is selected. The main content area is titled 'Public access' and includes buttons for Save, Revert, Refresh, and Test. It shows three radio button options for 'Public network access': 'Disabled' (unselected), 'Selected IP ranges' (unselected), and 'All networks' (selected). The 'All networks' option is highlighted with a blue circle.

Add an IP filter rule

To add an IP filter rule:

1. Go to the [Azure portal](#).
2. On the left-hand menu or on the portal page, select **All resources**.
3. Select your Device Provisioning Service.
4. In the **Settings** menu on the left-side, select *Networking*.
5. Under **Public network access**, select *Selected IP ranges*
6. Select + Add IP Filter Rule.

Contoso-DPS | Networking

Device Provisioning Service

Search (Ctrl+ /)

Public access Private access

Overview Activity log Access control (IAM) Tags Diagnose and solve problems

Settings

Quick Start Shared access policies Linked IoT hubs Certificates Manage enrollments Manage allocation policy Networking

Public network access Disabled Selected IP ranges All networks

IP Filter

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
Please create an ip filter rule to enable 'Save'.		
+ Add IP Filter Rule		

7. Fill in the following fields:

FIELD	DESCRIPTION
Name	A unique, case-insensitive, alphanumeric string up to 128 characters long. Only the ASCII 7-bit alphanumeric characters plus <code>{ '-' , ',' , '/' , '\\' , ',' , '+' , '%' , '_' , '#' , '*' , '?' , '!' , '(' , ')' , ',' , '=' , '@' , ';' , ''' }</code> are accepted.
Address Range	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.
Action	Select either Allow or Block .

Contoso-DPS | Networking

Device Provisioning Service

Search (Ctrl+ /)

Public access Private access

Save Revert Refresh Test

Overview Activity log Access control (IAM) Tags Diagnose and solve problems

Settings

Quick Start Shared access policies Linked IoT hubs Certificates Manage enrollments Manage allocation policy Networking Properties Locks

Public network access Disabled Selected IP ranges All networks

IP Filter

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="radio"/> Block
+ Add IP Filter Rule		

8. Select **Save**. You should see an alert notifying you that the update is in progress.

The screenshot shows the 'Networking' blade for the 'Contoso-DPS' device provisioning service. On the left, a navigation menu includes 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', 'Settings' (with 'Quick Start', 'Shared access policies', 'Linked IoT hubs', 'Certificates', 'Manage enrollments', 'Manage allocation policy'), 'Networking' (selected), 'Properties', and 'Locks'. The main area has tabs for 'Public access' (selected) and 'Private access'. Below the tabs are buttons for 'Save', 'Revert', 'Refresh', and 'Test'. A red box highlights a status message: 'Updating Device Provisioning Service'. Under 'Public network access', radio buttons are shown for 'Disabled', 'Selected IP ranges' (which is selected), and 'All networks'. The 'IP Filter' section contains a table:

NAME	ADDRESS RANGE	ACTION
MaliciousIP	6.6.6.6/6	Block

A button '+ Add IP Filter Rule' is at the bottom of the list. A note in a box states: '+ Add IP Filter Rule is disabled when you reach the maximum of 100 IP filter rules.'

NOTE

+ Add IP Filter Rule is disabled when you reach the maximum of 100 IP filter rules.

Edit an IP filter rule

To edit an existing rule:

1. Select the IP filter rule data you want to change.

The screenshot shows the 'Networking' blade for the 'Contoso-DPS' device provisioning service. The left navigation menu is identical to the previous screenshot. The main area shows the same IP filter configuration, but the row for 'MaliciousIP' is now highlighted with a red box. This indicates it is selected for editing. The 'IP Filter' table is:

NAME	ADDRESS RANGE	ACTION
MaliciousIP	6.6.6.6/6	Block

A red box also surrounds the entire table row for 'MaliciousIP'. A button '+ Add IP Filter Rule' is at the bottom of the list.

2. Make the change.

3. Select **Save**.

Delete an IP filter rule

To delete an IP filter rule:

1. Select the delete icon on the row of the IP rule you wish to delete.

The screenshot shows the DPS Networking blade. Under 'Public access', 'Selected IP ranges' is selected for 'Public network access'. In the 'IP Filter' section, there is one rule named 'MaliciousIP' with the address range '6.6.6.6/6' and action 'Block'. The delete icon for this rule is highlighted with a red box.

2. Select **Save**.

IP filter rule evaluation

IP filter rules are applied in order. 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.

To change the order of your IP filter rules:

1. Select the rule you want to move.
2. Drag and drop the rule to the desired location.
3. Select **Save**.

Update IP filter rules using Azure Resource Manager templates

There are two ways you can update your DPS IP filter:

1. Call the IoT Hub Resource REST API method. To learn how to update your IP filter rules using REST, see `IpFilterRule` in the [Definitions section](#) of the [IoT Hub Resource - Update method](#).
2. Use the Azure Resource Manager templates. For guidance on how to use the Resource Manager templates, see [Azure Resource Manager templates](#). The examples that follow show you how to create, edit, and delete DPS IP filter rules with Azure Resource Manager templates.

NOTE

Azure CLI and Azure PowerShell don't currently support DPS IP filter rules updates.

Add an IP filter rule

The following template example creates a new IP filter rule named "AllowAll" that accepts all traffic.

```
{  
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",  
    "contentVersion": "1.0.0.0",  
    "parameters": {  
        "iotDpsName": {  
            "type": "string",  
            "defaultValue": "[resourceGroup().name]",  
            "minLength": 3,  
            "metadata": {  
                "description": "Specifies the name of the IoT DPS service."  
            }  
        },  
        "location": {  
            "type": "string",  
            "defaultValue": "[resourceGroup().location]",  
            "metadata": {  
                "description": "Location for IoT DPS resource."  
            }  
        }  
    },  
    "variables": {  
        "iotDpsApiVersion": "2020-01-01"  
    },  
    "resources": [  
        {  
            "type": "Microsoft.Devices/provisioningServices",  
            "apiVersion": "[variables('iotDpsApiVersion')]",  
            "name": "[parameters('iotDpsName')]",  
            "location": "[parameters('location')]",  
            "sku": {  
                "name": "S1",  
                "tier": "Standard",  
                "capacity": 1  
            },  
            "properties": {  
                "IpFilterRules": [  
                    {  
                        "FilterName": "AllowAll",  
                        "Action": "Accept",  
                        "ipMask": "0.0.0.0/0"  
                    }  
                ]  
            }  
        }  
    ]  
}
```

Update the IP filter rule attributes of the template based on your requirements.

ATTRIBUTE	DESCRIPTION
-----------	-------------

ATTRIBUTE	DESCRIPTION
FilterName	<p>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</p> <div style="border: 1px solid black; padding: 2px; display: inline-block;"> {'-', ':', '/', '\', '.', '+', '%', '_', '#', '*', '?', '!', '(', ')', ',', '=', '@', ';', '\''} </div> <p>are accepted.</p>
Action	Accepted values are Accept or Reject as the action for the IP filter rule.
ipMask	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.

Update an IP filter rule

The following template example updates the IP filter rule named "AllowAll", shown previously, to reject all traffic.

```
{
"$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
"contentVersion": "1.0.0.0",
"parameters": {
"iotDpsName": {
"type": "string",
"defaultValue": "[resourceGroup().name]",
"minLength": 3,
"metadata": {
"description": "Specifies the name of the IoT DPS service."
}
},
"location": {
"type": "string",
"defaultValue": "[resourceGroup().location]",
"metadata": {
"description": "Location for IoT DPS resource."
}
}
},
"variables": {
"iotDpsApiVersion": "2020-01-01"
},
"resources": [
{
"type": "Microsoft.Devices/provisioningServices",
"apiVersion": "[variables('iotDpsApiVersion')]",
"name": "[parameters('iotDpsName')]",
"location": "[parameters('location')]",
"sku": {
"name": "S1",
"tier": "Standard",
"capacity": 1
},
"properties": {
"ipFilterRules": [
{
"filterName": "AllowAll",
"action": "Reject",
"ipMask": "0.0.0.0/0"
}
]
}
]
}
}
```

Delete an IP filter rule

The following template example deletes all IP filter rules for the DPS instance.

```
{  
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",  
  "contentVersion": "1.0.0.0",  
  "parameters": {  
    "iotDpsName": {  
      "type": "string",  
      "defaultValue": "[resourceGroup().name]",  
      "minLength": 3,  
      "metadata": {  
        "description": "Specifies the name of the IoT DPS service."  
      }  
    },  
    "location": {  
      "type": "string",  
      "defaultValue": "[resourceGroup().location]",  
      "metadata": {  
        "description": "Location for IoT DPS resource."  
      }  
    }  
  },  
  "variables": {  
    "iotDpsApiVersion": "2020-01-01"  
  },  
  "resources": [  
    {  
      "type": "Microsoft.Devices/provisioningServices",  
      "apiVersion": "[variables('iotDpsApiVersion')]",  
      "name": "[parameters('iotDpsName')]",  
      "location": "[parameters('location')]",  
      "sku": {  
        "name": "S1",  
        "tier": "Standard",  
        "capacity": 1  
      },  
      "properties": {}  
    }  
  ]  
}
```

Next steps

To further explore the managing DPS, see:

- [Understanding IoT DPS IP addresses](#)
- [Set up DPS using the Azure CLI](#)
- [Control access to DPS](#)

Manage public network access for your IoT Device Provisioning Service

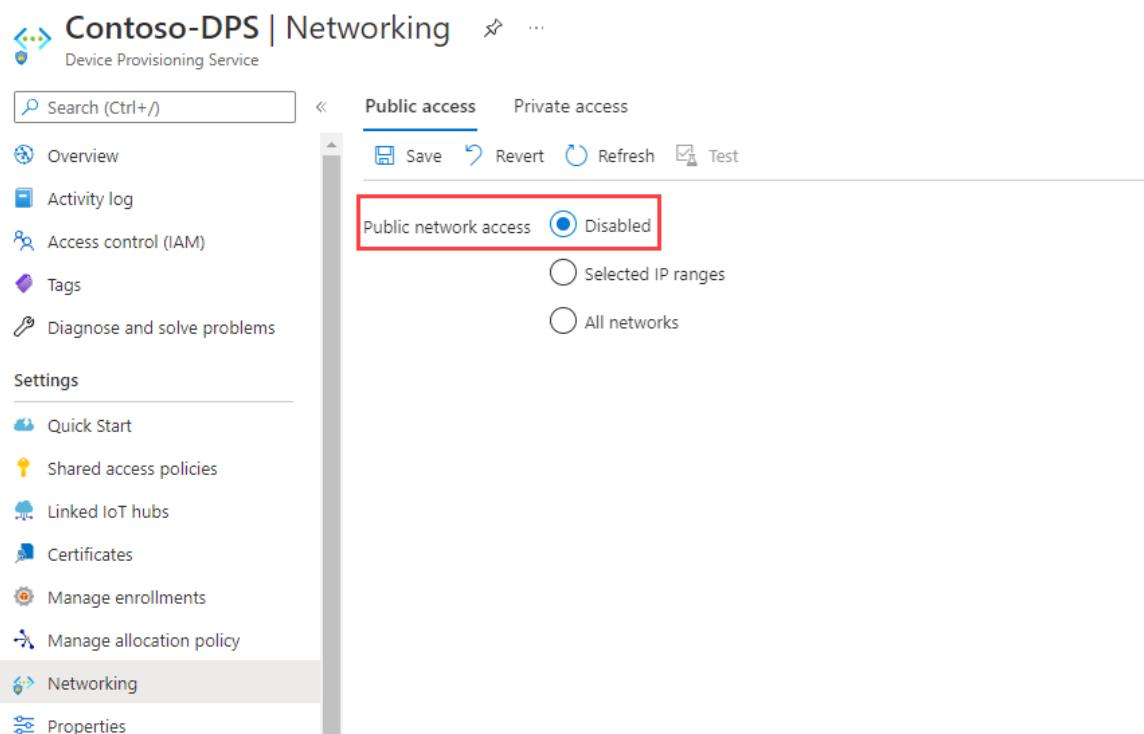
8/22/2022 • 2 minutes to read • [Edit Online](#)

To restrict access to a private endpoint for DPS in your VNet, disable public network access. To do so, use the Azure portal or the `publicNetworkAccess` API. You can also allow public access by using the portal or the `publicNetworkAccess` API.

Turn off public network access using the Azure portal

To turn off public network access:

1. Sign in to the [Azure portal](#).
2. On the left-hand menu or on the portal page, select **All resources**.
3. Select your Device Provisioning Service.
4. In the **Settings** menu on the left-side, select *Networking*.
5. Under **Public network access**, select *Disabled*
6. Select **Save**.



To turn on public network access:

1. Select **All networks**.
2. Select **Save**.

Disable public network access limitations

Note the following limitations when public network access is disabled:

- The DPS instance is accessible only through its VNET private endpoint using Azure private link.
- You can no longer use the Azure portal to manage enrollments for the DPS instance. Instead you can manage enrollments using the Azure CLI, PowerShell, or service APIs from machines inside the virtual network(s) configured on the DPS instance. To learn more, see [Private endpoint limitations](#).

DPS endpoint, IP address, and ports after disabling public network access

DPS is a multi-tenant Platform-as-a-Service (PaaS), where different customers share the same pool of compute, networking, and storage hardware resources. DPS's hostnames map to a public endpoint with a publicly routable IP address over the internet. Different customers share this DPS public endpoint, and IoT devices in wide-area networks and on-premises networks can all access it.

Disabling public network access is enforced on a specific DPS resource, ensuring isolation. To keep the service active for other customer resources using the public path, its public endpoint remains resolvable, IP addresses discoverable, and ports remain open. This is not a cause for concern as Microsoft integrates multiple layers of security to ensure complete isolation between tenants. To learn more, see [Isolation in the Azure Public Cloud](#).

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.

Turn on all network ranges

To turn on all network ranges:

1. Go to the [Azure portal](#).
2. On the left-hand menu or on the portal page, select **All resources**.
3. Select your Device Provisioning Service.
4. In the **Settings** menu on the left-side, select *Networking*.
5. Under **Public network access**, select *All networks*
6. Select **Save**.

Monitoring Azure IoT Hub Device Provisioning Service

8/22/2022 • 4 minutes to read • [Edit Online](#)

When you have critical applications and business processes relying on Azure resources, you want to monitor those resources for their availability, performance, and operation.

This article describes the monitoring data generated by Azure IoT Hub Device Provisioning Service (DPS). DPS uses [Azure Monitor](#). If you are unfamiliar with the features of Azure Monitor common to all Azure services that use it, read [Monitoring Azure resources with Azure Monitor](#).

Monitoring data

DPS collects the same kinds of monitoring data as other Azure resources that are described in [Monitoring data from Azure resources](#).

See [Monitoring Azure IoT Hub Device Provisioning Service data reference](#) for detailed information on the metrics and logs created by DPS.

Collection and routing

Platform metrics and the Activity log are collected and stored automatically, but can be routed to other locations by using a diagnostic setting.

Resource Logs are not collected and stored until you create a diagnostic setting and route them to one or more locations.

In Azure portal, you can select **Diagnostic settings** under **Monitoring** on the left-pane of your DPS instance followed by **Add diagnostic setting** to create diagnostic settings scoped to the logs and platform metrics emitted by your instance.

The following screenshot shows a diagnostic setting for routing to a Log Analytics workspace.

Diagnostic setting

X

Save Discard Delete Feedback

A diagnostic setting specifies a list of categories of platform logs and/or metrics that you want to collect from a resource, and one or more destinations that you would stream them to. Normal usage charges for the destination will occur. [Learn more about the different log categories and contents of those logs](#)

Diagnostic setting name *

Logs

Category groups ⓘ

- allLogs

Categories

- DeviceOperations
- ServiceOperations

Metrics

- AllMetrics

Destination details

Send to Log Analytics workspace

Subscription

Log Analytics workspace

Archive to a storage account

Stream to an event hub

Send to partner solution

See [Create diagnostic setting to collect platform logs and metrics in Azure](#) for the detailed process for creating a diagnostic setting using the Azure portal, CLI, or PowerShell. When you create a diagnostic setting, you specify which categories of logs to collect. The categories for DPS are listed in [Resource logs in the Azure IoT Hub Device Provisioning Service monitoring data reference](#).

The metrics and logs you can collect are discussed in the following sections.

Analyzing metrics

You can analyze metrics for DPS with metrics from other Azure services using metrics explorer by opening **Metrics** from the **Azure Monitor** menu. See [Getting started with Azure Metrics Explorer](#) for details on using this tool.

In Azure portal, you can select **Metrics** under **Monitoring** on the left-pane of your DPS instance to open metrics explorer scoped, by default, to the platform metrics emitted by your instance:

Home > MyExampleDps

MyExampleDps | Metrics ...

Azure IoT Hub Device Provisioning Service (DPS)

Search (Ctrl+ /) New chart Refresh Share Feedback Local Time: Last 24 hours (Automatic)

Settings

- Quick Start
- Shared access policies
- Linked IoT hubs
- Certificates
- Manage enrollments
- Manage allocation policy
- Networking
- Properties
- Locks

Monitoring

- Alerts
- Metrics**
- Diagnostic settings
- Logs

Automation

Chart Title

Add metric Add filter Apply splitting Line chart Drill into Logs New alert rule Save to dashboard ...

Scope	Metric Namespace	Metric	Aggregation
MyExampleDps	Device Provisioning Ser...	Select metric	Select aggregation

100
90
80
70
60
50
40
30
20
10
0

6 PM Thu 31 6 AM 12 PM UTC-07:00

Select a metric above to see data appear on this chart or learn more below:

- Filter + Split** Apply filters and splits to identify outlying segments
- Plot multiple metrics** Create charts with multiple metrics and resources
- Build custom dashboards** Pin charts to your dashboards

For a list of the platform metrics collected for DPS, see [Metrics in the Monitoring Azure IoT Hub Device Provisioning Service data reference](#).

For reference, you can see a list of [all resource metrics supported in Azure Monitor](#).

Analyzing logs

Data in Azure Monitor Logs is stored in tables where each table has its own set of unique properties.

To route data to Azure Monitor Logs, you must create a diagnostic setting to send resource logs or platform metrics to a Log Analytics workspace. To learn more, see [Collection and routing](#).

In Azure portal, you can select **Logs** under **Monitoring** on the left-pane of your DPS instance to perform Log Analytics queries scoped, by default, to the logs and metrics collected in Azure Monitor Logs for your instance.

TimeGenerated [UTC]	ResourceId	Category	
4/1/2022, 12:52:32.000 AM	/SUBSCRIPTIONS/747F1067-0B7C-4FE4-B054-9DEAA894152F/R...	ServiceOperati...	N
4/1/2022, 12:53:00.000 AM	/SUBSCRIPTIONS/747F1067-0B7C-4FE4-B054-9DEAA894152F/R...	ServiceOperati...	N
4/1/2022, 12:53:00.000 AM	/SUBSCRIPTIONS/747F1067-0B7C-4FE4-B054-9DEAA894152F/R...	ServiceOperati...	N
4/1/2022, 12:52:00.000 AM	/SUBSCRIPTIONS/747F1067-0B7C-4FE4-B054-9DEAA894152F/R...	ServiceOperati...	N
4/1/2022, 12:52:01.000 AM	/SUBSCRIPTIONS/747F1067-0B7C-4FE4-B054-9DEAA894152F/R...	ServiceOperati...	N
4/1/2022, 12:52:01.000 AM	/SUBSCRIPTIONS/747F1067-0B7C-4FE4-B054-9DEAA894152F/R...	ServiceOperati...	N
4/1/2022, 12:52:31.000 AM	/SUBSCRIPTIONS/747F1067-0B7C-4FE4-B054-9DEAA894152F/R...	ServiceOperati...	N
4/1/2022, 12:52:32.000 AM	/SUBSCRIPTIONS/747F1067-0B7C-4FE4-B054-9DEAA894152F/R...	ServiceOperati...	N
4/1/2022, 12:52:59.000 AM	/SUBSCRIPTIONS/747F1067-0B7C-4FE4-B054-9DEAA894152F/R...	ServiceOperati...	N
4/1/2022, 12:50:15.000 AM	/SUBSCRIPTIONS/747F1067-0B7C-4FE4-B054-9DEAA894152F/R...	ServiceOperati...	N

IMPORTANT

When you select **Logs** from the DPS menu, Log Analytics is opened with the query scope set to the current DPS instance. This means that log queries will only include data from that resource. If you want to run a query that includes data from other DPS instances or data from other Azure services, select **Logs** from the **Azure Monitor** menu. See [Log query scope and time range in Azure Monitor Log Analytics](#) for details.

Run queries against the **AzureDiagnostics** table to see the resource logs collected for the diagnostic settings you've created for your DPS instance.

```
AzureDiagnostics
```

All resource logs in Azure Monitor have the same fields followed by service-specific fields. The common schema is outlined in [Azure Monitor resource log schema](#). The schema for DPS resource logs is found in [Resource logs in the Monitoring Azure IoT Hub Device Provisioning Service data reference](#).

The [Activity log](#) is a type of platform log in Azure that provides insight into subscription-level events. You can view it independently or route it to Azure Monitor Logs, where you can do much more complex queries using

Log Analytics.

For a list of the types of resource logs collected for DPS, see [Resource logs in the Monitoring Azure IoT Hub Device Provisioning Service data reference](#).

For a list of the tables used by Azure Monitor Logs and queryable by Log Analytics, see [Azure Monitor Logs tables in the Monitoring Azure IoT Hub Device Provisioning Service data reference](#).

Alerts

Azure Monitor alerts proactively notify you when important conditions are found in your monitoring data. They allow you to identify and address issues in your system before your customers notice them. You can set alerts on [metrics](#), [logs](#), and the [activity log](#). Different types of alerts have benefits and drawbacks.

Next steps

- See [Monitoring Azure IoT Hub Device Provisioning Service data reference](#) for a reference of the metrics, logs, and other important values created by DPS.
- See [Monitoring Azure resources with Azure Monitor](#) for details on monitoring Azure resources.

Monitoring Azure IoT Hub Device Provisioning Service data reference

8/22/2022 • 3 minutes to read • [Edit Online](#)

See [Monitoring IoT Hub Device Provisioning Service](#) for details on collecting and analyzing monitoring data for Azure IoT Hub Device Provisioning Service (DPS).

Metrics

This section lists all the automatically collected platform metrics collected for DPS.

Resource Provider and Type: [Microsoft.Devices/provisioningServices](#).

METRIC	EXPORTABLE VIA DIAGNOSTIC SETTINGS?	METRIC DISPLAY NAME	UNIT	AGGREGATION TYPE	DESCRIPTION	DIMENSIONS
AttestationAttempts	Yes	Attestation attempts	Count	Total	Number of device attestations attempted	ProvisioningServiceName, Status, Protocol
DeviceAssignments	Yes	Devices assigned	Count	Total	Number of devices assigned to an IoT hub	ProvisioningServiceName, IoTHubName
RegistrationAttempts	Yes	Registration attempts	Count	Total	Number of device registrations attempted	ProvisioningServiceName, IoTHubName, Status

For more information, see a list of [all platform metrics supported in Azure Monitor](#).

Metric dimensions

DPS has the following dimensions associated with its metrics.

DIMENSION NAME	DESCRIPTION
IoTHubName	The name of the target IoT hub.
Protocol	The device or service protocol used.
ProvisioningServiceName	The name of the DPS instance.
Status	The status of the operation.

For more information on what metric dimensions are, see [Multi-dimensional metrics](#).

Resource logs

This section lists the types of resource logs you can collect for DPS.

Resource Provider and Type: [Microsoft.Devices/provisioningServices](#).

CATEGORY	DESCRIPTION
DeviceOperations	Logs related to device attestation events. See device APIs listed in Billable service operations and pricing .
ServiceOperations	Logs related to DPS service events. See DPS service APIs listed in Billable service operations and pricing .

For reference, see a list of [all resource logs category types supported in Azure Monitor](#).

DPS uses the [AzureDiagnostics](#) table to store resource log information. The following columns are relevant.

PROPERTY	DATA TYPE	DESCRIPTION
ApplicationId	GUID	Application ID used in bearer authorization.
CallerIpAddress	String	A masked source IP address for the event.
Category	String	Type of operation, either ServiceOperations or DeviceOperations .
CorrelationId	GUID	Unique identifier for the event.
DurationMs	String	How long it took to perform the event in milliseconds.
Level	Int	The logging severity of the event. For example, Information or Error.
OperationName	String	The type of action performed during the event. For example: Query, Get, Upsert, and so on.
OperationVersion	String	The API Version used during the event.
Resource	String	The name forOF the resource where the event took place. For example, "MYEXAMPLEDPS".
ResourceGroup	String	The name of the resource group where the resource is located.
ResourceId	String	The Azure Resource Manager Resource ID for the resource where the event took place.
ResourceProvider	String	The resource provider for the the event. For example, "MICROSOFT.DEVICES".

PROPERTY	DATA TYPE	DESCRIPTION
ResourceType	String	The resource type for the event. For example, "PROVISIONINGSERVICES".
ResultDescription	String	Error details for the event if unsuccessful.
ResultSignature	String	HTTP status code for the event if unsuccessful.
ResultType	String	Outcome of the event: Success, Failure, ClientError, and so on.
SubscriptionId	GUID	The subscription ID of the Azure subscription where the resource is located.
TenantId	GUID	The tenant ID for the Azure tenant where the resource is located.
TimeGenerated	DateTime	The date and time that this event occurred, in UTC.
location_s	String	The Azure region where the event took place.
properties_s	JSON	Additional information details for the event.

DeviceOperations

The following JSON is an example of a successful attestation attempt from a device. The registration ID for the device is identified in the `properties_s` property.

```
{
  "CallerIPAddress": "24.18.226.XXX",
  "Category": "DeviceOperations",
  "CorrelationId": "68952383-80c0-436f-a2e3-f8ae9a41c69d",
  "DurationMs": "226",
  "Level": "Information",
  "OperationName": "AttestationAttempt",
  "OperationVersion": "March2019",
  "Resource": "MYEXAMPLEDPS",
  "ResourceGroup": "MYRESOURCEGROUP",
  "ResourceId": "/SUBSCRIPTIONS/747F1067-xxx-xxx-xxxx-
9DEAA894152F/RESOURCEGROUPS/MYRESOURCEGROUP/PROVIDERS/MICROSOFT.DEVICES/PROVISIONINGSERVICES/MYEXAMPLEDPS",
  "ResourceProvider": "MICROSOFT.DEVICES",
  "ResourceType": "PROVISIONINGSERVICES",
  "ResultDescription": "",
  "ResultSignature": "",
  "ResultType": "Success",
  "SourceSystem": "Azure",
  "SubscriptionId": "747F1067-xxx-xxx-xxxx-9DEAA894152F",
  "TenantId": "37dc621-xxxx-xxxx-xxxx-e8c8addbc4e5",
  "TimeGenerated": "2022-04-02T00:05:51Z",
  "Type": "AzureDiagnostics",
  "_ResourceId": "/subscriptions/747F1067-xxx-xxx-xxxx-
9DEAA894152F/resourcegroups/myresourcegroup/providers/microsoft.devices/provisioningservices/myexampledps",
  "location_s": "centralus",
  "properties_s": "{\"id\":\"my-device-1\",\"type\":\"Registration\",\"protocol\":\"Mqtt\"}",
}
}
```

ServiceOperations

The following JSON is an example of a successful add (`upsert`) individual enrollment operation. The registration ID for the enrollment and the type of enrollment are identified in the `properties_s` property.

```
{
  "CallerIPAddress": "13.91.244.XXX",
  "Category": "ServiceOperations",
  "CorrelationId": "23bd419d-d294-452b-9b1b-520afef5ef52",
  "DurationMs": "98",
  "Level": "Information",
  "OperationName": "Upsert",
  "OperationVersion": "October2021",
  "Resource": "MYEXAMPLEDPS",
  "ResourceGroup": "MYRESOURCEGROUP",
  "ResourceId": "/SUBSCRIPTIONS/747F1067-xxxx-xxxx-xxxx-
9DEAA894152F/RESOURCEGROUPS/MYRESOURCEGROUP/PROVIDERS/MICROSOFT.DEVICES/PROVISIONINGSERVICES/MYEXAMPLEDPS",
  "ResourceProvider": "MICROSOFT.DEVICES",
  "ResourceType": "PROVISIONINGSERVICES",
  "ResultDescription": "",
  "ResultSignature": "",
  "ResultType": "Success",
  "SourceSystem": "Azure",
  "SubscriptionId": "747f1067-xxxx-xxxx-xxxx-9deaa894152f",
  "TenantId": "37dc621-xxxx-xxxx-xxxx-e8c8addbc4e5",
  "TimeGenerated": "2022-04-01T00:52:00Z",
  "Type": "AzureDiagnostics",
  "_ResourceId": "/subscriptions/747F1067-xxxx-xxxx-xxxx-
9DEAA894152F/resourcegroups/myresourcegroup/providers/microsoft.devices/provisioningservices/myexampledps",
  "location_s": "centralus",
  "properties_s": "{\"id\":\"my-device-1\",\"type\":\"IndividualEnrollment\",\"protocol\":\"Http\"}",
}
}
```

Azure Monitor Logs tables

This section refers to all of the Azure Monitor Logs Kusto tables relevant to DPS and available for query by Log Analytics. For a list of these tables and links to more information for the DPS resource type, see [Device Provisioning Services](#) in the Azure Monitor Logs table reference.

For a reference of all Azure Monitor Logs / Log Analytics tables, see the [Azure Monitor Log Table Reference](#).

Activity log

For more information on the schema of Activity Log entries, see [Activity Log schema](#).

See Also

- See [Monitoring Azure IoT Hub Device Provisioning Service](#) for a description of monitoring Azure IoT Hub Device Provisioning Service.
- See [Monitoring Azure resources with Azure Monitor](#) for details on monitoring Azure resources.

Create and provision IoT Edge devices at scale with a TPM on Linux

8/22/2022 • 20 minutes to read • [Edit Online](#)

Applies to: IoT Edge 1.1 Other versions: IoT Edge 1.2, [IoT Edge 1.3](#)

Applies to: IoT Edge 1.2 IoT Edge 1.3 Other versions: [IoT Edge 1.1](#)

This article provides instructions for autoprovioning an Azure IoT Edge for Linux device by using a Trusted Platform Module (TPM). You can automatically provision IoT Edge devices with the [Azure IoT Hub device provisioning service](#). If you're unfamiliar with the process of autoprovioning, review the [provisioning overview](#) before you continue.

This article outlines two methodologies. Select your preference based on the architecture of your solution:

- Autoprovion a Linux device with physical TPM hardware. An example is the [Infineon OPTIGA™ TPM SLB 9670](#).
- Autoprovion a Linux virtual machine (VM) with a simulated TPM running on a Windows development machine with Hyper-V enabled. We recommend using this methodology only as a testing scenario. A simulated TPM doesn't offer the same security as a physical TPM.

Instructions differ based on your methodology, so make sure you're on the correct tab going forward.

- [Physical device](#)
- [Virtual machine](#)

The tasks are as follows:

1. Retrieve provisioning information for your TPM.
2. Create an individual enrollment for your device in an instance of the IoT Hub device provisioning service.
3. Install the IoT Edge runtime and connect the device to the IoT hub.

Prerequisites

Cloud resources

- An active IoT hub
- An instance of the IoT Hub device provisioning service in Azure, linked to your IoT hub
 - If you don't have a device provisioning service instance, you can follow the instructions in the [Create a new IoT Hub device provisioning service](#) and [Link the IoT hub and your device provisioning service](#) sections of the IoT Hub device provisioning service quickstart.
 - After you have the device provisioning service running, copy the value of ID Scope from the overview page. You use this value when you configure the IoT Edge runtime.

Device requirements

- [Physical device](#)
- [Virtual machine](#)

A physical Linux device to be the IoT Edge device.

NOTE

TPM 2.0 is required when you use TPM attestation with the device provisioning service.

You can only create individual, not group, device provisioning service enrollments when you use a TPM.

Set up your device

- [Physical device](#)
- [Virtual machine](#)

If you're using a physical Linux device with a TPM, there are no extra steps to set up your device.

You're ready to continue.

Retrieve provisioning information for your TPM

In this section, you build a tool that you can use to retrieve the registration ID and endorsement key for your TPM.

1. Sign in to your device, and then follow the steps in [Set up a Linux development environment](#) to install and build the Azure IoT device SDK for C.
2. Run the following commands to build the SDK tool that retrieves your device provisioning information for your TPM.

```
cd azure-iot-sdk-c/cmake  
cmake -Duse_prov_client:BOOL=ON ..  
cd provisioning_client/tools/tpm_device_provision  
make  
sudo ./tpm_device_provision
```

3. The output window displays the device's **Registration ID** and the **Endorsement key**. Copy these values for use later when you create an individual enrollment for your device in the device provisioning service.

TIP

If you don't want to use the SDK tool to retrieve the information, you need to find another way to obtain the provisioning information. The endorsement key, which is unique to each TPM chip, is obtained from the TPM chip manufacturer associated with it. You can derive a unique registration ID for your TPM device. For example, you can create an SHA-256 hash of the endorsement key.

After you have your registration ID and endorsement key, you're ready to continue.

Create a device provisioning service enrollment

Use your TPM's provisioning information to create an individual enrollment in the device provisioning service.

When you create an enrollment in the device provisioning service, you have the opportunity to declare an **Initial Device Twin State**. In the device twin, you can set tags to group devices by any metric used in your solution, like region, environment, location, or device type. These tags are used to create [automatic deployments](#).

TIP

The steps in this article are for the Azure portal, but you can also create individual enrollments by using the Azure CLI. For more information, see [az iot dps enrollment](#). As part of the CLI command, use the **edge-enabled** flag to specify that the enrollment is for an IoT Edge device.

1. In the [Azure portal](#), go to your instance of the IoT Hub device provisioning service.
2. Under **Settings**, select **Manage enrollments**.
3. Select **Add individual enrollment**, and then complete the following steps to configure the enrollment:
 - a. For **Mechanism**, select **TPM**.
 - b. Provide the **Endorsement key** and **Registration ID** that you copied from your VM or physical device.
 - c. Provide an ID for your device if you want. If you don't provide a device ID, the registration ID is used.
 - d. Select **True** to declare that your VM or physical device is an IoT Edge device.
 - e. Choose the linked IoT hub that you want to connect your device to, or select **Link to new IoT Hub**. You can choose multiple hubs, and the device will be assigned to one of them according to the selected assignment policy.
 - f. Add a tag value to the **Initial Device Twin State** if you want. You can use tags to target groups of devices for module deployment. For more information, see [Deploy IoT Edge modules at scale](#).
 - g. Select **Save**.

Now that an enrollment exists for this device, the IoT Edge runtime can automatically provision the device during installation.

Install IoT Edge

In this section, you prepare your Linux VM or physical device for IoT Edge. Then, you install IoT Edge.

First, run the following commands to add the package repository and then add the Microsoft package signing key to your list of trusted keys.

- [Ubuntu](#)
- [Debian](#)
- [Raspberry Pi OS](#)
- [Red Hat Enterprise Linux](#)

Installing can be done with a few commands. Open a terminal and run the following commands:

- 20.04:

```
wget https://packages.microsoft.com/config/ubuntu/20.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
rm packages-microsoft-prod.deb
```

- 18.04:

```
 wget https://packages.microsoft.com/config/ubuntu/18.04/multiarch/packages-microsoft-prod.deb -O packages-microsoft-prod.deb  
 sudo dpkg -i packages-microsoft-prod.deb  
 rm packages-microsoft-prod.deb
```

NOTE

Azure IoT Edge software packages are subject to the license terms located in each package (`usr/share/doc/{package-name}` or the `LICENSE` directory). Read the license terms prior to using a package. Your installation and use of a package constitutes your acceptance of these terms. If you don't agree with the license terms, don't use that package.

Install a container engine

Azure IoT Edge relies on an OCI-compatible container runtime. For production scenarios, we recommend that you use the Moby engine. The Moby engine is the only container engine officially supported with IoT Edge. Docker CE/EE container images are compatible with the Moby runtime.

- [Ubuntu](#)
- [Debian](#)
- [Raspberry Pi OS](#)
- [Red Hat Enterprise Linux](#)

Install the Moby engine.

```
sudo apt-get update; \  
 sudo apt-get install moby-engine
```

Once the Moby engine is successfully installed, configure it to use [local](#) logging driver as the logging mechanism. To learn more about logging configuration, see [Production Deployment Checklist](#).

- Create or open the Docker daemon's config file at `/etc/docker/daemon.json`.
- Set the default logging driver to the `local` logging driver as shown in the example below.

```
{  
    "log-driver": "local"  
}
```

- Restart the container engine for the changes to take effect.

TIP

If you get errors when you install the Moby container engine, verify your Linux kernel for Moby compatibility. Some embedded device manufacturers ship device images that contain custom Linux kernels without the features required for container engine compatibility. Run the following command, which uses the `check-config` script provided by Moby, to check your kernel configuration:

```
curl -ssl https://raw.githubusercontent.com/moby/moby/master/contrib/check-config.sh -o check-config.sh  
chmod +x check-config.sh  
./check-config.sh
```

In the output of the script, check that all items under `Generally Necessary` and `Network Drivers` are enabled. If you're missing features, enable them by rebuilding your kernel from source and selecting the associated modules for inclusion in the appropriate kernel .config. Similarly, if you're using a kernel configuration generator like `defconfig` or `menuconfig`, find and enable the respective features and rebuild your kernel accordingly. After you've deployed your newly modified kernel, run the `check-config` script again to verify that all the required features were successfully enabled.

Install the IoT Edge runtime

The IoT Edge security daemon provides and maintains security standards on the IoT Edge device. The daemon starts on every boot and bootstraps the device by starting the rest of the IoT Edge runtime.

The steps in this section represent the typical process to install the latest version on a device that has internet connection. If you need to install a specific version, like a pre-release version, or need to install while offline, follow the **Offline or specific version installation** steps later in this article.

Install IoT Edge version 1.1.* along with the **libiothsm-std** package:

- [Ubuntu](#)
- [Debian](#)
- [Raspberry Pi OS](#)
- [Red Hat Enterprise Linux](#)

```
sudo apt-get update; \  
sudo apt-get install iotedge
```

NOTE

IoT Edge version 1.1 is the long-term support branch of IoT Edge. If you are running an older version, we recommend installing or updating to the latest patch as older versions are no longer supported.

The IoT Edge service provides and maintains security standards on the IoT Edge device. The service starts on every boot and bootstraps the device by starting the rest of the IoT Edge runtime.

Beginning with version 1.2, the IoT identity service handles identity provisioning and management for IoT Edge and for other device components that need to communicate with IoT Hub.

The steps in this section represent the typical process to install the latest version on a device that has internet connection. If you need to install a specific version, like a pre-release version, or need to install while offline, follow the **Offline or specific version installation** steps later in this article.

NOTE

The steps in this section show you how to install the latest IoT Edge version.

If you already have an IoT Edge device running an older version and want to upgrade to the latest release, use the steps in [Update the IoT Edge security daemon and runtime](#). Later versions are sufficiently different from previous versions of IoT Edge that specific steps are necessary to upgrade.

- [Ubuntu](#)
- [Debian](#)
- [Raspberry Pi OS](#)
- [Red Hat Enterprise Linux](#)

Install the latest version of IoT Edge and the IoT identity service package:

```
sudo apt-get update; \
sudo apt-get install aziot-edge defender-iot-micro-agent-edge
```

The `defender-iot-micro-agent-edge` package includes the Microsoft Defender for IoT security micro-agent that provides endpoint visibility into security posture management, vulnerabilities, threat detection, fleet management and more to help you secure your IoT Edge devices. It is recommended to install the micro agent with the Edge agent to enable security monitoring and hardening of your Edge devices. To learn more about Microsoft Defender for IoT, see [What is Microsoft Defender for IoT for device builders](#).

Provision the device with its cloud identity

After the runtime is installed on your device, configure the device with the information it uses to connect to the device provisioning service and IoT Hub.

1. Know your device provisioning service **ID Scope** and device **Registration ID** that were gathered previously.
2. Open the configuration file on your IoT Edge device.

```
sudo nano /etc/iotedge/config.yaml
```

3. Find the provisioning configuration section of the file. Uncomment the lines for TPM provisioning, and make sure any other provisioning lines are commented out.

The `provisioning:` line should have no preceding whitespace, and nested items should be indented by two spaces.

```
# DPS TPM provisioning configuration
provisioning:
  source: "dps"
  global_endpoint: "https://global.azure-devices-provisioning.net"
  scope_id: "SCOPE_ID_HERE"
  attestation:
    method: "tpm"
    registration_id: "REGISTRATION_ID_HERE"

  # always_reprovision_on_startup: true
  # dynamic_reprovisioning: false
```

4. Update the values of `scope_id` and `registration_id` with your device provisioning service and device

information. The `scope_id` value is the **ID Scope** from your device provisioning service instance's overview page.

5. Optionally, use the `always_reprovision_on_startup` or `dynamic_reprovisioning` lines to configure your device's reprovisioning behavior. If a device is set to reprovision on startup, it will always attempt to provision with DPS first and then fall back to the provisioning backup if that fails. If a device is set to dynamically reprovision itself, IoT Edge (and all modules) will restart and reprovision if a reprovisioning event is detected, like if the device is moved from one IoT Hub to another. Specifically, IoT Edge checks for `bad_credential` or `device_disabled` errors from the SDK to detect the reprovision event. To trigger this event manually, disable the device in IoT Hub. For more information, see [IoT Hub device reprovisioning concepts](#).

6. Save and close the file.

1. Know your device provisioning service **ID Scope** and device **Registration ID** that were gathered previously.
2. Create a configuration file for your device based on a template file that's provided as part of the IoT Edge installation.

```
sudo cp /etc/aziot/config.toml.edge.template /etc/aziot/config.toml
```

3. Open the configuration file on the IoT Edge device.

```
sudo nano /etc/aziot/config.toml
```

4. Find the provisioning configurations section of the file. Uncomment the lines for TPM provisioning, and make sure any other provisioning lines are commented out.

```
# DPS provisioning with TPM
[provisioning]
source = "dps"
global_endpoint = "https://global.azure-devices-provisioning.net"
id_scope = "SCOPE_ID_HERE"

[provisioning.attestation]
method = "tpm"
registration_id = "REGISTRATION_ID_HERE"

# auto_reprovisioning_mode = Dynamic
```

5. Update the values of `id_scope` and `registration_id` with your device provisioning service and device information. The `scope_id` value is the **ID Scope** from your device provisioning service instance's overview page.
6. Optionally, find the auto reprovisioning mode section of the file. Use the `auto_reprovisioning_mode` parameter to configure your device's reprovisioning behavior. **Dynamic** - Reprovision when the device detects that it may have been moved from one IoT Hub to another. This is the default. **AlwaysOnStartup** - Reprovision when the device is rebooted or a crash causes the daemon(s) to restart. **OnErrorOnly** - Never trigger device reprovisioning automatically. Each mode has an implicit device reprovisioning fallback if the device is unable to connect to IoT Hub during identity provisioning due to connectivity errors. For more information, see [IoT Hub device reprovisioning concepts](#).
7. Save and close the file.

Give IoT Edge access to the TPM

The IoT Edge runtime needs to access the TPM to automatically provision your device.

You can give TPM access to the IoT Edge runtime by overriding the systemd settings so that the `iotedge` service has root privileges. If you don't want to elevate the service privileges, you can also use the following steps to manually provide TPM access.

1. Create a new rule that will give the IoT Edge runtime access to `tpm0` and `tpmrm0`.

```
sudo touch /etc/udev/rules.d/tpmaccess.rules
```

2. Open the rules file.

```
sudo nano /etc/udev/rules.d/tpmaccess.rules
```

3. Copy the following access information into the rules file. The `tpmrm0` might not be present on devices that use a kernel earlier than 4.12. Devices that don't have `tpmrm0` will safely ignore that rule.

```
# allow iotedge access to tpm0
KERNEL=="tpm0", SUBSYSTEM=="tpm", OWNER="iotedge", MODE="0600"
KERNEL=="tpmrm0", SUBSYSTEM=="tpmrm", OWNER="iotedge", MODE="0600"
```

4. Save and exit the file.

5. Trigger the `udev` system to evaluate the new rule.

```
/bin/udevadm trigger --subsystem-match=tpm --subsystem-match=tpmrm
```

6. Verify that the rule was successfully applied.

```
ls -l /dev/tpm*
```

Successful output appears as follows:

```
crw----- 1 iotedge root 10, 224 Jul 20 16:27 /dev/tpm0
crw----- 1 iotedge root 10, 224 Jul 20 16:27 /dev/tpmrm0
```

If you don't see that the correct permissions have been applied, try rebooting your machine to refresh `udev`.

7. Restart the IoT Edge runtime so that it picks up all the configuration changes that you made on the device.

```
sudo systemctl restart iotedge
```

The IoT Edge runtime relies on a TPM service that brokers access to a device's TPM. This service needs to access the TPM to automatically provision your device.

You can give access to the TPM by overriding the systemd settings so that the `aziottpm` service has root privileges. If you don't want to elevate the service privileges, you can also use the following steps to manually provide TPM access.

1. Create a new rule that will give the IoT Edge runtime access to `tpm0` and `tpmrm0`.

```
sudo touch /etc/udev/rules.d/tpmaccess.rules
```

2. Open the rules file.

```
sudo nano /etc/udev/rules.d/tpmaccess.rules
```

3. Copy the following access information into the rules file. The `tpmrm0` might not be present on devices that use a kernel earlier than 4.12. Devices that don't have `tpmrm0` will safely ignore that rule.

```
# allow aziottpm access to tpm0 and tpmrm0
KERNEL=="tpm0", SUBSYSTEM=="tpm", OWNER="aziottpm", MODE="0660"
KERNEL=="tpmrm0", SUBSYSTEM=="tpmrm", OWNER="aziottpm", MODE="0660"
```

4. Save and exit the file.

5. Trigger the `udev` system to evaluate the new rule.

```
/bin/udevadm trigger --subsystem-match=tpm --subsystem-match=tpmrm
```

6. Verify that the rule was successfully applied.

```
ls -l /dev/tpm*
```

Successful output appears as follows:

```
crw-rw---- 1 root aziottpm 10, 224 Jul 20 16:27 /dev/tpm0
crw-rw---- 1 root aziottpm 10, 224 Jul 20 16:27 /dev/tpmrm0
```

If you don't see that the correct permissions have been applied, try rebooting your machine to refresh `udev`.

7. Apply the configuration changes that you made on the device.

```
sudo iotedge config apply
```

Verify successful installation

If you didn't already, restart the IoT Edge runtime so that it picks up all the configuration changes that you made on the device.

```
sudo systemctl restart iotedge
```

Check to see that the IoT Edge runtime is running.

```
sudo systemctl status iotedge
```

Examine daemon logs.

```
journalctl -u iotedge --no-pager --no-full
```

If you see provisioning errors, it might be that the configuration changes haven't taken effect yet. Try restarting the IoT Edge daemon again.

```
sudo systemctl daemon-reload
```

Or, try restarting your VM to see if the changes take effect on a fresh start.

If you didn't already, apply the configuration changes that you made on the device.

```
sudo iotedge config apply
```

Check to see that the IoT Edge runtime is running.

```
sudo iotedge system status
```

Examine daemon logs.

```
sudo iotedge system logs
```

If you see provisioning errors, it might be that the configuration changes haven't taken effect yet. Try restarting the IoT Edge daemon.

```
sudo systemctl daemon-reload
```

Or, try restarting your VM to see if the changes take effect on a fresh start.

If the runtime started successfully, you can go into your IoT hub and see that your new device was automatically provisioned. Now your device is ready to run IoT Edge modules.

List running modules.

```
iotedge list
```

You can verify that the individual enrollment that you created in the device provisioning service was used. Go to your device provisioning service instance in the Azure portal. Open the enrollment details for the individual enrollment that you created. Notice that the status of the enrollment is **assigned** and the device ID is listed.

Next steps

The device provisioning service enrollment process lets you set the device ID and device twin tags at the same time as you provision the new device. You can use those values to target individual devices or groups of devices by using automatic device management.

Learn how to [deploy and monitor IoT Edge modules at scale by using the Azure portal](#) or [the Azure CLI](#).

Create and provision IoT Edge devices at scale with a TPM on Windows

8/22/2022 • 7 minutes to read • [Edit Online](#)

Applies to: IoT Edge 1.1

This article provides instructions for autoprovisioning an Azure IoT Edge for Windows device by using a Trusted Platform Module (TPM). You can automatically provision IoT Edge devices with the [Azure IoT Hub device provisioning service](#). If you're unfamiliar with the process of autoprovisioning, review the [provisioning overview](#) before you continue.

NOTE

Azure IoT Edge with Windows containers will not be supported starting with version 1.2 of Azure IoT Edge.

Consider using the new method for running IoT Edge on Windows devices, [Azure IoT Edge for Linux on Windows](#).

If you want to use Azure IoT Edge for Linux on Windows, you can follow the steps in the [equivalent how-to guide](#).

This article outlines two methodologies. Select your preference based on the architecture of your solution:

- Autoprovision a Windows device with physical TPM hardware.
- Autoprovision a Windows device running a simulated TPM. We recommend this methodology only as a testing scenario. A simulated TPM doesn't offer the same security as a physical TPM.

Instructions differ based on your methodology, so make sure you're on the correct tab going forward.

The tasks are as follows:

- [Physical TPM](#)
 - [Simulated TPM](#)
-
- Retrieve your device's provisioning information.
 - Create an individual enrollment for the device.
 - Install the IoT Edge runtime and connect the device to IoT Hub.

Prerequisites

The prerequisites are the same for physical TPM and virtual TPM solutions.

Cloud resources

- An active IoT hub
- An instance of the IoT Hub device provisioning service in Azure, linked to your IoT hub
 - If you don't have a device provisioning service instance, you can follow the instructions in the [Create a new IoT Hub device provisioning service](#) and [Link the IoT hub and your device provisioning service](#) sections of the IoT Hub device provisioning service quickstart.
 - After you have the device provisioning service running, copy the value of ID Scope from the overview page. You use this value when you configure the IoT Edge runtime.

Device requirements

A Windows development machine. This article uses Windows 10.

NOTE

TPM 2.0 is required when you use TPM attestation with the device provisioning service.

You can only create individual, not group, device provisioning service enrollments when you use a TPM.

Set up your TPM

- [Physical TPM](#)
- [Simulated TPM](#)

In this section, you build a tool that you can use to retrieve the registration ID and endorsement key for your TPM.

1. Follow the steps in [Set up a Windows development environment](#) to install and build the Azure IoT device SDK for C.
2. Run the following commands in an elevated PowerShell session to build the SDK tool that retrieves your device provisioning information for your TPM.

```
cd azure-iot-sdk-c\cmake  
cmake -Duse_prov_client:BOOL=ON ..  
cd provisioning_client\tools\tpm_device_provision  
make  
.\\tpm_device_provision
```

3. The output window displays the device's **Registration ID** and the **Endorsement key**. Copy these values for use later when you create an individual enrollment for your device in the device provisioning service.

TIP

If you don't want to use the SDK tool to retrieve the information, you need to find another way to obtain the provisioning information. The endorsement key, which is unique to each TPM chip, is obtained from the TPM chip manufacturer associated with it. You can derive a unique registration ID for your TPM device. For example, you can create an SHA-256 hash of the endorsement key.

After you have your registration ID and endorsement key, you're ready to continue.

Create a device provisioning service enrollment

Use your TPM's provisioning information to create an individual enrollment in the device provisioning service.

When you create an enrollment in the device provisioning service, you have the opportunity to declare an **Initial Device Twin State**. In the device twin, you can set tags to group devices by any metric used in your solution, like region, environment, location, or device type. These tags are used to create [automatic deployments](#).

TIP

The steps in this article are for the Azure portal, but you can also create individual enrollments by using the Azure CLI. For more information, see [az iot dps enrollment](#). As part of the CLI command, use the **edge-enabled** flag to specify that the enrollment is for an IoT Edge device.

1. In the [Azure portal](#), go to your instance of the IoT Hub device provisioning service.

2. Under **Settings**, select **Manage enrollments**.
3. Select **Add individual enrollment**, and then complete the following steps to configure the enrollment:
 - a. For **Mechanism**, select **TPM**.
 - b. Provide the **Endorsement key** and **Registration ID** that you copied from your VM or physical device.
 - c. Provide an ID for your device if you want. If you don't provide a device ID, the registration ID is used.
 - d. Select **True** to declare that your VM or physical device is an IoT Edge device.
 - e. Choose the linked IoT hub that you want to connect your device to, or select **Link to new IoT Hub**. You can choose multiple hubs, and the device will be assigned to one of them according to the selected assignment policy.
 - f. Add a tag value to the **Initial Device Twin State** if you want. You can use tags to target groups of devices for module deployment. For more information, see [Deploy IoT Edge modules at scale](#).
 - g. Select **Save**.

Now that an enrollment exists for this device, the IoT Edge runtime can automatically provision the device during installation.

Install IoT Edge

In this section, you prepare your Windows VM or physical device for IoT Edge. Then, you install IoT Edge.

Azure IoT Edge relies on an OCI-compatible container runtime. [Moby](#), a Moby-based engine, is included in the installation script, which means there are no additional steps to install the engine.

To install the IoT Edge runtime:

1. Run PowerShell as an administrator.

Use an AMD64 session of PowerShell, not PowerShell(x86). If you're unsure which session type you're using, run the following command:

```
(Get-Process -Id $PID).StartInfo.EnvironmentVariables["PROCESSOR_ARCHITECTURE"]
```

2. Run the [Deploy-IoTEdge](#) command, which performs the following tasks:

- Checks that your Windows machine is on a supported version
- Turns on the containers feature
- Downloads the moby engine and the IoT Edge runtime

```
. {Invoke-WebRequest -useb https://aka.ms/iotedge-win} | Invoke-Expression; `  
Deploy-IoTEdge
```

3. Restart your device if prompted.

When you install IoT Edge on a device, you can use additional parameters to modify the process including:

- Direct traffic to go through a proxy server
- Point the installer to a local directory for offline installation

For more information about these additional parameters, see [PowerShell scripts for IoT Edge with Windows containers](#).

Provision the device with its cloud identity

After the runtime is installed on your device, configure the device with the information it uses to connect to the device provisioning service and IoT Hub.

1. Know your device provisioning service **ID Scope** and device **Registration ID** that were gathered in the previous sections.
2. Open a PowerShell window in administrator mode. Be sure to use an AMD64 session of PowerShell when you install IoT Edge, not PowerShell (x86).
3. The `Initialize-IoTEdge` command configures the IoT Edge runtime on your machine. The command defaults to manual provisioning with Windows containers. Use the `-Dps` flag to use the device provisioning service instead of manual provisioning.

Replace the placeholder values for `paste_scope_id_here` and `paste_registration_id_here` with the data you collected earlier.

```
. {Invoke-WebRequest -useb https://aka.ms/iotedge-win} | Invoke-Expression; `  
Initialize-IoTEdge -Dps -ScopeId paste_scope_id_here -RegistrationId paste_registration_id_here
```

Verify successful installation

If the runtime started successfully, go into your IoT hub and start deploying IoT Edge modules to your device. Use the following commands on your device to verify that the runtime installed and started successfully.

1. Check the status of the IoT Edge service.

```
Get-Service iotedge
```

2. Examine service logs from the last 5 minutes.

```
. {Invoke-WebRequest -useb aka.ms/iotedge-win} | Invoke-Expression; Get-IoTEdgeLog
```

3. List running modules.

```
iotedge list
```

Next steps

The device provisioning service enrollment process lets you set the device ID and device twin tags at the same time as you provision the new device. You can use those values to target individual devices or groups of devices by using automatic device management.

Learn how to [deploy and monitor IoT Edge modules at scale by using the Azure portal](#) or [the Azure CLI](#).

Troubleshooting with Azure IoT Hub Device Provisioning Service

8/22/2022 • 3 minutes to read • [Edit Online](#)

Provisioning issues for IoT devices can be difficult to troubleshoot because there are many possible points of failures such as attestation failures, registration failures, etc. This article provides guidance on how to detect and troubleshoot device provisioning issues via Azure Monitor. To learn more about using Azure Monitor with DPS, see [Monitor Device Provisioning Service](#).

Using Azure Monitor to view metrics and set up alerts

To view and set up alerts on IoT Hub Device Provisioning Service metrics:

1. Sign in to the [Azure portal](#).
2. Browse to your IoT Hub Device Provisioning Service.
3. Select **Metrics**.
4. Select the desired metric. For supported metrics, see [Metrics](#).
5. Select desired aggregation method to create a visual view of the metric.
6. To set up an alert of a metric, select **New alert rules** from the top right of the metric blade, similarly you can go to **Alert** blade and select **New alert rules**.
7. Select **Add condition**, then select the desired metric and threshold by following prompts.

To learn more about viewing metrics and setting up alerts on your DPS instance, see [Analyzing metrics](#) and [Alerts](#) in Monitor Device Provisioning Service.

Using Log Analytics to view and resolve errors

1. Sign in to the [Azure portal](#).
2. Browse to your Device Provisioning Service.
3. Select **Diagnostics settings**.
4. Select **Add diagnostic setting**.
5. Configure the desired logs to be collected. For supported categories, see [Resource logs](#).
6. Tick the box **Send to Log Analytics** ([see pricing](#)) and save.
7. Go to **Logs** tab in the Azure portal under Device Provisioning Service resource.
8. Write **AzureDiagnostics** as a query and click **Run** to view recent events.
9. If there are results, look for `OperationName`, `ResultType`, `ResultSignature`, and `ResultDescription` (error message) to get more detail on the error.

Common error codes

Use this table to understand and resolve common errors.

Error code	Description	HTTP status code
400	The body of the request is not valid; for example, it cannot be parsed, or the object cannot be validated.	400 Bad format
401	The authorization token cannot be validated; for example, it is expired or does not apply to the request's URI. This error code is also returned to devices as part of the TPM attestation flow.	401 Unauthorized
404	The Device Provisioning Service instance, or a resource (e.g. an enrollment) does not exist.	404 Not Found
405	The client service knows the request method, but the target service doesn't recognize this method; for example, a rest operations is missing the enrollment or registration Id parameters	405 Method Not Allowed
409	The request could not be completed due to a conflict with the current state of the target Device Provisioning Service instance; for example, the customer has already created the data point and is attempting to recreate the same datapoint again.	409 Conflict
412	The ETag in the request does not match the ETag of the existing resource, as per RFC7232.	412 Precondition failed
415	The server refuses to accept the request because the payload format is in an unsupported format. For supported formats, see IoT Hub Device Provisioning Service REST API	415 Unsupported Media Type
429	Operations are being throttled by the service. For specific service limits, see IoT Hub Device Provisioning Service limits .	429 Too many requests
500	An internal error occurred.	500 Internal Server Error

If an IoT Edge device fails to start with error message

`failed to provision with IoT Hub, and no valid device backup was found dps client error.`, see [DPS Client error](#) in the IoT Edge (1.1) documentation.

Next Steps

- To learn more about using Azure Monitor with DPS, see [Monitor Device Provisioning Service](#).
- To learn about metrics, logs, and schemas emitted for DPS in Azure Monitor, see [Monitoring Device](#)

Microsoft SDKs for IoT Hub Device Provisioning Service

8/22/2022 • 2 minutes to read • [Edit Online](#)

The Azure IoT Hub Device Provisioning Service (DPS) is a helper service for IoT Hub. The DPS package provides SDKs to help you build backend and device applications that leverage DPS to provide zero-touch, just-in-time provisioning to one or more IoT hubs. The SDKs are published in a variety of popular languages and handle the underlying transport and security protocols between your devices or backend apps and DPS, freeing developers to focus on application development. Additionally, using the SDKs provides you with support for future updates to DPS, including security updates.

There are three categories of software development kits (SDKs) for working with DPS:

- [DPS device SDKs](#) provide data plane operations for devices. You use the device SDK to provision a device through DPS.
- [DPS service SDKs](#) provide data plane operations for backend apps. You can use the service SDKs to create and manage individual enrollments and enrollment groups, and to query and manage device registration records.
- [DPS management SDKs](#) provide control plane operations for backend apps. You can use the management SDKs to create and manage DPS instances and metadata. For example, to create and manage DPS instances in your subscription, to upload and verify certificates with a DPS instance, or to create and manage authorization policies or allocation policies in a DPS instance.

The DPS SDKs help you provision devices to your IoT hubs. Microsoft also provides a set of SDKs to help you build device apps and backend apps that communicate directly with Azure IoT Hub. For example, to help your provisioned devices send telemetry to your IoT hub, and, optionally, to receive messages and job, method, or twin updates from your IoT hub. To learn more, see [Azure IoT Hub SDKs](#).

Device SDKs

The DPS device SDKs provide implementations of the [Register](#) API and others that devices call to provision through DPS. The device SDKs can run on general MPU-based computing devices such as a PC, tablet, smartphone, or Raspberry Pi. The SDKs support development in C and in modern managed languages including in C#, NodeJS, Python, and Java.

PLATFORM	PACKAGE	CODE REPOSITORY	SAMPLES	QUICKSTART	REFERENCE
.NET	NuGet	GitHub	Samples	Quickstart	Reference
C	apt-get, MBED, Arduino IDE or iOS	GitHub	Samples	Quickstart	Reference
Java	Maven	GitHub	Samples	Quickstart	Reference
Node.js	npm	GitHub	Samples	Quickstart	Reference

PLATFORM	PACKAGE	CODE REPOSITORY	SAMPLES	QUICKSTART	REFERENCE
Python	pip	GitHub	Samples	Quickstart	Reference

WARNING

The C SDK listed above is **not** suitable for embedded applications due to its memory management and threading model. For embedded devices, refer to the [Embedded device SDKs](#).

Embedded device SDKs

These SDKs were designed and created to run on devices with limited compute and memory resources and are implemented using the C language.

RTOS	SDK	SOURCE	SAMPLES	REFERENCE
Azure RTOS	Azure RTOS Middleware	GitHub	Quickstarts	Reference
FreeRTOS	FreeRTOS Middleware	GitHub	Samples	Reference
Bare Metal	Azure SDK for Embedded C	GitHub	Samples	Reference

Learn more about the device and embedded device SDKs in the [IoT Device Development documentation](#).

Service SDKs

The DPS service SDKs help you build backend applications to manage enrollments and registration records in DPS instances.

PLATFORM	PACKAGE	CODE REPOSITORY	SAMPLES	QUICKSTART	REFERENCE
.NET	NuGet	GitHub	Samples	Quickstart	Reference
Java	Maven	GitHub	Samples	Quickstart	Reference
Node.js	npm	GitHub	Samples	Quickstart	Reference

Management SDKs

The DPS management SDKs help you build backend applications that manage the DPS instances and their metadata in your Azure subscription.

PLATFORM	PACKAGE	CODE REPOSITORY	REFERENCE
.NET	NuGet	GitHub	Reference
Java	Maven	GitHub	Reference
Node.js	npm	GitHub	Reference

PLATFORM	PACKAGE	CODE REPOSITORY	REFERENCE
Python	pip	GitHub	Reference

Next steps

The Device Provisioning Service documentation provides [tutorials](#) and [additional samples](#) that you can use to try out the SDKs and libraries.

What are the Azure IoT support and help options?

8/22/2022 • 2 minutes to read • [Edit Online](#)

Here are suggestions for where you can get help when developing your Azure IoT solutions.

Create an Azure support request

Explore the range of [Azure support options and choose the plan](#) that best fits, whether you're a developer just starting your cloud journey or a large organization deploying business-critical, strategic applications. Azure customers can create and manage support requests in the Azure portal.

- [Azure portal](#)
- [Azure portal for the United States government](#)

NOTE

Azure IoT solutions depend on services which may have different log retention periods. To help resolve your issue, please open a support request as soon as possible to help with troubleshooting.

Post a question on Microsoft Q&A

For quick and reliable answers on your technical product questions from Microsoft Engineers, Azure Most Valuable Professionals (MVPs), or our expert community, engage with us on [Microsoft Q&A](#), Azure's preferred destination for community support.

If you can't find an answer to your problem using search, submit a new question to Microsoft Q&A. Use one of the following tags when you ask your question:

- [Azure IoT](#)
- [Azure IoT Central](#)
- [Azure IoT Edge](#)
- [Azure IoT Hub](#)
- [Azure IoT Hub Device Provisioning Service \(DPS\)](#)
- [Azure IoT SDKs](#)
- [Azure Digital Twins](#)
- [Azure RTOS](#)
- [Azure Sphere](#)
- [Azure Time Series Insights](#)
- [Azure Maps](#)
- [Azure Percept](#)

Post a question on Stack Overflow

For answers on your developer questions from the largest community developer ecosystem, ask your question on Stack Overflow.

If you do submit a new question to Stack Overflow, please use one or more of the following tags when you create the question:

- [Azure IoT Central](#)
- [Azure IoT Edge](#)
- [Azure IoT Hub](#)
- [Azure IoT SDKs](#)
- [Azure Digital Twins](#)
- [Azure RTOS](#)
- [Azure Sphere](#)
- [Azure Time Series Insights](#)
- [Azure Percept](#)

Stay informed of updates and new releases



Learn about important product updates, roadmap, and announcements in [Azure Updates](#).

News and information about Azure IoT is shared at the [Azure blog](#) and on the [Internet of Things Show on Channel 9](#).

Also, share your experiences, engage and learn from experts in the [Internet of Things Tech Community](#).

Next steps

[What is Azure IoT?](#)

Glossary of IoT terms

8/22/2022 • 33 minutes to read • [Edit Online](#)

This article lists some of the common terms used in the IoT articles.

A

Advanced Message Queueing Protocol

One of the messaging protocols that [IoT Hub](#) and IoT Central support for communicating with [devices](#).

[Learn more](#)

Casing rules: Always *Advanced Message Queueing Protocol*.

First and subsequent mentions: Depending on the context spell out in full. Otherwise use the abbreviation AMQP.

Abbreviation: AMQP

Applies to: IoT Hub, IoT Central, Device developer

Allocation policy

In the [Device Provisioning Service](#), the allocation policy determines how the service assigns [devices](#) to a [Linked IoT hub](#).

Casing rules: Always lowercase.

Applies to: Device Provisioning Service

Attestation mechanism

In the [Device Provisioning Service](#), the attestation mechanism is the method used to confirm a [device's](#) identity. The attestation mechanism is configured on an [enrollment](#).

Attestation mechanisms include X.509 certificates, Trusted Platform [Modules](#), and symmetric keys.

Casing rules: Always lowercase.

Applies to: Device Provisioning Service

Automatic deployment

A feature in [IoT Edge](#) that 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](#)

Casing rules: Always lowercase.

Applies to: IoT Edge

Automatic device configuration

A feature of [IoT Hub](#) that enables your [solution](#) back end to assign [desired properties](#) to a set of [device twins](#) and report [device](#) status using system and custom metrics.

[Learn more](#)

Casing rules: Always lowercase.

Applies to: IoT Hub

Automatic device management

A feature of [IoT Hub](#) that automates many of the repetitive and complex tasks of managing large [device](#) fleets over the entirety of their lifecycles. The feature lets you 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](#).

Casing rules: Always lowercase.

Applies to: IoT Hub

Azure Certified Device program

Azure Certified [Device](#) is a free program that enables you to differentiate, certify, and promote your IoT devices built to run on Azure.

[Learn more](#)

Casing rules: Always capitalize as *Azure Certified Device*.

Applies to: IoT Hub, IoT Central

Azure Digital Twins

A platform as a service (PaaS) offering for creating digital representations of real-world things, places, business processes, and people. Build twin graphs that represent entire environments, and use them to gain insights to drive better products, optimize operations and costs, and create breakthrough customer experiences.

[Learn more](#)

Casing rules: Always capitalize when you're referring to the service.

First and subsequent mentions: When you're referring to the service, always spell out in full as *Azure Digital Twins*.

Example usage: The data in your *Azure Digital Twins* model can be routed to downstream Azure services for more analytics or storage.

Applies to: Digital Twins

Azure Digital Twins instance

A single instance of the [Azure Digital Twins](#) service in a customer's subscription. While Azure [Digital Twins](#) refers to the Azure service as a whole, your Azure Digital Twins *instance* is your individual Azure Digital Twins resource.

Casing rules: Always capitalize the service name.

First and subsequent mentions: Always spell out in full as *Azure Digital Twins instance*.

Applies to: Digital Twins

Azure IoT Explorer

A tool you can use 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 [IoT Plug and Play devices](#).

[Learn more](#)

Casing rules: Always capitalize as *Azure IoT Explorer*.

Applies to: IoT Hub, Device developer

Azure IoT Tools

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 can easily develop an IoT project in VS Code.

Casing rules: Always capitalize as *Azure IoT Tools*.

Applies to: IoT Hub, IoT Edge, IoT Central, Device developer

Azure IoT device SDKs

These SDKs, available for multiple languages, enable you to create device apps that interact with an IoT hub or an IoT Central application.

[Learn more](#)

Casing rules: Always refer to as *Azure IoT device SDKs*.

First and subsequent mentions: On first mention, always use *Azure IoT device SDKs*. On subsequent mentions abbreviate to *device SDKs*.

Example usage: The *Azure IoT device SDKs* are a set of device client libraries, developer guides, samples, and documentation. The *device SDKs* help you to programmatically connect devices to Azure IoT services.

Applies to: IoT Hub, IoT Central, Device developer

Azure IoT service SDKs

These SDKs, available for multiple languages, enable you to create back-end apps that interact with an IoT hub.

[Learn more](#)

Casing rules: Always refer to as *Azure IoT service SDKs*.

First and subsequent mentions: On first mention, always use *Azure IoT service SDKs*. On subsequent mentions abbreviate to *service SDKs*.

Applies to: IoT Hub

B

Back-end app

In the context of IoT Hub, 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 for simplicity many of the tutorials show back-end apps as console apps running on your local development machine.

Casing rules: Always lowercase.

Applies to: IoT Hub

Built-in endpoints

Endpoints built into IoT Hub. For example, every IoT hub includes a built-in endpoint that is Event Hubs-compatible.

Casing rules: Always lowercase.

Applies to: IoT Hub

C

Cloud gateway

A cloud-hosted app that enables connectivity for [devices](#) that cannot connect directly to [IoT Hub](#) or IoT Central. A [cloud gateway](#) is hosted in the cloud in contrast to a [field gateway](#) that runs local to your devices. A common use case for a cloud gateway is to implement protocol translation for your devices.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central

Cloud property

A feature in IoT Central that lets you store [device](#) metadata in the IoT Central application. Cloud [properties](#) are defined in the [device template](#), but aren't part of the [device model](#). Cloud properties are never synchronized with a device.

Casing rules: Always lowercase.

Applies to: IoT Central

Cloud-to-device

Messages sent from an [IoT hub](#) to a connected [device](#). Often, these messages are [commands](#) that instruct the device to take an action.

Casing rules: Always lowercase.

Abbreviation: Do not use *C2D*.

Applies to: IoT Hub

Command

A command is defined in an IoT Plug and Play [interface](#) to represent a method that can be called on the [digital twin](#). For example, a command to reboot a [device](#). In IoT Central, commands are defined in the [device template](#).

Applies to: IoT Hub, IoT Central, Device developer

Component

In IoT Plug and Play and [Azure Digital Twins](#), components let you build a [model interface](#) as an assembly of other interfaces. A [device model](#) can combine multiple interfaces as components. For example, a model might include a switch component and thermostat component. Multiple components in a model can also use the same interface type. For example, a model might include two thermostat components.

Casing rules: Always lowercase.

Applies to: IoT Hub, Digital Twins, Device developer

Configuration

In the context of [automatic device configuration](#) in [IoT Hub](#), it defines the [desired configuration](#) for a set of [devices](#) twins and provides a set of metrics to report status and progress.

Casing rules: Always lowercase.

Applies to: IoT Hub

Connection string

Use 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.

Casing rules: Always lowercase.

Applies to: IoT Hub, Device developer

Custom endpoints

User-defined [endpoints](#) on an [IoT hub](#) that deliver messages dispatched by a [routing rule](#). These endpoints connect directly to an event hub, a Service Bus queue, or a Service Bus topic.

Casing rules: Always lowercase.

Applies to: IoT Hub

Custom gateway

Enables connectivity for [devices](#) that cannot connect directly to [IoT Hub](#) or IoT Central. You can use Azure [IoT Edge](#) to build custom [gateways](#) that implement custom logic to handle messages, custom protocol conversions, and other processing.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central

D

Default component

All [IoT Plug and Play device models](#) have a default [component](#). A simple [device model](#) only has a default component - such a model is also known as a no-component [device](#). A more complex model has multiple components nested below the default component.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Device developer

Deployment manifest

A JSON document in [IoT Edge](#) that contains the [configuration](#) data for one or more [IoT Edge device module twins](#).

Casing rules: Always lowercase.

Applies to: IoT Edge, IoT Central

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.

Casing rules: Always lowercase.

Applies to: IoT Hub

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](#). IoT Central uses the term [writable properties](#).

Casing rules: Always lowercase.

Applies to: 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 certified devices.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, IoT Edge, Device Provisioning Service, Device developer

Device Provisioning Service

A helper service for [IoT Hub](#) and IoT Central that you use to configure zero-touch [device provisioning](#). With the DPS, you can provision millions of [devices](#) in a secure and scalable manner.

Casing rules: Always capitalized as *Device Provisioning Service*.

First and subsequent mentions: IoT Hub Device Provisioning Service

Abbreviation: DPS

Applies to: IoT Hub, Device Provisioning Service, IoT Central

Device REST API

A REST API you can use on 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 [Azure IoT device SDKs](#).

[Learn more](#)

Casing rules: Always *device REST API*.

Applies to: IoT Hub

Device app

A [device](#) app runs on your device and handles the communication with your [IoT hub](#) or IoT Central application. Typically, you use one of the [Azure IoT device SDKs](#) when you implement a device app.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Device developer

Device builder

The person responsible for creating the code to run on your [devices](#). Device builders typically use one of the [Azure IoT device SDKs](#) to implement the device client. A device builder uses a [device model](#) and [interfaces](#) when implementing code to run on an [IoT Plug and Play device](#).

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, IoT Edge, Device developer

Device identity

A unique identifier assigned to every [device](#) registered in the [IoT Hub identity registry](#) or in an IoT Central application.

Casing rules: Always lowercase. If you're using the abbreviation, *ID* is all upper case.

Abbreviation: Device ID

Applies to: IoT Hub, IoT Central

Device management

Device management encompasses the full lifecycle associated with managing the devices in your IoT [solution](#)

including planning, provisioning, configuring, monitoring, and retiring.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central

Device model

A description, that uses the [Digital Twins Definition Language](#), of the capabilities of a [device](#). Capabilities include [telemetry](#), [properties](#), and [commands](#).

[Learn more](#)

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Device developer, Digital Twins

Device provisioning

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](#). The [Device Provisioning Service](#) can automatically provision devices in an IoT hub or IoT Central application.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Device Provisioning Service

Device template

In IoT Central, a [device](#) template is a blueprint that defines the characteristics and behaviors of a type of device that connects to your application.

For example, the device template can define the [telemetry](#) that a device sends so that IoT Central can create visualizations that use the correct units and data types. A [device model](#) is part of the device template.

Casing rules: Always lowercase.

Abbreviation: Avoid abbreviating to *template* as IoT Central also has application templates.

Applies to: IoT Central

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 for the status of long-running operations.

See also [Digital twin](#)

Casing rules: Always lowercase.

Applies to: IoT Hub

Device-to-cloud

Refers to messages sent from a connected [device](#) to [IoT Hub](#) or IoT Central.

Casing rules: Always lowercase.

Abbreviation: Do not use *D2C*.

Applies to: IoT Hub

Digital Twins Definition Language

A JSON-LD language for describing [models](#) and [interfaces](#) for [IoT Plug and Play](#) devices and [Azure Digital Twins](#)

entities. The language enables the IoT platform and IoT [solutions](#) to use the semantics of the entity.

[Learn more](#)

First and subsequent mentions: Spell out in full as *Digital Twins Definition Language*.

Abbreviation: DTDL

Applies to: IoT Hub, IoT Central, Digital Twins

Digital twin

A digital twin is a collection of digital data that represents a physical object. Changes in the physical object are reflected in the digital twin. In some scenarios, you can use the digital twin to manipulate the physical object. The [Azure Digital Twins service](#) uses [models](#) expressed in the [Digital Twins Definition Language](#) to represent digital twins of [physical devices](#) or higher-level abstract business concepts, enabling a wide range of cloud-based digital twin [solutions](#). An [IoT Plug and Play device](#) has a digital twin, described by a Digital Twins Definition Language [device model](#).

See also [Device twin](#)

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Digital Twins, Device developer

Digital twin change events

When an [IoT Plug and Play device](#) is connected to an [IoT hub](#), the hub can use its routing capability to send notifications of [digital twin](#) changes. The IoT Central data export feature can also forward digital twin change events to other services. For example, whenever a property value changes on a [device](#), IoT Hub can send a notification to an [endpoint](#) such as an event hub.

Casing rules: Always lowercase.

Abbreviation: Always spell out in full to distinguish from other types of change event.

Applies to: IoT Hub, IoT Central

Digital twin graph

In the [Azure Digital Twins](#) service, you can connect [digital twins](#) with [relationships](#) to create knowledge graphs that digitally represent your entire physical environment. A single [Azure Digital Twins instance](#) can host many disconnected graphs, or one single interconnected graph.

Casing rules: Always lowercase.

First and subsequent mentions: Use *digital twin graph* on first mention, then use *twin graph*.

Applies to: IoT Hub

Direct method

A way to trigger a method to execute on a [device](#) by invoking an API on your [IoT hub](#).

Casing rules: Always lowercase.

Applies to: IoT Hub

Downstream service

A relative term describing services that receive data from the current context. For example, in the context of [Azure Digital Twins](#), Time Series Insights is a downstream service if you set up your data to flow from Azure [Digital Twins](#) into Time Series Insights.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Digital Twins

E

Endpoint

A named representation of a data routing service that can receive data from other services.

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. There are service-facing management endpoints that enable [back-end apps](#) to perform operations such as [device identity](#) 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](#).

Casing rules: Always lowercase.

Applies to: IoT Hub

Enrollment

In the [Device Provisioning Service](#), an enrollment is the record of individual [devices](#) or groups of devices that may register with a [linked IoT hub](#) through autoprovisioning.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Device Provisioning Service

Enrollment group

In the [Device Provisioning Service](#) and IoT Central, an [enrollment](#) group identifies a group of [devices](#) that share an X.509 or symmetric key [attestation mechanism](#).

Casing rules: Always lowercase.

Applies to: IoT Hub, Device Provisioning Service, IoT Central

Event Hubs-compatible endpoint

An [IoT Hub endpoint](#) that lets you use any Event Hubs-compatible method to read [device](#) messages sent to the hub. Event Hubs-compatible methods include the [Event Hubs SDKs](#) and [Azure Stream Analytics](#).

Casing rules: Always lowercase.

Applies to: IoT Hub

Event handler

A process that's triggered by the arrival of an event. For example, you can create event handlers by adding event processing code to an Azure function, and sending data to it using [endpoints](#) and [event routing](#).

Casing rules: Always lowercase.

Applies to: IoT Hub

Event routing

The process of sending events and their data from one [device](#) or service to the [endpoint](#) of another.

In [IoT Hub](#), you can define [routing rules](#) to describe how messages should be sent. In [Azure Digital Twins](#), event routes are entities that are created for this purpose. Azure [Digital Twins](#) event routes can contain filters to limit what types of events are sent to each endpoint.

Casing rules: Always lowercase.

Applies to: IoT Hub, Digital Twins

F

Field gateway

Enables connectivity for [devices](#) that can't connect directly to [IoT Hub](#) and is typically deployed locally with your devices.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central

G

Gateway

A gateway enables connectivity for [devices](#) that cannot connect directly to [IoT Hub](#). See also [field gateway](#), [cloud gateway](#), and [custom gateway](#).

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central

Gateway device

An example of a [field gateway](#). A [gateway device](#) can be standard IoT device or an [IoT Edge device](#).

A gateway device enables connectivity for downstream devices that cannot connect directly to [IoT Hub](#).

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, IoT Edge

H

Hardware security module

Used for secure, hardware-based storage of [device](#) secrets. It's the most secure form of secret storage for a device. A hardware security [module](#) can store both X.509 certificates and symmetric keys. In the [Device Provisioning Service](#), an [attestation mechanism](#) can use a hardware security module.

Casing rules: Always lowercase.

First and subsequent mentions: Spell out in full on first mention as *hardware security module*.

Abbreviation: HSM

Applies to: IoT Hub, Device developer, Device Provisioning Service

I

ID scope

A unique value assigned to a [Device Provisioning Service](#) instance when it's created.

IoT Central applications make use of DPS instances and make the ID scope available through the IoT Central UI.

Casing rules: Always use *ID scope*.

Applies to: IoT Hub, IoT Central, Device Provisioning Service

Identity registry

A built-in [component](#) of an [IoT hub](#) that stores information about the individual [devices](#) permitted to connect to the hub.

Casing rules: Always lowercase.

Applies to: IoT Hub

Individual enrollment

Identifies a single [device](#) that the [Device Provisioning Service](#) can provision to an [IoT hub](#).

Casing rules: Always lowercase.

Applies to: IoT Hub, Device Provisioning Service

Industry 4.0

Refers to the fourth revolution that's occurred in manufacturing. Companies can build connected [solutions](#) to manage the manufacturing facility and equipment more efficiently by enabling manufacturing equipment to be cloud connected, allowing remote access and management from the cloud, and enabling OT personnel to have a single pane view of their entire facility.

Applies to: IoT Hub, IoT Central

Interface

In IoT Plug and Play, an interface describes related capabilities that are implemented by a [IoT Plug and Play device](#) or [digital twin](#). You can reuse interfaces across different [device models](#). When an interface is used in a [device model](#), it defines a [component](#) of the device. A simple device only contains a default interface.

In [Azure Digital Twins](#), *interface* may be used to refer to the top-level code item in a [Digital Twins Definition Language](#) model definition.

Casing rules: Always lowercase.

Applies to: Device developer, Digital Twins

IoT Edge

A service and related client libraries and runtime that 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 sending the data to the cloud.

[Learn more](#)

Casing rules: Always capitalize as *IoT Edge*.

First and subsequent mentions: Spell out as *Azure IoT Edge*.

Applies to: IoT Edge

IoT Edge agent

The part of the [IoT Edge runtime](#) responsible for deploying and monitoring [modules](#).

Casing rules: Always capitalize as *IoT Edge agent*.

Applies to: IoT Edge

IoT Edge device

A [device](#) that uses containerized [IoT Edge modules](#) to run Azure services, third-party services, or your own code. On the device, the [IoT Edge runtime](#) manages the modules. You can remotely monitor and manage an IoT Edge device from the cloud.

Casing rules: Always capitalize as *IoT Edge device*.

Applies to: IoT Edge

IoT Edge hub

The part of the [IoT Edge runtime](#) responsible for [module](#) to module, upstream, and downstream communications.

Casing rules: Always capitalize as *IoT Edge hub*.

Applies to: IoT Edge

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.

Casing rules: Always capitalize as *IoT Edge runtime*.

Applies to: IoT Edge

IoT Hub

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.

[Learn more](#)

Casing rules: When referring to the service, capitalize as *IoT Hub*. When referring to an instance, capitalize as *IoT hub*.

First and subsequent mentions: Spell out in full as *Azure IoT Hub*. Subsequent mentions can be *IoT Hub*. If the context is clear, use *hub* to refer to an instance.

Example usage: The Azure IoT Hub service enables secure, bidirectional communication. The device sends data to your IoT hub.

Applies to: IoT Hub

IoT Hub Resource REST API

An API you can use to manage the [IoT hubs](#) in your Azure subscription with operations such as creating, updating, and deleting hubs.

Casing rules: Always capitalize as *IoT Hub Resource REST API*.

Applies to: IoT Hub

IoT Hub metrics

A feature in the Azure portal that lets you monitor the state of your [IoT hubs](#). IoT Hub metrics enable you to assess the overall health of an IoT hub and the [devices](#) connected to it.

Casing rules: Always capitalize as *IoT Hub metrics*.

Applies to: IoT Hub

IoT Hub query language

A SQL-like language for [IoT Hub](#) that lets you query your [jobs](#), [digital twins](#), and [device twins](#).

Casing rules: Always capitalize as *IoT Hub query language*.

First and subsequent mentions: Spell out in full as *IoT Hub query language*, if the context is clear subsequent mentions can be *query language*.

Applies to: IoT Hub

IoT Plug and Play bridge

An open-source application that enables existing sensors and peripherals attached to Windows or Linux [gateways](#) to connect as [IoT Plug and Play devices](#).

Casing rules: Always capitalize as *IoT Plug and Play bridge*.

First and subsequent mentions: Spell out in full as *IoT Plug and Play bridge*. If the context is clear, subsequent mentions can be *bridge*.

Applies to: IoT Hub, Device developer, IoT Central

IoT Plug and Play conventions

A set of conventions that IoT [devices](#) should follow when they exchange data with a [solution](#).

Casing rules: Always capitalize as *IoT Plug and Play conventions*.

Applies to: IoT Hub, IoT Central, Device developer

IoT Plug and Play device

Typically a small-scale, standalone computing [device](#) that collects data or controls other devices, and that runs software or firmware that implements a [device model](#). For example, an IoT Plug and Play device might be an environmental monitoring device, or a controller for a smart-agriculture irrigation system. An IoT Plug and Play device might be implemented directly or as an [IoT Edge module](#).

Casing rules: Always capitalize as *IoT Plug and Play device*.

Applies to: IoT Hub, IoT Central, Device developer

IoT extension for Azure CLI

An extension for the Azure CLI. The extension lets you complete tasks such as managing your [devices](#) in the [identity registry](#), sending and receiving device messages, and monitoring your [IoT hub](#) operations.

[Learn more](#)

Casing rules: Always capitalize as *IoT extension for Azure CLI*.

Applies to: IoT Hub, IoT Central, IoT Edge, Device Provisioning Service, Device developer

J

Job

In the context of [IoT Hub](#), jobs let you 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 jobs to import to and export from the [identity registry](#).

In the context of IoT Central, jobs let you manage your connected devices in bulk by setting [properties](#) and calling [commands](#). IoT Central jobs also let you update cloud properties in bulk.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central

L

Leaf device

A [device](#) with no downstream devices connected. Typically leaf devices are connected to a [gateway device](#).

Casing rules: Always lowercase.

Applies to: IoT Edge, IoT Central, Device developer

Lifecycle event

In [Azure Digital Twins](#), this type of event is fired when a data item—such as a [digital twin](#), a [relationship](#), or an [event handler](#) is created or deleted from your [Azure Digital Twins instance](#).

Casing rules: Always lowercase.

Applies to: Digital Twins, IoT Hub, IoT Central

Linked IoT hub

An [IoT hub](#) that is linked to a [Device Provisioning Service](#) instance. A DPS instance can register a [device](#) ID and set the initial [configuration](#) in the [device twins](#) in linked IoT hubs.

Casing rules: Always capitalize as *linked IoT hub*.

Applies to: IoT Hub, Device Provisioning Service

M

MQTT

One of the messaging protocols that [IoT Hub](#) and IoT Central support for communicating with [devices](#). MQTT doesn't stand for anything.

[Learn more](#)

First and subsequent mentions: MQTT

Abbreviation: MQTT

Applies to: IoT Hub, IoT Central, Device developer

Model

A definition of a type of entity in your physical environment, including its [properties](#), telemetries, and [components](#). Models are used to create [digital twins](#) that represent specific physical objects of this type. Models are written in the [Digital Twins Definition Language](#).

In the [Azure Digital Twins](#) service, models define [devices](#) or higher-level abstract business concepts. In IoT Plug and Play, [device models](#) describe devices specifically.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Digital Twins, Device developer

Model ID

When an [IoT Plug and Play device](#) connects to an [IoT Hub](#) or IoT Central application, it sends the [model ID](#) of the [Digital Twins Definition Language](#) model it implements. Every model has a unique model ID. This model ID enables the [solution](#) to find the [device model](#).

Casing rules: Always capitalize as *model ID*.

Applies to: IoT Hub, IoT Central, Device developer, Digital Twins

Model repository

Stores [Digital Twins Definition Language](#) [models](#) and [interfaces](#). A [solution](#) uses a [model ID](#) to retrieve a model from a repository.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Digital Twins

Model repository REST API

An API for managing and interacting with a [model repository](#). For example, you can use the API to add and search for [device models](#).

Casing rules: Always capitalize as *model repository REST API*.

Applies to: IoT Hub, IoT Central, Digital Twins

Module

The [IoT Hub device](#) SDKs let you instantiate modules where each one opens an independent connection to your IoT hub. This lets you 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.

In [IoT Edge](#), a 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](#).

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Edge, Device developer

Module identity

A unique identifier assigned to every [module](#) that belongs to a [device](#). Module identities are also registered in the [identity registry](#).

The module identity details the security credentials the module uses to authenticate with the [IoT Hub](#) or, in the case of an [IoT Edge](#) module to the [IoT Edge hub](#).

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Edge, Device developer

Module image

The docker image the [IoT Edge runtime](#) uses to instantiate [module](#) instances.

Casing rules: Always lowercase.

Applies to: IoT Edge

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.

Casing rules: Always lowercase.

Applies to: IoT Hub

O

Ontology

In the context of [Digital Twins](#), a set of [models](#) for a particular domain, such as real estate, smart cities, IoT systems, energy grids, and more. Ontologies are often used as schemas for knowledge graphs like the ones in [Azure Digital Twins](#), because they provide a starting point based on industry standards and best practices.

[Learn more](#)

Applies to: Digital Twins

Operational technology

That hardware and software in an industrial facility that monitors and controls equipment, processes, and infrastructure.

Casing rules: Always lowercase.

Abbreviation: OT

Applies to: IoT Hub, IoT Central, IoT Edge

Operations monitoring

A feature of [IoT Hub](#) that lets you 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.

Casing rules: Always lowercase.

Applies to: IoT Hub

P

Physical device

A real IoT [device](#) that connects to an [IoT hub](#). For convenience, many tutorials and quickstarts run IoT device code on a desktop machine rather than a physical device.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Device developer, IoT Edge

Primary and secondary keys

When you connect to a [device-facing](#) or service-facing [endpoint](#) on an [IoT hub](#) or IoT Central application, 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 or IoT Central application.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central

Properties

In the context of a [digital twin](#), data fields defined in an [interface](#) that represent some persistent state of the digital twin. You can declare properties as read-only or writable. Read-only properties, such as serial number, are set by code running on the [IoT Plug and Play device](#) itself. Writable properties, such as an alarm threshold, are typically set from the cloud-based IoT [solution](#).

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Digital Twins, Device developer

Property change event

An event that results from a property change in a [digital twin](#).

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Digital Twins

Protocol gateway

A [gateway](#) typically deployed in the cloud to provide protocol translation services for [devices](#) connecting to an [IoT hub](#) or IoT Central application.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central

R

Registration

A record of a [device](#) in the [IoT Hub identity registry](#). You can register or device directly, or use the [Device Provisioning Service](#) to automate device registration.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Device Provisioning Service

Registration ID

A unique [device identity](#) in the [Device Provisioning Service](#). The [registration ID](#) may be the same value as the [device identity](#).

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Device Provisioning Service

Relationship

Used in the [Azure Digital Twins](#) service to connect [digital twins](#) into knowledge graphs that digitally represent your entire physical environment. The types of relationships that your twins can have are defined in the [Digital Twins Definition Language model](#).

Casing rules: Always lowercase.

Applies to: Digital Twins

Reported configuration

In the context of a [device twin](#), this refers to the complete set of [properties](#) and metadata in the [device](#) twin that are reported to the [solution](#) back end.

Casing rules: Always lowercase.

Applies to: IoT Hub, Device developer

Reported properties

In the context of a [device twin](#), reported [properties](#) is a subsection of the [device](#) twin. Reported properties can only be set by the device but can be read and queried by a [back-end app](#).

Casing rules: Always lowercase.

Applies to: IoT Hub, Device developer

Retry policy

A way to handle transient errors when you connect to a cloud service.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Device developer

Routing rule

A feature of [IoT Hub](#) used to route [device-to-cloud](#) messages to a built-in [endpoint](#) or to [custom endpoints](#) for processing by your [solution](#) back end.

Casing rules: Always lowercase.

Applies to: IoT Hub

S

SASL/PLAIN

A protocol that [Advanced Message Queueing Protocol](#) uses to transfer security tokens.

[Learn more](#)

Abbreviation: SASL/PLAIN

Applies to: IoT Hub

Service REST API

A REST API you can use from the [solution](#) back end to manage your [devices](#). For example, you can use the [IoT Hub](#) service API to retrieve and update [device twin properties](#), invoke [direct methods](#), and schedule [jobs](#).

Typically, you should use one of the higher-level service SDKs.

Casing rules: Always *service REST API*.

Applies to: IoT Hub, IoT Central, Device Provisioning Service, IoT Edge

Service operations endpoint

An [endpoint](#) that an administrator uses to manage service settings. For example, in the [Device Provisioning Service](#) you use the service endpoint to manage [enrollments](#).

Casing rules: Always lowercase.

Applies to: IoT Hub, Device Provisioning Service, IoT Edge, Digital Twins

Shared access policy

A way to define 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.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Edge, Device Provisioning Service

Shared access signature

A shared access signature is a signed URI that points to one or more resources such as an [IoT hub endpoint](#). The URI includes a token that indicates how the resources can be accessed by the client. One of the query parameters, the signature, is constructed from the SAS parameters and signed with the key that was used to create the SAS. This signature is used by Azure Storage to authorize access to the storage resource.

Casing rules: Always lowercase.

Abbreviation: SAS

Applies to: IoT Hub, Digital Twins, IoT Central, IoT Edge

Simulated device

For convenience, many of the tutorials and quickstarts run [device](#) code with simulated sensors on your local development machine. In contrast, a [physical device](#) such as an MXCHIP has real sensors and connects to an [IoT](#)

hub.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Device developer, IoT Edge, Digital Twins, Device Provisioning Service

Solution

In the context of IoT, *solution* typically refers to an IoT solution that includes elements such as [devices](#), [device apps](#), an [IoT hub](#), other Azure services, and [back-end apps](#).

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Device Provisioning Service, IoT Edge, Digital Twins

System properties

In the context of a [device twin](#), the read-only [properties](#) that include information regarding the [device](#) usage such as last activity time and connection state.

Casing rules: Always lowercase.

Applies to: IoT Hub

T

Tag

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.

Casing rules: Always lowercase.

Applies to: IoT Hub

Target condition

In an [IoT Edge](#) deployment, the target condition selects the target [devices](#) of the deployment. The target condition is continuously evaluated to include any new devices that meet the requirements or remove devices that no longer do.

Casing rules: Always lowercase.

Applies to: IoT Edge

Telemetry

The data, such as wind speed or temperature, sent to an [IoT hub](#) that was collected by a [device](#) from its sensors.

Unlike [properties](#), telemetry is not stored on a [digital twin](#); it is a stream of time-bound data events that need to be handled as they occur.

In IoT Plug and Play and [Azure Digital Twins](#), telemetry fields defined in an [interface](#) represent measurements. These measurements are typically values such as sensor readings that are sent by devices, like [IoT Plug and Play devices](#), as a stream of data.

Casing rules: Always lowercase.

Example usage: Don't use the word *telemetries*, telemetry refers to the collection of data a device sends. For example: When the device connects to your IoT hub, it starts sending telemetry. One of the telemetry values the device sends is the environmental temperature.

Applies to: IoT Hub, IoT Central, Digital Twins, IoT Edge, Device developer

Telemetry event

An event in an [IoT hub](#) that indicates the arrival of [telemetry](#) data.

Casing rules: Always lowercase.

Applies to: IoT Hub

Twin queries

A feature of [IoT Hub](#) that lets you use a SQL-like query language to retrieve information from your [device twins](#) or [module twins](#).

Casing rules: Always lowercase.

Applies to: IoT Hub

Twin synchronization

The process in [IoT Hub](#) that 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.

Casing rules: Always lowercase.

Applies to: IoT Hub

U

Upstream service

A relative term describing services that feed data into the current context. For instance, in the context of [Azure Digital Twins](#), [IoT Hub](#) is considered an upstream service because data flows from IoT Hub into Azure [Digital Twins](#).

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Digital Twins

Summary of customer data request features

8/22/2022 • 3 minutes to read • [Edit Online](#)

The Azure IoT Hub Device Provisioning Service is a REST API-based cloud service targeted at enterprise customers that enables seamless, automated zero-touch provisioning of devices to Azure IoT Hub with security that begins at the device and ends with the cloud.

NOTE

This article provides steps about how to delete personal data from the device or service and can be used to support your obligations under the GDPR. For general information about GDPR, see the [GDPR section of the Microsoft Trust Center](#) and the [GDPR section of the Service Trust portal](#).

Individual devices are assigned a registration ID and device ID by a tenant administrator. Data from and about these devices is based on these IDs. Microsoft maintains no information and has no access to data that would allow correlation of these devices to an individual.

Many of the devices managed in Device Provisioning Service 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. Device Provisioning Service 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 or registration ID.

NOTE

Devices that have been provisioned in Azure IoT Hub through Device Provisioning Service have additional data stored in the Azure IoT Hub service. See the [Azure IoT Hub reference documentation](#) in order to complete a full request for a given device.

Deleting customer data

Device Provisioning Service stores enrollments and registration records. Enrollments contain information about devices that are allowed to be provisioned, and registration records show which devices have already gone through the provisioning process.

Tenant administrators may remove enrollments from the Azure portal, and this removes any associated registration records as well.

For more information, see [How to manage device enrollments](#).

It is also possible to perform delete operations for enrollments and registration records using REST APIs:

- To delete enrollment information for a single device, you can use [Device Enrollment - Delete](#).
- To delete enrollment information for a group of devices, you can use [Device Enrollment Group - Delete](#).
- To delete information about devices that have been provisioned, you can use [Registration State - Delete Registration State](#).

Exporting customer data

Device Provisioning Service stores enrollments and registration records. Enrollments contain information about devices that are allowed to be provisioned, and registration records show which devices have already gone through the provisioning process.

Tenant administrators can view enrollments and registration records through the Azure portal and export them using copy and paste.

For more information on how to manage enrollments, see [How to manage device enrollments](#).

It is also possible to perform export operations for enrollments and registration records using REST APIs:

- To export enrollment information for a single device, you can use [Device Enrollment - Get](#).
- To export enrollment information for a group of devices, you can use [Device Enrollment Group - Get](#).
- To export information about devices that have already been provisioned, you can use [Registration State - Get Registration State](#).

NOTE

When you use Microsoft's enterprise services, Microsoft generates some information, known as system-generated logs. Some Device Provisioning Service 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

For full documentation of Device Provisioning Service APIs, see [Azure IoT Hub Device Provisioning Service REST API](#).

Azure IoT Hub [customer data request features](#).