
AWS IoT Core

Developer Guide



AWS IoT Core: Developer Guide

Copyright © Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

What is AWS IoT?	1
How your devices and apps access AWS IoT	1
What AWS IoT can do	2
IoT in Industry	2
IoT in Home automation	3
How AWS IoT works	3
The IoT universe	3
AWS IoT services overview	5
AWS IoT Core services	8
Learn more about AWS IoT	11
Training resources for AWS IoT	11
AWS IoT resources and guides	11
AWS IoT in social media	11
AWS services used by the AWS IoT Core rules engine	12
Communication protocols supported by AWS IoT Core	13
What's new in the AWS IoT console	13
Legend	15
Getting started with AWS IoT Core	16
Connect your first device to AWS IoT Core	16
Set up your AWS account	17
Sign up for an AWS account	17
Create a user and grant permissions	18
Open the AWS IoT console	19
Try the AWS IoT Core interactive demo	19
Try the AWS IoT quick connect	23
Step 1. Start the tutorial	24
Step 2. Create a thing object	26
Step 3. Download files to your device	28
Step 4. Run the sample	30
Step 5. Explore further	32
Explore AWS IoT Core services in hands-on tutorial	33
Which device option is the best for you?	34
Create AWS IoT resources	34
Configure your device	38
View MQTT messages with the AWS IoT MQTT client	62
Viewing MQTT messages in the MQTT client	62
Publishing MQTT messages from the MQTT client	64
Connecting to AWS IoT Core	66
AWS IoT Core - control plane endpoints	66
AWS IoT device endpoints	67
AWS IoT Core for LoRaWAN gateways and devices	68
Connecting to AWS IoT Core service endpoints	68
AWS CLI for AWS IoT Core	69
AWS SDKs	69
AWS Mobile SDKs	73
REST APIs of the AWS IoT Core services	74
Connecting devices to AWS IoT	74
AWS IoT device data and service endpoints	74
AWS IoT Device SDKs	76
Device communication protocols	77
MQTT topics	93
Configurable endpoints	109
Connecting to AWS IoT FIPS endpoints	114
AWS IoT Core - control plane endpoints	114

AWS IoT Core - data plane endpoints	114
AWS IoT Device Management - jobs data endpoints	115
AWS IoT Device Management - Fleet Hub endpoints	115
AWS IoT Device Management - secure tunneling endpoints	115
AWS IoT tutorials	116
Building demos with the AWS IoT Device Client	116
Prerequisites to building demos with the AWS IoT Device Client	116
Preparing your devices for the AWS IoT Device Client	118
Installing and configuring the AWS IoT Device Client	128
Demonstrate MQTT message communication with the AWS IoT Device Client	136
Demonstrate remote actions (jobs) with the AWS IoT Device Client	150
Cleaning up	159
Building solutions with the AWS IoT Device SDKs	166
Start building solutions with the AWS IoT Device SDKs	166
Connecting a device to AWS IoT Core by using the AWS IoT Device SDK	166
Creating AWS IoT rules to route device data to other services	182
Retaining the device state while the device is offline	211
Creating a custom authorizer for AWS IoT Core	231
Monitoring soil moisture with AWS IoT and Raspberry Pi	242
Managing devices with AWS IoT	251
How to manage things with the registry	251
Create a thing	251
List things	252
Describe things	253
Update a thing	254
Delete a thing	254
Attach a principal to a thing	254
Detach a principal from a thing	254
Thing types	255
Create a thing type	255
List thing types	255
Describe a thing type	256
Associate a thing type with a thing	256
Deprecate a thing type	257
Delete a thing type	258
Static thing groups	258
Create a static thing group	259
Describe a thing group	260
Add a thing to a static thing group	260
Remove a thing from a static thing group	261
List things in a thing group	261
List thing groups	261
List groups for a thing	263
Update a static thing group	263
Delete a thing group	263
Attach a policy to a static thing group	264
Detach a policy from a static thing group	264
List the policies attached to a static thing group	264
List the groups for a policy	265
Get effective policies for a thing	265
Test authorization for MQTT actions	266
Dynamic thing groups	267
Create a dynamic thing group	268
Describe a dynamic thing group	268
Update a dynamic thing group	269
Delete a dynamic thing group	270
Limitations and conflicts	270

Tagging your AWS IoT resources	273
Tag basics	273
Tag restrictions and limitations	274
Using tags with IAM policies	274
Billing groups	276
Viewing cost allocation and usage data	276
Security	278
Security in AWS IoT	278
Authentication	279
AWS training and certification	279
X.509 Certificate overview	279
Server authentication	280
Client authentication	282
Custom authentication	301
Authorization	312
AWS training and certification	314
AWS IoT Core policies	314
Authorizing direct calls to AWS services using AWS IoT Core credential provider	355
Cross account access with IAM	359
Data protection	361
Data encryption in AWS IoT	361
Transport security in AWS IoT	362
Data encryption	363
Identity and access management	364
Audience	364
Authenticating with IAM identities	364
Managing access using policies	366
How AWS IoT works with IAM	368
Identity-based policy examples	385
Troubleshooting	388
Logging and Monitoring	390
Monitoring Tools	390
Compliance validation	391
Resilience	391
Using AWS IoT Core with VPC endpoints	392
Creating VPC endpoints for AWS IoT Core	392
Controlling Access to AWS IoT Core over VPC endpoints	393
Limitations of VPC endpoints	394
Scaling VPC endpoints with IoT Core	394
Using custom domains with VPC endpoints	394
Availability of VPC endpoints for AWS IoT Core	394
Infrastructure security	394
Security monitoring	395
Security best practices	395
Protecting MQTT connections in AWS IoT	395
Keep your device's clock in sync	397
Validate the server certificate	397
Use a single identity per device	398
Use a second AWS Region as backup	398
Use just in time provisioning	398
Permissions to run AWS IoT Device Advisor tests	398
AWS training and certification	399
Monitoring AWS IoT	400
Configure AWS IoT logging	400
Configure logging role and policy	401
Configure default logging in the AWS IoT (console)	402
Configure default logging in AWS IoT (CLI)	403

Configure resource-specific logging in AWS IoT (CLI)	404
Log levels	406
Monitor AWS IoT alarms and metrics using Amazon CloudWatch	406
Using AWS IoT metrics	407
Creating CloudWatch alarms in AWS IoT	407
AWS IoT metrics and dimensions	410
Monitor AWS IoT using CloudWatch Logs	421
Viewing AWS IoT logs in the CloudWatch console	421
CloudWatch AWS IoT log entries	422
Log AWS IoT API calls using AWS CloudTrail	440
AWS IoT information in CloudTrail	440
Understanding AWS IoT log file entries	441
Rules	443
Granting AWS IoT the required access	444
Pass role permissions	445
Creating an AWS IoT rule	446
Viewing your rules	450
Deleting a rule	450
AWS IoT rule actions	450
Apache Kafka	452
CloudWatch alarms	459
CloudWatch Logs	460
CloudWatch metrics	461
DynamoDB	463
DynamoDBv2	465
Elasticsearch	466
HTTP	468
IoT Analytics	495
IoT Events	496
IoT SiteWise	498
Kinesis Data Firehose	502
Kinesis Data Streams	503
Lambda	505
OpenSearch	507
Republish	509
S3	510
Salesforce IoT	511
SNS	512
SQS	514
Step Functions	515
Timestream	516
Troubleshooting a rule	521
Accessing cross-account resources using AWS IoT rules	521
Prerequisites	521
Cross-account setup for Amazon SQS	521
Cross-account setup for Amazon SNS	522
Cross-account setup for Amazon S3	523
Cross-account setup for AWS Lambda	525
Error handling (error action)	526
Error action message format	527
Error action example	528
Reducing messaging costs with basic ingest	528
Using basic ingest	529
AWS IoT SQL reference	529
SELECT clause	530
FROM clause	531
WHERE clause	532

Data types	533
Operators	536
Functions	541
Literals	583
Case statements	584
JSON extensions	584
Substitution templates	586
Nested object queries	587
Binary payloads	588
SQL versions	589
Device Shadow service	591
Using shadows	591
Choosing to use named or unnamed shadows	591
Accessing shadows	592
Using shadows in devices, apps, and other cloud services	592
Message order	593
Trim shadow messages	594
Using shadows in devices	594
Initializing the device on first connection to AWS IoT	595
Processing messages while the device is connected to AWS IoT	596
Processing messages when the device reconnects to AWS IoT	597
Using shadows in apps and services	597
Initializing the app or service on connection to AWS IoT	598
Processing state changes while the app or service is connected to AWS IoT	598
Detecting a device is connected	598
Simulating Device Shadow service communications	599
Setting up the simulation	600
Initialize the device	600
Send an update from the app	602
Respond to update in device	604
Observe the update in the app	608
Going beyond the simulation	609
Interacting with shadows	609
Protocol support	609
Requesting and reporting state	610
Updating a shadow	610
Retrieving a shadow document	613
Deleting shadow data	613
Device Shadow REST API	615
GetThingShadow	616
UpdateThingShadow	617
DeleteThingShadow	618
ListNamedShadowsForThing	618
Device Shadow MQTT topics	619
/get	620
/get/accepted	621
/get/rejected	621
/update	622
/update/delta	623
/update/accepted	624
/update/documents	624
/update/rejected	625
/delete	626
/delete/accepted	626
/delete/rejected	627
Device Shadow service documents	627
Shadow document examples	628

Document properties	632
Delta state	633
Versioning shadow documents	634
Client tokens in shadow documents	634
Empty shadow document properties	635
Array values in shadow documents	635
Device Shadow error messages	636
Jobs	637
Jobs key concepts	637
Managing jobs	639
Code signing for jobs	639
Job document	639
Presigned URLs	639
Create and manage jobs by using the AWS Management Console	640
Create and manage jobs by using the AWS CLI	641
Job templates	649
Custom and AWS managed templates	649
Deploy common remote operations by using AWS managed templates	649
Create custom job templates	660
Devices and jobs	665
Programming devices to work with jobs	666
Job configurations	675
Specify additional configurations	676
Job rollout and abort configuration	676
Job executions timeout configuration	678
AWS IoT jobs APIs	679
Job management and control API	679
Job management and control data types	680
Jobs device MQTT and HTTPS APIs	680
Jobs device MQTT and HTTPS data types	681
Job limits	683
AWS IoT secure tunneling	684
Secure tunneling concepts	684
Secure tunnel lifecycle	684
Controlling access to tunnels	685
Tunnel access prerequisites	685
<code>iot:OpenTunnel</code>	685
<code>iot:DescribeTunnel</code>	686
<code>iot>ListTunnels</code>	687
<code>iot>ListTagsForResource</code>	687
<code>iot:CloseTunnel</code>	688
<code>iot:TagResource</code>	688
<code>iot:UntagResource</code>	688
AWS IoT Secure Tunneling tutorials	689
AWS IoT Secure Tunneling demo	689
Open a tunnel and start SSH session to remote device	689
Local proxy	693
Local proxy security best practices	693
Configure local proxy for devices that use web proxy	694
Build the local proxy	694
Configure your web proxy	694
Configure and start the local proxy	695
Multiplex data streams in a secure tunnel	699
Example use case	699
How to set up a multiplexed tunnel	699
IoT agent snippet	701
Configuring a remote device	703

Device provisioning	704
Provisioning devices in AWS IoT	704
Fleet provisioning APIs	705
Provisioning devices that don't have device certificates using fleet provisioning	706
Provisioning by claim	706
Provisioning by trusted user	708
Using pre-provisioning hooks with the AWS CLI	709
Provisioning devices that have device certificates	711
Single thing provisioning	712
Just-in-time provisioning	712
Bulk registration	715
Provisioning templates	716
Parameters section	716
Resources section	716
Template example for JITP and bulk registration	720
Fleet provisioning	721
Pre-provisioning hooks	724
Pre-provision hook input	724
Pre-provision hook return value	724
Pre-provisioning hook Lambda example	725
Device provisioning MQTT API	726
CreateCertificateFromCsr	727
CreateKeysAndCertificate	728
RegisterThing	730
Fleet indexing service	732
Managing thing indexing	732
Enabling thing indexing	732
Describing a thing index	738
Querying a thing index	739
Restrictions and limitations	740
Authorization	742
Managing thing group indexing	742
Enabling thing group indexing	742
Describing group indexes	743
Querying a thing group index	743
Authorization	743
Querying for aggregate data	744
GetStatistics	744
GetCardinality	746
GetPercentiles	747
GetBucketsAggregation	748
Authorization	749
Query syntax	749
Example thing queries	750
Example thing group queries	752
Fleet metrics	753
Getting started tutorial	753
Managing fleet metrics	758
MQTT-based file delivery	762
What is a stream?	762
Managing a stream in the AWS Cloud	763
Grant permissions to your devices	763
Connect your devices to AWS IoT	764
Using AWS IoT MQTT-based file delivery in devices	764
Use DescribeStream to get stream data	764
Get data blocks from a stream file	766
Handling errors from AWS IoT MQTT-based file delivery	769

An example use case in FreeRTOS OTA	771
AWS IoT Device Defender	772
AWS training and certification	772
Getting started with AWS IoT Device Defender	772
Setting up	772
Audit guide	774
ML Detect guide	786
Customize when and how you view AWS IoT Device Defender audit results	808
Audit	818
Issue severity	818
Next steps	819
Audit checks	819
Audit commands	844
Audit finding suppressions	870
Detect	879
Monitoring the behavior of unregistered devices	880
Security use cases	881
Concepts	885
Behaviors	886
ML Detect	888
Custom metrics	892
Device-side metrics	897
Cloud-side metrics	912
Scoping metrics in security profiles using dimensions	919
Permissions	925
Detect commands	927
How to use AWS IoT Device Defender detect	928
Mitigation actions	929
Audit mitigation actions	929
Detect mitigation actions	932
How to define and manage mitigation actions	932
Apply mitigation actions	936
Permissions	941
Mitigation action commands	944
Using AWS IoT Device Defender with other AWS services	945
Using AWS IoT Device Defender with devices running AWS IoT Greengrass	945
Using AWS IoT Device Defender with FreeRTOS and embedded devices	945
Security best practices for device agents	945
Device Advisor	948
Setting up	948
Create an IoT thing	948
Create an IAM role to be used as your device role	949
Create a custom-managed policy for an IAM user to use Device Advisor	950
Create an IAM user to use Device Advisor	950
Configure your device	951
Getting started with Device Advisor in the console	952
Device Advisor workflow	958
Prerequisites	958
Create a test suite definition	958
Get a test suite definition	960
Get a test endpoint	960
Start a test suite run	960
Get a test suite run	961
Stop a test suite run	961
Get a qualification report for a successful qualification test suite run	961
Device Advisor detailed console workflow	962
Prerequisites	962

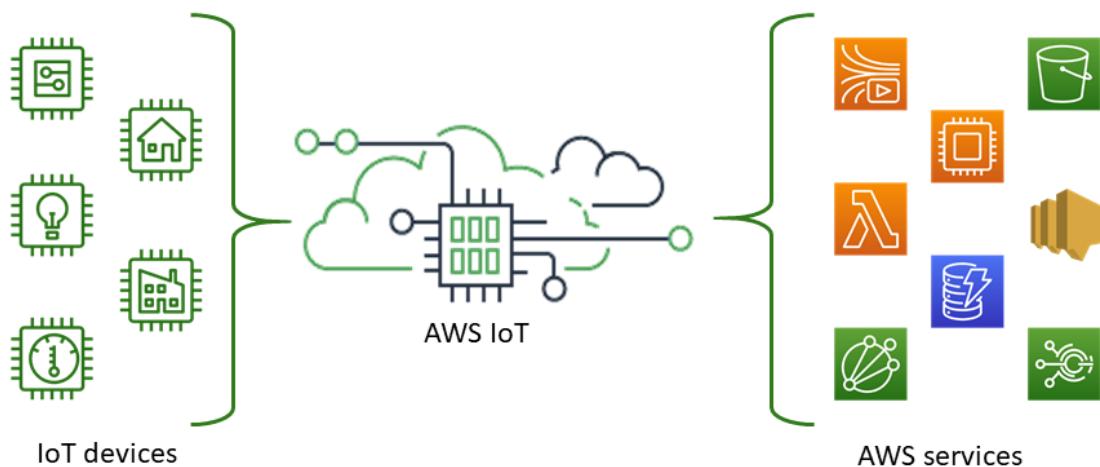
Create a test suite definition	962
Start a test suite run	966
Stop a test suite run (optional)	968
View test suite run details and logs	969
Download an AWS IoT qualification report	970
Device Advisor test cases	971
TLS	971
Permissions and policies	974
MQTT	974
Shadow	979
Job Execution	981
Event messages	982
How event messages are generated	982
Policy for receiving event messages	982
Enable events for AWS IoT	982
Registry events	983
Thing events	983
Thing type events	984
Thing group events	986
Jobs events	990
Lifecycle events	993
Connect/Disconnect events	993
Subscribe/Unsubscribe events	996
AWS IoT Core for LoRaWAN	998
Introduction	998
How to use AWS IoT Core for LoRaWAN	998
AWS IoT Core for LoRaWAN Regions and endpoints	999
AWS IoT Core for LoRaWAN pricing	999
What is AWS IoT Core for LoRaWAN?	999
What is LoRaWAN?	1000
How AWS IoT Core for LoRaWAN works	1000
Connecting gateways and devices to AWS IoT Core for LoRaWAN	1002
Naming conventions for your devices, gateways, profiles, and destinations	1002
Mapping of device data to service data	1002
Using the console to onboard your device and gateway to AWS IoT Core for LoRaWAN	1002
Describe your AWS IoT Core for LoRaWAN resources	1003
Onboard your gateways to AWS IoT Core for LoRaWAN	1004
Onboard your devices to AWS IoT Core for LoRaWAN	1010
Connecting to AWS IoT Core for LoRaWAN through a VPC interface endpoint	1021
Onboard AWS IoT Core for LoRaWAN control plane API endpoint	1022
Onboard AWS IoT Core for LoRaWAN data plane API endpoints	1025
Managing gateways with AWS IoT Core for LoRaWAN	1030
LoRa Basics Station software requirement	1031
Using qualified gateways from the AWS Partner Device Catalog	1031
Using CUPS and LNS protocols	1031
Configure your gateway's subbands and filtering capabilities	1031
Update gateway firmware using CUPS service with AWS IoT Core for LoRaWAN	1033
Managing devices with AWS IoT Core for LoRaWAN	1044
Device considerations	1044
Using devices with gateways qualified for AWS IoT Core for LoRaWAN	1044
LoRaWAN version	1044
Activation modes	1044
Device classes	1044
View format of uplink messages sent from LoRaWAN devices	1045
Create multicast groups to send a downlink payload to multiple devices	1047
Firmware Updates Over-The-Air (FUOTA) for AWS IoT Core for LoRaWAN devices	1057
Monitoring your wireless resource fleet in real time using network analyzer	1067

Add resources and update the network analyzer configuration	1068
Stream network analyzer trace messages with WebSockets	1071
View and monitor network analyzer trace message logs in real time	1075
Monitoring and logging for AWS IoT Core for LoRaWAN using Amazon CloudWatch	1078
Configure Logging for AWS IoT Core for LoRaWAN	1079
Monitor AWS IoT Core for LoRaWAN using CloudWatch Logs	1089
Data security with AWS IoT Core for LoRaWAN	1097
How data is secured throughout the system	1097
LoRaWAN device and gateway transport security	1098
Amazon Sidewalk Integration for AWS IoT Core	1099
How to onboard your Sidewalk device	1099
Getting started with Amazon Sidewalk Integration for AWS IoT Core	1099
Add your Sidewalk account credentials	1100
Add a destination for your Sidewalk device	1101
Create rules to process Sidewalk device messages	1103
Connect your Sidewalk device and view uplink metadata format	1104
Event notifications for Amazon Sidewalk Integration for AWS IoT Core	1105
How your resources can be notified of events	1105
Enable events for Sidewalk devices	1106
Pricing for Sidewalk events	1107
Event types for Sidewalk devices	1107
Device registration state events	1107
Proximity events	1109
Alexa Voice Service (AVS) Integration for AWS IoT	1112
Getting started with Alexa Voice Service (AVS) Integration for AWS IoT on an NXP device	1113
Preview Alexa Voice Service (AVS) Integration for AWS IoT with a preconfigured NXP account ..	1114
Interact with Alexa	1124
Use your AWS and Alexa Voice Service developer accounts to set up AVS for AWS IoT	1126
AWS IoT Device SDKs, Mobile SDKs, and AWS IoT Device Client	1127
AWS IoT Device SDKs	1127
AWS IoT Device SDK for Embedded C	1128
Earlier AWS IoT Device SDKs versions	1129
AWS Mobile SDKs	1129
AWS IoT Device Client	1130
Troubleshooting	1131
Diagnosing connectivity issues	1131
Connection	1131
Authentication	1131
Authorization	1132
Security and identity	1133
Diagnosing rules issues	1133
Configuring CloudWatch Logs for troubleshooting	1133
Diagnosing external services	1134
Diagnosing SQL problems	1134
Diagnosing problems with shadows	1135
Diagnosing Salesforce action issues	1136
Execution trace	1136
Action success and failure	1136
Fleet indexing troubleshooting guide	1137
Troubleshooting aggregation queries for the fleet indexing service	1137
Troubleshooting fleet metrics	1137
Troubleshooting "Stream limit exceeded for your AWS account"	1138
AWS IoT Device Defender troubleshooting guide	1138
Device Advisor troubleshooting guide	1141
Troubleshooting device fleet disconnects	1142
AWS IoT errors	1143
AWS IoT quotas	1144

What is AWS IoT?

AWS IoT provides the cloud services that connect your IoT devices to other devices and AWS cloud services. AWS IoT provides device software that can help you integrate your IoT devices into AWS IoT-based solutions. If your devices can connect to AWS IoT, AWS IoT can connect them to the cloud services that AWS provides.

For a hands-on introduction to AWS IoT, visit [Getting started with AWS IoT Core \(p. 16\)](#).



AWS IoT lets you select the most appropriate and up-to-date technologies for your solution. To help you manage and support your IoT devices in the field, AWS IoT Core supports these protocols:

- [MQTT \(Message Queuing and Telemetry Transport\) \(p. 80\)](#)
- [MQTT over WSS \(Websockets Secure\) \(p. 80\)](#)
- [HTTPS \(Hypertext Transfer Protocol - Secure\) \(p. 90\)](#)
- [LoRaWAN \(Long Range Wide Area Network\) \(p. 998\)](#)

The AWS IoT Core message broker supports devices and clients that use MQTT and MQTT over WSS protocols to publish and subscribe to messages. It also supports devices and clients that use the HTTPS protocol to publish messages.

AWS IoT Core for LoRaWAN helps you connect and manage wireless LoRaWAN (low-power long-range Wide Area Network) devices. AWS IoT Core for LoRaWAN replaces the need for you to develop and operate a LoRaWAN Network Server (LNS).

If you don't require AWS IoT features such as device communications, [rules \(p. 443\)](#), or [jobs \(p. 637\)](#), see [AWS Messaging](#) for information about other AWS IoT messaging services that might better fit your requirements.

How your devices and apps access AWS IoT

AWS IoT provides the following interfaces for [AWS IoT tutorials \(p. 116\)](#):

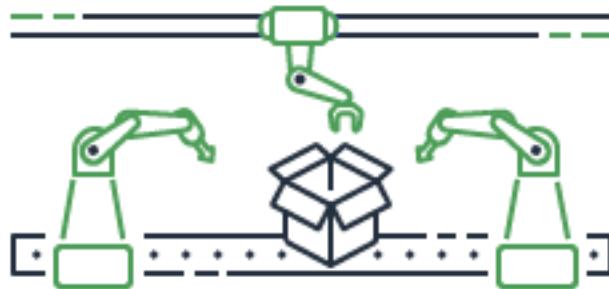
- **AWS IoT Device SDKs**—Build applications on your devices that send messages to and receive messages from AWS IoT. For more information, see [AWS IoT Device SDKs, Mobile SDKs, and AWS IoT Device Client \(p. 1127\)](#).
- **AWS IoT Core for LoRaWAN**—Connect and manage your long range WAN (LoRaWAN) devices and gateways by using [AWS IoT Core for LoRaWAN \(p. 998\)](#).
- **AWS Command Line Interface (AWS CLI)**—Run commands for AWS IoT on Windows, macOS, and Linux. These commands allow you to create and manage thing objects, certificates, rules, jobs, and policies. To get started, see the [AWS Command Line Interface User Guide](#). For more information about the commands for AWS IoT, see `iot` in the [AWS CLI Command Reference](#).
- **AWS IoT API**—Build your IoT applications using HTTP or HTTPS requests. These API actions allow you to programmatically create and manage thing objects, certificates, rules, and policies. For more information about the API actions for AWS IoT, see [Actions](#) in the [AWS IoT API Reference](#).
- **AWS SDKs**—Build your IoT applications using language-specific APIs. These SDKs wrap the HTTP/HTTPS API and allow you to program in any of the supported languages. For more information, see [AWS SDKs and Tools](#).

You can also access AWS IoT through the [AWS IoT console](#), which provides a graphical user interface (GUI) through which you can configure and manage the thing objects, certificates, rules, jobs, policies, and other elements of your IoT solutions.

What AWS IoT can do

This topic describes some of the solutions that you might need that AWS IoT supports.

IoT in Industry



These are some examples of AWS IoT solutions for [industrial use cases](#) that apply IoT technologies to improve the performance and productivity of industrial processes.

Solutions for industrial use cases

- [Use AWS IoT to build predictive quality models in industrial operations](#)

See how AWS IoT can collect and analyze data from industrial operations to build predictive quality models. [Learn more](#)

- [Use AWS IoT to support predictive maintenance in industrial operations](#)

See how AWS IoT can help plan preventive maintenance to reduce unplanned downtime. [Learn more](#)

IoT in Home automation



These are some examples of AWS IoT solutions for [home automation use cases](#) that apply IoT technologies to build scalable IoT applications that automate household activities using connected home devices.

Solutions for home automation

- [Use AWS IoT in your connected home](#)

See how AWS IoT can provide integrated home automation solutions.

- [Use AWS IoT to provide home security and monitoring](#)

See how AWS IoT can apply machine learning and edge computing to your home automation solution.

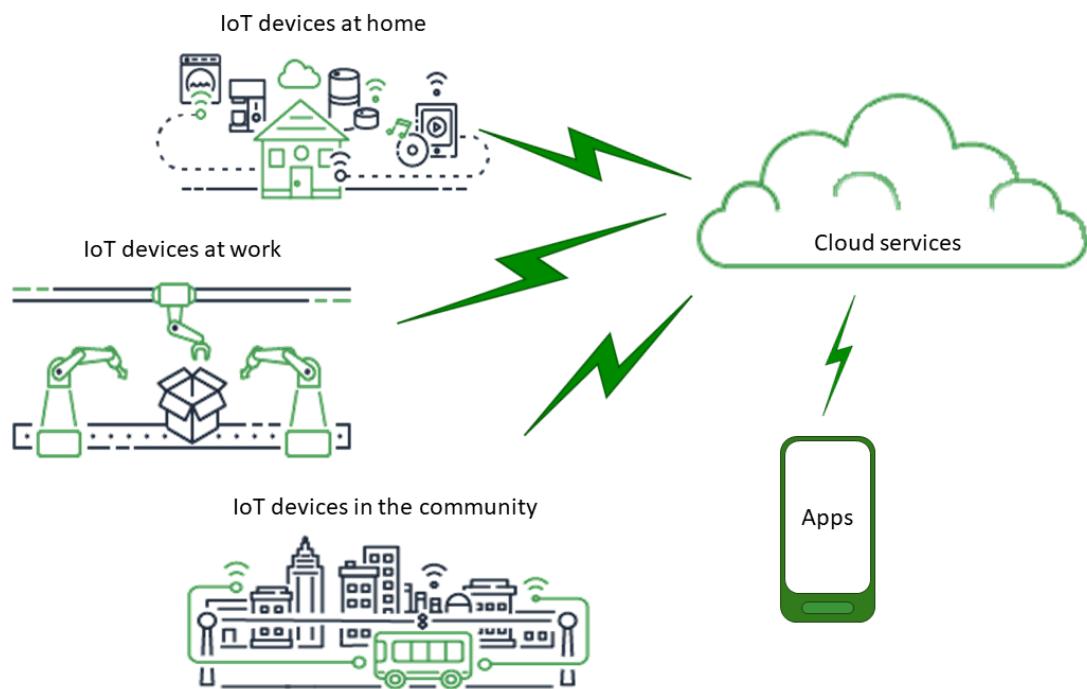
For a list of solutions for industrial, consumer, and commercial use cases, see the [AWS IoT Solution Repository](#).

How AWS IoT works

AWS IoT provides cloud services and device support that you can use to implement IoT solutions. AWS provides many cloud services to support IoT-based applications. So to help you understand where to start, this section provides a diagram and definition of essential concepts to introduce you to the IoT universe.

The IoT universe

In general, the Internet of Things (IoT) consists of the key components shown in this diagram.



Apps

Apps give end users access to IoT devices and the features provided by the cloud services to which those devices are connected.

Cloud services

Cloud services are distributed, large-scale data storage and processing services that are connected to the internet. Examples include:

- IoT connection and management services.
AWS IoT is an example of an IoT connection and management service.
- Compute services, such as Amazon Elastic Compute Cloud and AWS Lambda.
- Database services, such as Amazon DynamoDB

Communications

Devices communicate with cloud services by using various technologies and protocols. Examples include:

- Wi-Fi/Broadband internet
- Broadband cellular data
- Narrow-band cellular data
- Long-range Wide Area Network (LoRaWAN)
- Proprietary RF communications

Devices

A device is a type of hardware that manages interfaces and communications. Devices are usually located in close proximity to the real-world interfaces they monitor and control. Devices can include computing and storage resources, such as microcontrollers, CPU, memory. Examples include:

- Raspberry Pi
- Arduino
- Voice-interface assistants
- LoRaWAN and devices
- Amazon Sidewalk devices
- Custom IoT devices

Interfaces

An interface is a component that connects a device to the physical world.

- User interfaces

Components that allow devices and users to communicate with each other.

- Input interfaces

Enable a user to communicate with a device

Examples: keypad, button

- Output interfaces

Enable a device to communicate with a user

Examples: Alpha-numeric display, graphical display, indicator light, alarm bell

- Sensors

Input components that measure or sense something in the outside world in a way that a device understands. Examples include:

- Temperature sensor (converts temperature to an analog or digital signal)
- Humidity sensor (converts relative humidity to an analog or digital signal)
- Analog to digital convertor (converts an analog voltage to a numeric value)
- Ultrasonic distance measuring unit (converts a distance to a numeric value)
- Optical sensor (converts a light level to a numeric value)
- Camera (converts image data to digital data)

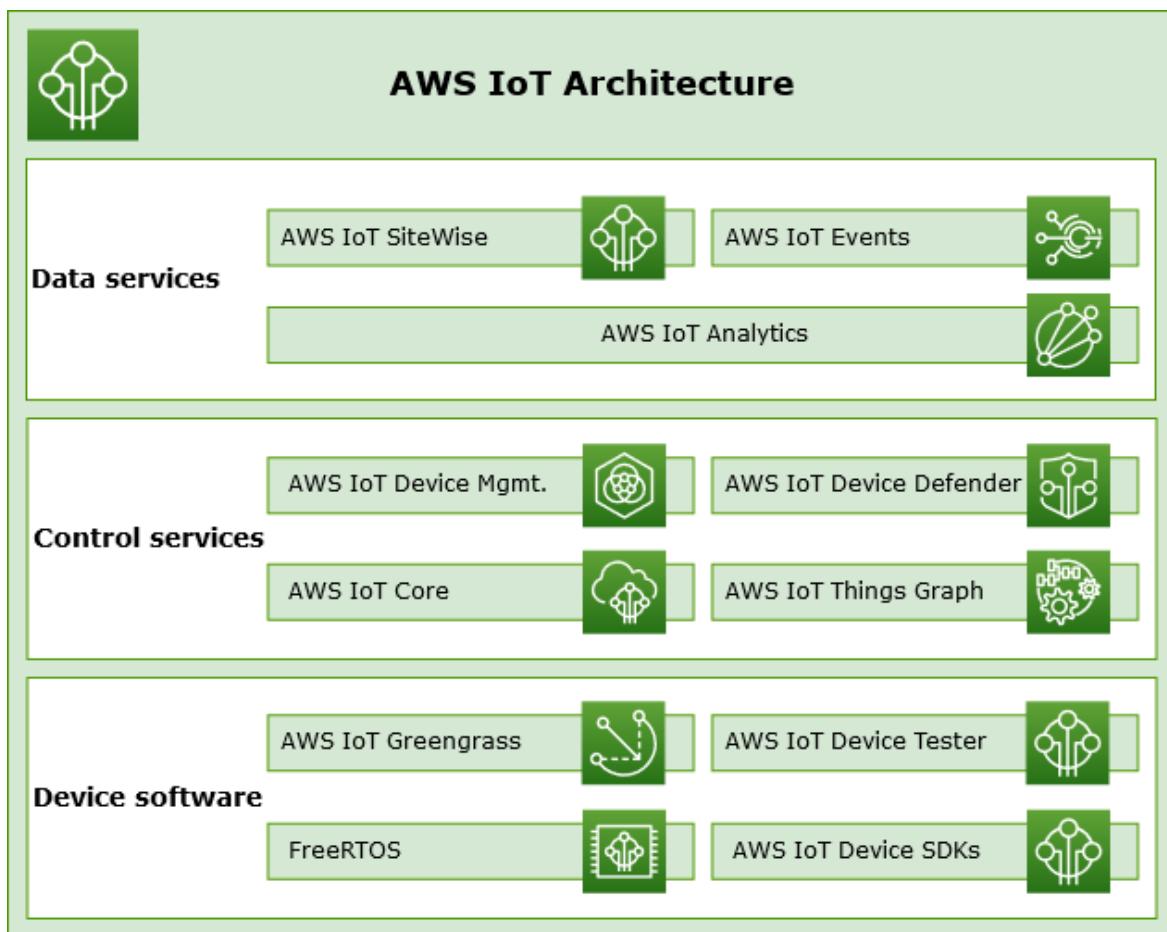
- Actuators

Output components that the device can use to control something in the outside world. Examples include:

- Stepper motors (convert electric signals to movement)
- Relays (control high electric voltages and currents)

AWS IoT services overview

In the IoT universe, AWS IoT provides the services that support the devices that interact with the world and the data that passes between them and AWS IoT. AWS IoT is made up of the services that are shown in this illustration to support your IoT solution.



AWS IoT device software

AWS IoT provides this software to support your IoT devices.

AWS IoT Greengrass

AWS IoT Greengrass extends AWS IoT to edge devices so they can act locally on the data they generate and use the cloud for management, analytics, and durable storage. With AWS IoT Greengrass, connected devices can run AWS Lambda functions, Docker containers, or both, execute predictions based on machine learning models, keep device data in sync, and communicate with other devices securely – even when they are not connected to the Internet.

AWS IoT Device Tester

AWS IoT Device Tester for FreeRTOS and AWS IoT Greengrass is a test automation tool for microcontrollers. AWS IoT Device Tester, test your device to determine if it will run FreeRTOS or AWS IoT Greengrass and interoperate with AWS IoT services.

AWS IoT Device SDKs

The AWS IoT Device and Mobile SDKs (p. 1127) help you efficiently connect your devices to AWS IoT. The AWS IoT Device and Mobile SDKs include open-source libraries, developer guides with samples, and porting guides so that you can build innovative IoT products or solutions on your choice of hardware platforms.

FreeRTOS

[FreeRTOS](#) is an open source, real-time operating system for microcontrollers that lets you include small, low-power edge devices in your IoT solution. FreeRTOS includes a kernel and a growing set of software libraries that support many applications. FreeRTOS systems can securely connect your small, low-power devices to [AWS IoT](#) and support more powerful edge devices running [AWS IoT Greengrass](#).

AWS IoT control services

Connect to the following AWS IoT services to manage the devices in your IoT solution.

AWS IoT Core

[AWS IoT Core](#) is a managed cloud service that enables connected devices to securely interact with cloud applications and other devices. AWS IoT Core can support many devices and messages, and it can process and route those messages to AWS IoT endpoints and other devices. With AWS IoT Core, your applications can interact with all of your devices even when they aren't connected.

AWS IoT Device Management

[AWS IoT Device Management](#) services help you track, monitor, and manage the plethora of connected devices that make up your devices fleets. AWS IoT Device Management services help you ensure that your IoT devices work properly and securely after they have been deployed. They also provide secure tunneling to access your devices, monitor their health, detect and remotely troubleshoot problems, as well as services to manage device software and firmware updates.

AWS IoT Device Defender

[AWS IoT Device Defender](#) helps you secure your fleet of IoT devices. AWS IoT Device Defender continuously audits your IoT configurations to make sure that they aren't deviating from security best practices. AWS IoT Device Defender sends an alert when it detects any gaps in your IoT configuration that might create a security risk, such as identity certificates being shared across multiple devices or a device with a revoked identity certificate trying to connect to [AWS IoT Core](#).

AWS IoT Things Graph

[AWS IoT Things Graph](#) is a service that lets you visually connect different devices and web services to build IoT applications. AWS IoT Things Graph provides a visual drag-and-drop interface for connecting and coordinating interactions between devices and web services, so that you can build IoT applications efficiently.

AWS IoT data services

Analyze the data from the devices in your IoT solution and take appropriate action by using the following AWS IoT services.

AWS IoT Analytics

[AWS IoT Analytics](#) lets you efficiently run and operationalize sophisticated analytics on massive volumes of unstructured IoT data. AWS IoT Analytics automates each difficult step that is required to analyze data from IoT devices. AWS IoT Analytics filters, transforms, and enriches IoT data before storing it in a time-series data store for analysis. You can analyze your data by running one-time or scheduled queries using the built-in SQL query engine or machine learning.

AWS IoT SiteWise

[AWS IoT SiteWise](#) collects, stores, organizes, and monitors data passed from industrial equipment by MQTT messages or APIs at scale by providing software that runs on a gateway in your facilities. The

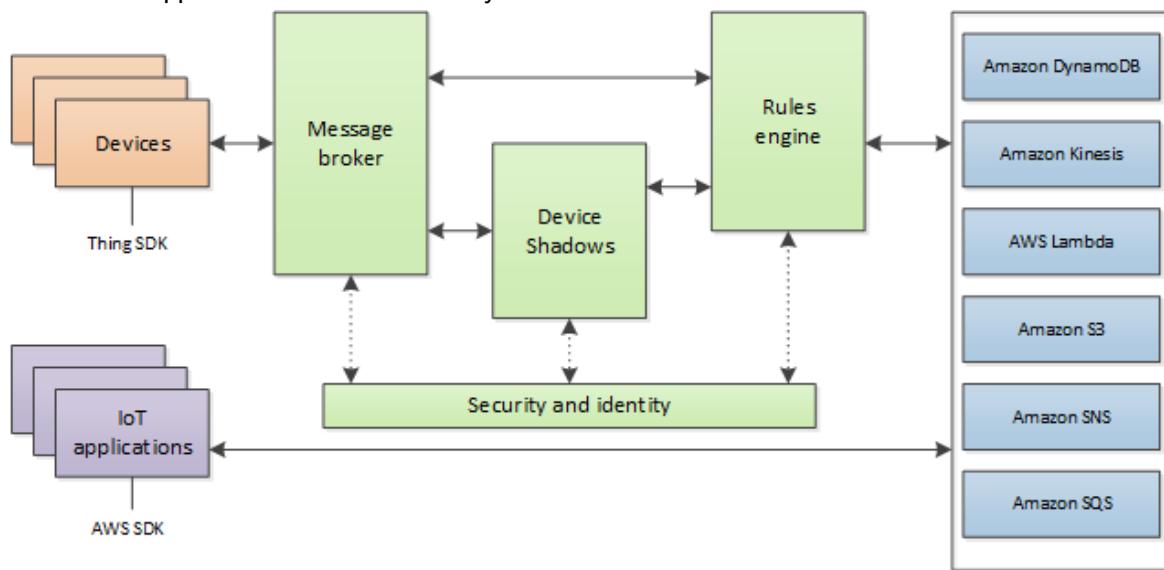
gateway securely connects to your on-premises data servers and automates the process of collecting and organizing the data and sending it to the AWS Cloud.

AWS IoT Events

[AWS IoT Events](#) detects and responds to events from IoT sensors and applications. Events are patterns of data that identify more complicated circumstances than expected, such as motion detectors using movement signals to activate lights and security cameras. AWS IoT Events continuously monitors data from multiple IoT sensors and applications, and integrates with other services, such as AWS IoT Core, IoT SiteWise, DynamoDB, and others to enable early detection and unique insights.

AWS IoT Core services

AWS IoT Core provides the services that connect your IoT devices to the AWS Cloud so that other cloud services and applications can interact with your internet-connected devices.



The next section describes each of the AWS IoT Core services shown in the illustration.

AWS IoT Core messaging services

The AWS IoT Core connectivity services provide secure communication with the IoT devices and manage the messages that pass between them and AWS IoT.

Device gateway

Enables devices to securely and efficiently communicate with AWS IoT. Device communication is secured by secure protocols that use of X.509 certificates.

Message broker

Provides a secure mechanism for devices and AWS IoT applications to publish and receive messages from each other. You can use either the MQTT protocol directly or MQTT over WebSocket to publish and subscribe. For more information about the protocols that AWS IoT supports, see [the section called "Device communication protocols" \(p. 77\)](#). Devices and clients can also use the HTTP REST interface to publish data to the message broker.

The message broker distributes device data to devices that have subscribed to it and to other AWS IoT Core services, such as the Device Shadow service and the rules engine.

AWS IoT Core for LoRaWAN

AWS IoT Core for LoRaWAN makes it possible to set up a private LoRaWAN network by connecting your LoRaWAN devices and gateways to AWS without the need to develop and operate a LoRaWAN Network Server (LNS). Messages received from LoRaWAN devices are sent to the rules engine where they can be formatted and sent to other AWS IoT services.

Rules engine

The Rules engine connects data from the message broker to other AWS IoT services for storage and additional processing. For example, you can insert, update, or query a DynamoDB table or invoke a Lambda function based on an expression that you defined in the Rules engine. You can use an SQL-based language to select data from message payloads, and then process and send the data to other services, such as Amazon Simple Storage Service (Amazon S3), Amazon DynamoDB, and AWS Lambda. You can also create rules that republish messages to the message broker and on to other subscribers. For more information, see [Rules for AWS IoT \(p. 443\)](#).

AWS IoT Core control services

The AWS IoT Core control services provide device security, management, and registration features.

Custom Authentication service

You can define custom authorizers that allow you to manage your own authentication and authorization strategy using a custom authentication service and a Lambda function. Custom authorizers allow AWS IoT to authenticate your devices and authorize operations using bearer token authentication and authorization strategies.

Custom authorizers can implement various authentication strategies; for example, JSON Web Token verification or OAuth provider callout. They must return policy documents that are used by the device gateway to authorize MQTT operations. For more information, see [Custom authentication \(p. 301\)](#).

Device Provisioning service

Allows you to provision devices using a template that describes the resources required for your device: a *thing object*, a certificate, and one or more policies. A thing object is an entry in the registry that contains attributes that describe a device. Devices use certificates to authenticate with AWS IoT. Policies determine which operations a device can perform in AWS IoT.

The templates contain variables that are replaced by values in a dictionary (map). You can use the same template to provision multiple devices just by passing in different values for the template variables in the dictionary. For more information, see [Device provisioning \(p. 704\)](#).

Group registry

Groups allow you to manage several devices at once by categorizing them into groups. Groups can also contain groups—you can build a hierarchy of groups. Any action that you perform on a parent group will apply to its child groups. The same action also applies to all the devices in the parent group and all devices in the child groups. Permissions granted to a group will apply to all devices in the group and in all of its child groups. For more information, see [Managing devices with AWS IoT \(p. 251\)](#).

Jobs service

Allows you to define a set of remote operations that are sent to and run on one or more devices connected to AWS IoT. For example, you can define a job that instructs a set of devices to download and install application or firmware updates, reboot, rotate certificates, or perform remote troubleshooting operations.

To create a job, you specify a description of the remote operations to be performed and a list of targets that should perform them. The targets can be individual devices, groups or both. For more information, see [Jobs \(p. 637\)](#).

Registry

Organizes the resources associated with each device in the AWS Cloud. You register your devices and associate up to three custom attributes with each one. You can also associate certificates and MQTT client IDs with each device to improve your ability to manage and troubleshoot them. For more information, see [Managing devices with AWS IoT \(p. 251\)](#).

Security and Identity service

Provides shared responsibility for security in the AWS Cloud. Your devices must keep their credentials safe to securely send data to the message broker. The message broker and rules engine use AWS security features to send data securely to devices or other AWS services. For more information, see [Authentication \(p. 279\)](#).

AWS IoT Core data services

The AWS IoT Core data services help your IoT solutions provide a reliable application experience even with devices that are not always connected.

Device shadow

A JSON document used to store and retrieve current state information for a device.

Device Shadow service

The Device Shadow service maintains a device's state so that applications can communicate with a device whether the device is online or not. When a device is offline, the Device Shadow service manages its data for connected applications. When the device reconnects, it synchronizes its state with that of its shadow in the Device Shadow service. Your devices can also publish their current state to a shadow for use by applications or other devices that might not be connected all the time. For more information, see [AWS IoT Device Shadow service \(p. 591\)](#).

AWS IoT Core support service

Alexa Voice Service (AVS) Integration for AWS IoT

Brings Alexa Voice to any connected device. AVS for AWS IoT reduces the cost and complexity of integrating Alexa. This feature uses AWS IoT to offload intensive computational and memory audio tasks from the device to the cloud. Because of the resulting reduction in the engineering bill of materials (EBOM) cost, device makers can cost-effectively bring Alexa to resource-constrained IoT devices, and enable consumers to talk directly to Alexa in parts of their home, office, or hotel rooms for an ambient experience.

AVS for AWS IoT enables Alexa built-in functionality on MCUs, such as the ARM Cortex M class with less than 1 MB embedded RAM. To do so, AVS offloads memory and compute tasks to a virtual Alexa Built-in device in the cloud. This reduces EBOM cost by up to 50 percent. For more information, see [Alexa Voice Service \(AVS\) Integration for AWS IoT \(p. 1112\)](#).

Amazon Sidewalk Integration for AWS IoT Core

[Amazon Sidewalk](#) is a shared network that improves connectivity options to help devices work together better. Amazon Sidewalk supports a wide range of customer devices such as those that locate pets or valuables, those that provide smart home security and lighting control, and those that provide remote diagnostics for appliances and tools. Amazon Sidewalk Integration for AWS IoT Core makes it possible for device manufacturers to add their Sidewalk device fleet to the AWS IoT Cloud.

For more information, see [Amazon Sidewalk Integration for AWS IoT Core \(p. 1099\)](#)

Learn more about AWS IoT

This topic helps you get familiar with the world of AWS IoT. You can get general information about how IoT solutions are applied in various use cases, training resources, links to social media for AWS IoT and all other AWS services, and a list of services and communication protocols that AWS IoT uses.

Training resources for AWS IoT

We provide these training courses to help you learn about AWS IoT and how to apply them to your solution design.

- [Introduction to AWS IoT](#)

A video overview of AWS IoT and its core services.

- [Deep Dive into AWS IoT Authentication and Authorization](#)

An advanced course that explores the concepts of AWS IoT authentication and authorization. You will learn how to authenticate and authorize clients to access the AWS IoT control plane and data plane APIs.

- [Internet of Things Foundation Series](#)

A learning path of IoT eLearning modules on different IoT technologies and features.

AWS IoT resources and guides

These are in-depth technical resources on specific aspects of AWS IoT.

- [IoT Lens – AWS IoT Well-Architected Framework](#)

A document that describes the best practices for architecting your IoT applications on AWS.

- [Designing MQTT Topics for AWS IoT Core](#)

A PDF document that describes the best practices for designing MQTT topics in AWS IoT Core and leveraging AWS IoT Core features with MQTT.

- [Device Manufacturing and Provisioning with X.509 Certificates in AWS IoT Core](#)

A PDF document that describes the different ways that AWS IoT provides to provision large fleets of devices.

- [AWS IoT Resources](#)

IoT-specific resources, such as Technical Guides, Reference Architectures, eBooks, and curated blog posts, presented in a searchable index.

- [IoT Atlas](#)

Overviews on how to solve common IoT design problems. The *IoT Atlas* provides in-depth looks into the design challenges that you are likely to encounter while developing your IoT solution.

- [AWS Whitepapers & Guides](#)

Our current collection of whitepapers and guides on AWS IoT and other AWS technologies.

AWS IoT in social media

These social media channels provide information about AWS IoT and AWS-related topics.

- [The Internet of Things on AWS IoT – Official Blog](#)
- [AWS IoT videos in the Amazon Web Services channel on YouTube](#)

These social media accounts cover all AWS services, including AWS IoT

- [The Amazon Web Services channel on YouTube](#)
- [Amazon Web Services on Twitter](#)
- [Amazon Web Services on Facebook](#)
- [Amazon Web Services on Instagram](#)
- [Amazon Web Services on LinkedIn](#)

AWS services used by the AWS IoT Core rules engine

The AWS IoT Core rules engine can connect to these AWS services.

- **[Amazon DynamoDB](#)**

Amazon DynamoDB is a scalable, NoSQL database service that provides fast and predictable database performance.

- **[Amazon Kinesis](#)**

Amazon Kinesis makes it easy to collect, process, and analyze real-time, streaming data so you can get timely insights and react quickly to new information. Amazon Kinesis can ingest real-time data such as video, audio, application logs, website clickstreams, and IoT telemetry data for machine learning, analytics, and other applications.

- **[AWS Lambda](#)**

AWS Lambda lets you run code without provisioning or managing servers. You can set up your code to automatically trigger from AWS IoT data and events or call it directly from a web or mobile app.

- **[Amazon Simple Storage Service](#)**

Amazon Simple Storage Service (Amazon S3) can store and retrieve any amount of data at any time, from anywhere on the web. AWS IoT rules can send data to Amazon S3 for storage.

- **[Amazon Simple Notification Service](#)**

Amazon Simple Notification Service (Amazon SNS) is a web service that enables applications, end users, and devices to send and receive notifications from the cloud.

- **[Amazon Simple Queue Service](#)**

Amazon Simple Queue Service (Amazon SQS) is a message queuing service that decouples and scales microservices, distributed systems, and serverless applications.

- **[Amazon OpenSearch Service](#)**

Amazon OpenSearch Service (OpenSearch Service) is a managed service that makes it easy to deploy, operate, and scale OpenSearch, a popular open-source search and analytics engine.

- **[Amazon Machine Learning](#)**

Amazon Machine Learning can create machine learning (ML) models by finding patterns in your IoT data. The service uses these models to process new data and generate predictions for your application.

- **[Amazon CloudWatch](#)**

Amazon CloudWatch provides a reliable, scalable, and flexible monitoring solution to help set up, manage, and scale your own monitoring systems and infrastructure.

Communication protocols supported by AWS IoT Core

These topics provide more information about the communication protocols used by AWS IoT. For more information about the protocols used by AWS IoT and connecting devices and services to AWS IoT, see [Connecting to AWS IoT Core \(p. 66\)](#).

- [MQTT \(Message Queuing Telemetry Transport\)](#)

The home page of the MQTT.org site where you can find the MQTT protocol specifications. For more information about how AWS IoT supports MQTT, see [MQTT \(p. 80\)](#).

- [HTTPS \(Hypertext Transfer Protocol - Secure\)](#)

Devices and apps can access AWS IoT services by using HTTPS.

- [LoRaWAN \(Long Range Wide Area Network\)](#)

LoRaWAN devices and gateways can connect to AWS IoT Core by using AWS IoT Core for LoRaWAN.

- [TLS \(Transport Layer Security\) v1.2](#)

The specification of the TLS v1.2 (RFC 5246). AWS IoT uses TLS v1.2 to establish secure connections between devices and AWS IoT.

What's new in the AWS IoT console

We're in the process of updating the user interface of the AWS IoT console to a new experience. We're updating the user interface in stages, so some pages in the console will have a new experience, some might have both the original and the new experience, and some might have only the original experience.

This table displays the current state of individual areas of the AWS IoT console user interface.

AWS IoT console user interface status

Console page	Original experience	New experience	Comments
Monitor	Not available	Available	
Activity	Not available	Available	
Onboard - Get started	Available	Not available yet	
Onboard - Fleet provisioning templates	Available	Not available yet	
Manage - Things	Available	Available	
Manage - Types	Available	Available	
Manage - Thing groups	Available	Available	
Manage - Billing groups	Available	Available	
Manage - Jobs	Available	Available	
Manage - Job templates	Available	Available	The save a job as a job template and create a job with a job

Console page	Original experience	New experience	Comments
			template features are not available in the original experience.
Manage - Tunnels	Not available	Available	
Fleet Hub - Get started	Not available	Available	Not available in all AWS Regions
Fleet Hub - Applications	Not available	Available	Not available in all AWS Regions
Greengrass - Getting started	Not available	Available	Not available in all AWS Regions
Greengrass - Core devices	Not available	Available	Not available in all AWS Regions
Greengrass - Components	Not available	Available	Not available in all AWS Regions
Greengrass - Deployments	Not available	Available	Not available in all AWS Regions
Greengrass - Classic (V1)	Available	Not available	Not available in all AWS Regions
Wireless connectivity - Intro	Not available	Available	Not available in all AWS Regions
Wireless connectivity - Gateways	Not available	Available	Not available in all AWS Regions
Wireless connectivity - Devices	Not available	Available	Not available in all AWS Regions
Wireless connectivity - Profiles	Not available	Available	Not available in all AWS Regions
Wireless connectivity - Destinations	Not available	Available	Not available in all AWS Regions
Secure - Certificates	Available	Not available yet	
Secure - Policies	Available	Not available yet	
Secure - CAs	Available	Not available yet	
Secure - Role Aliases	Available	Not available yet	
Secure - Authorizers	Available	Not available yet	
Defend - Intro	Available	Not available yet	
Defend - Audit	Available	Not available yet	
Defend - Detect	Available	Not available yet	
Defend - Mitigation actions	Available	Not available yet	

Console page	Original experience	New experience	Comments
Defend - Settings	Available	Not available yet	Not available in all AWS Regions
Act - Rules	Available	Not available yet	
Act - Destinations	Available	Not available yet	
Test - Device Advisor	Available	Not available yet	Not available in all AWS Regions
Test - MQTT test client	Available	Available	
Software	Available	Not available yet	
Settings	Not available	Available	
Learn	Available	Not available yet	

Legend

Status values

- **Available**

This user interface experience can be used.

- **Not available**

This user interface experience can't be used.

- **Not available yet**

The new user interface experience is being worked on, but it's not ready, yet.

- **In progress**

The new user interface experience is in the process of being updated. Some pages might still have the original user experience, however.

Getting started with AWS IoT Core

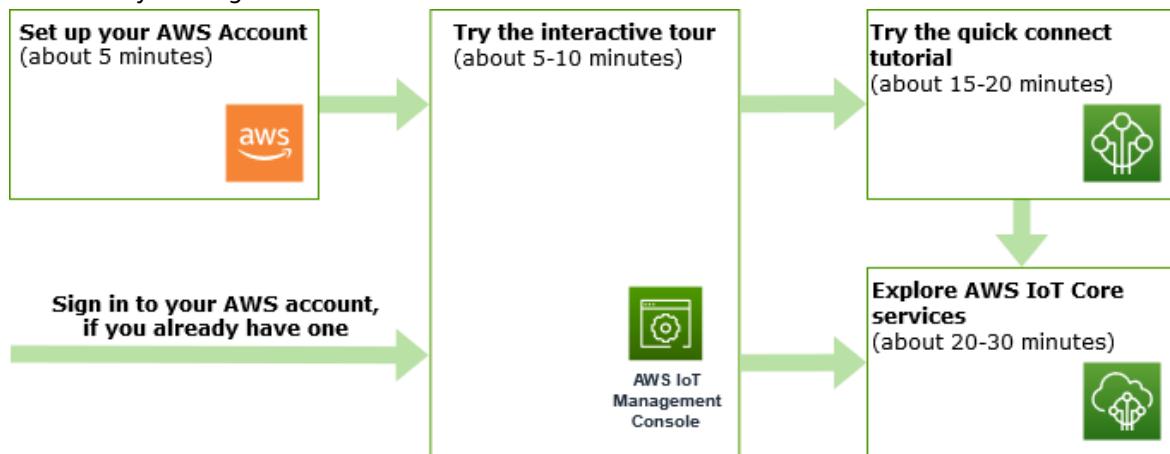
Whether you're new to IoT or you have years of experience, these resources present the AWS IoT concepts and terms that will help you start using AWS IoT.

- **Look** inside AWS IoT and its components in [How AWS IoT works \(p. 3\)](#).
- **Learn** more about AWS IoT ([p. 11](#)) from our collection of training materials and videos. This topic also includes a list of services that AWS IoT can connect to, social media links, and links to communication protocol specifications.
- **the section called “Connect your first device to AWS IoT Core” ([p. 16](#))**.
- **Develop** your IoT solutions by [Connecting to AWS IoT Core \(p. 66\)](#) and exploring the [AWS IoT tutorials \(p. 116\)](#).
- **Test and validate** your IoT devices for secure and reliable communication by using the [Device Advisor \(p. 948\)](#).
- **Manage** your solution by using AWS IoT Core management services such as [Fleet indexing service \(p. 732\)](#), [Jobs \(p. 637\)](#), and [AWS IoT Device Defender \(p. 772\)](#).
- **Analyze** the data from your devices by using the [AWS IoT data services \(p. 7\)](#).

Connect your first device to AWS IoT Core

AWS IoT Core services connect IoT devices to AWS IoT services and other AWS services. AWS IoT Core includes the device gateway and the message broker, which connect and process messages between your IoT devices and the cloud.

Here's how you can get started with AWS IoT Core and AWS IoT.



This section presents a tour of the AWS IoT Core to introduce its key services and provides several examples of how to connect a device to AWS IoT Core and pass messages between them. Passing messages between devices and the cloud is fundamental to every IoT solution and is how your devices can interact with other AWS services.

- [Set up your AWS account \(p. 17\)](#)

Before you can use AWS IoT services, you must set up an AWS account. If you already have an AWS account and an IAM user for yourself, you can use them and skip this step.

- [Try the interactive demo \(p. 19\)](#)

This demo is best if you want to see what a basic AWS IoT solution can do without connecting a device or downloading any software. The interactive demo presents a simulated solution built on AWS IoT Core services that illustrates how they interact.

- [Try the quick connect tutorial \(p. 23\)](#)

This tutorial is best if you want to quickly get started with AWS IoT and see how it works in a limited scenario. In this tutorial, you'll need a device and you'll install some AWS IoT software on it. If you don't have an IoT device, you can use your Windows, Linux, or macOS personal computer as a device for this tutorial. If you want to try AWS IoT, but you don't have a device, try the next option.

- [Explore AWS IoT Core services with a hands-on tutorial \(p. 33\)](#)

This tutorial is best for developers who want to get started with AWS IoT so they can continue to explore other AWS IoT Core features such as the rules engine and shadows. This tutorial follows a process similar to the quick connect tutorial, but provides more details on each step to enable a smoother transition to the more advanced tutorials.

- [View MQTT messages with the AWS IoT MQTT client \(p. 62\)](#)

Learn how to use the MQTT test client to watch your first device publish MQTT messages to AWS IoT. The MQTT test client is a useful tool to monitor and troubleshoot device connections.

Note

If you want to try more than one of these getting started tutorials or repeat the same tutorial, you should delete the thing object that you created from an earlier tutorial before you start another one. If you don't delete the thing object from an earlier tutorial, you will need to use a different thing name for subsequent tutorials. This is because the thing name must be unique in your account and AWS Region.

For more information about AWS IoT Core, see [What Is AWS IoT Core \(p. 1\)?](#)

Set up your AWS account

Before you use AWS IoT Core for the first time, complete the following tasks:

- [Sign up for an AWS account \(p. 17\)](#)
- [Create a user and grant permissions \(p. 18\)](#)
- [Open the AWS IoT console \(p. 19\)](#)

If you already have an AWS account and an IAM user for yourself, you can use them and skip ahead to the section called "[Open the AWS IoT console](#)" (p. 19).

Sign up for an AWS account

When you sign up for AWS, your account is automatically signed up for all services in AWS. If you have an AWS account already, skip this procedure. If you don't have an AWS account, use the following procedure to create one.

You can expect to spend about 5 minutes setting up your AWS account.

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open <https://portal.aws.amazon.com/billing/signup>.
2. Follow the online instructions.

Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

Note

Save your AWS account number, because you need it for the next task.

Create a user and grant permissions

This procedure describes how to create an IAM user for yourself and add that user to a group that has administrative permissions from an attached managed policy. IAM is the AWS service that manages users of and access to AWS resources. You must do this so that you can create the AWS IoT resources in your account and grant them permission to do what they need to do.

To create an administrator user for yourself and add the user to an administrators group (console)

1. Sign in to the [IAM console](#) as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

Note

We strongly recommend that you adhere to the best practice of using the **Administrator** IAM user that follows and securely lock away the root user credentials. Sign in as the root user only to perform a few [account and service management tasks](#).

2. In the navigation pane, choose **Users** and then choose **Add user**.
3. For **User name**, enter **Administrator**.
4. Select the check box next to **AWS Management Console access**. Then select **Custom password**, and then enter your new password in the text box.
5. (Optional) By default, AWS requires the new user to create a new password when first signing in. You can clear the check box next to **User must create a new password at next sign-in** to allow the new user to reset their password after they sign in.
6. Choose **Next: Permissions**.
7. Under **Set permissions**, choose **Add user to group**.
8. Choose **Create group**.
9. In the **Create group** dialog box, for **Group name** enter **Administrators**.
10. Choose **Filter policies**, and then select **AWS managed - job function** to filter the table contents.
11. In the policy list, select the check box for **AdministratorAccess**. Then choose **Create group**.

Note

You must activate IAM user and role access to Billing before you can use the **AdministratorAccess** permissions to access the AWS Billing and Cost Management console. To do this, follow the instructions in [step 1 of the tutorial about delegating access to the billing console](#).

12. Back in the list of groups, select the check box for your new group. Choose **Refresh** if necessary to see the group in the list.
13. Choose **Next: Tags**.

14. (Optional) Add metadata to the user by attaching tags as key-value pairs. For more information about using tags in IAM, see [Tagging IAM entities](#) in the *IAM User Guide*.
15. Choose **Next: Review** to see the list of group memberships to be added to the new user. When you are ready to proceed, choose **Create user**.

You can use this same process to create more groups and users and to give your users access to your AWS account resources. To learn about using policies that restrict user permissions to specific AWS resources, see [Access management](#) and [Example policies](#).

Open the AWS IoT console

Most of the console-oriented topics in this section start from the [AWS IoT console](#). If you aren't already signed in to your AWS account, sign in, then open the [AWS IoT console](#) and continue to the next section to continue getting started with AWS IoT.

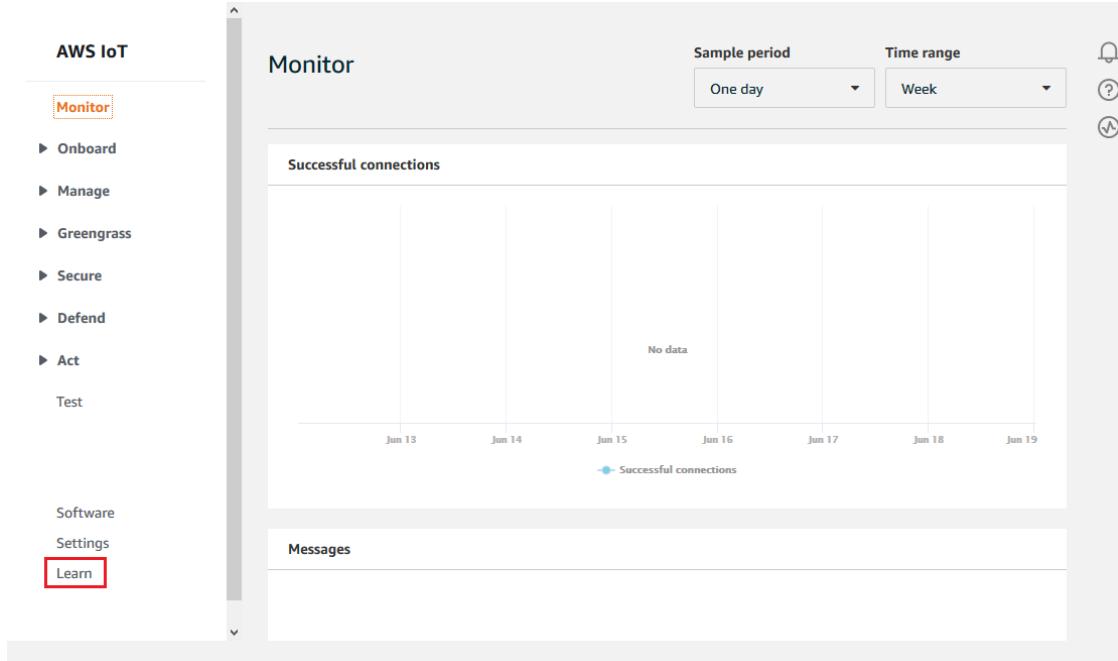
Try the AWS IoT Core interactive demo

The interactive demo shows the components of a simple IoT solution built on AWS IoT and how they interact with AWS IoT Core services. It does not create any AWS IoT resources nor does it require that you download any software or do any coding.

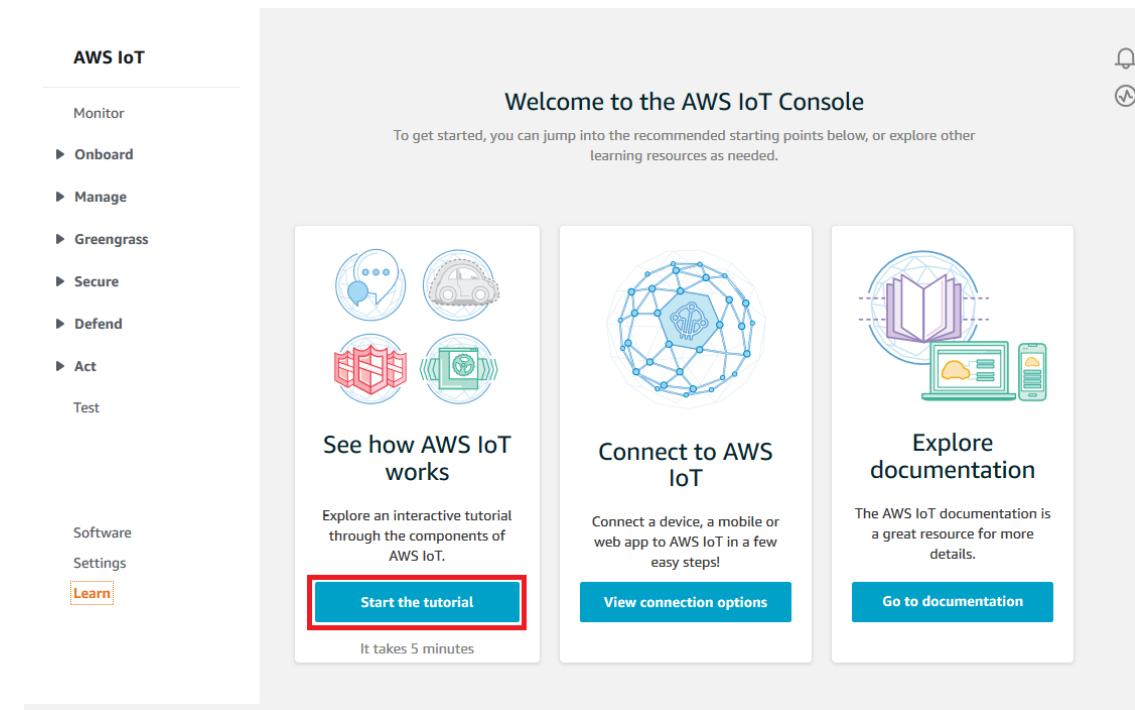
You can expect to spend about 10 minutes with this demo. The tile in the console says that it takes 5 minutes, and you can probably do it in 5 minutes, but 10 minutes will give you time to explore each of the different steps more thoroughly.

To run the AWS IoT Core interactive demo

1. Open your [AWS IoT console](#) and from the left menu, choose **Learn**.



2. On the **See how AWS IoT works** tile, choose **Start the tutorial**.



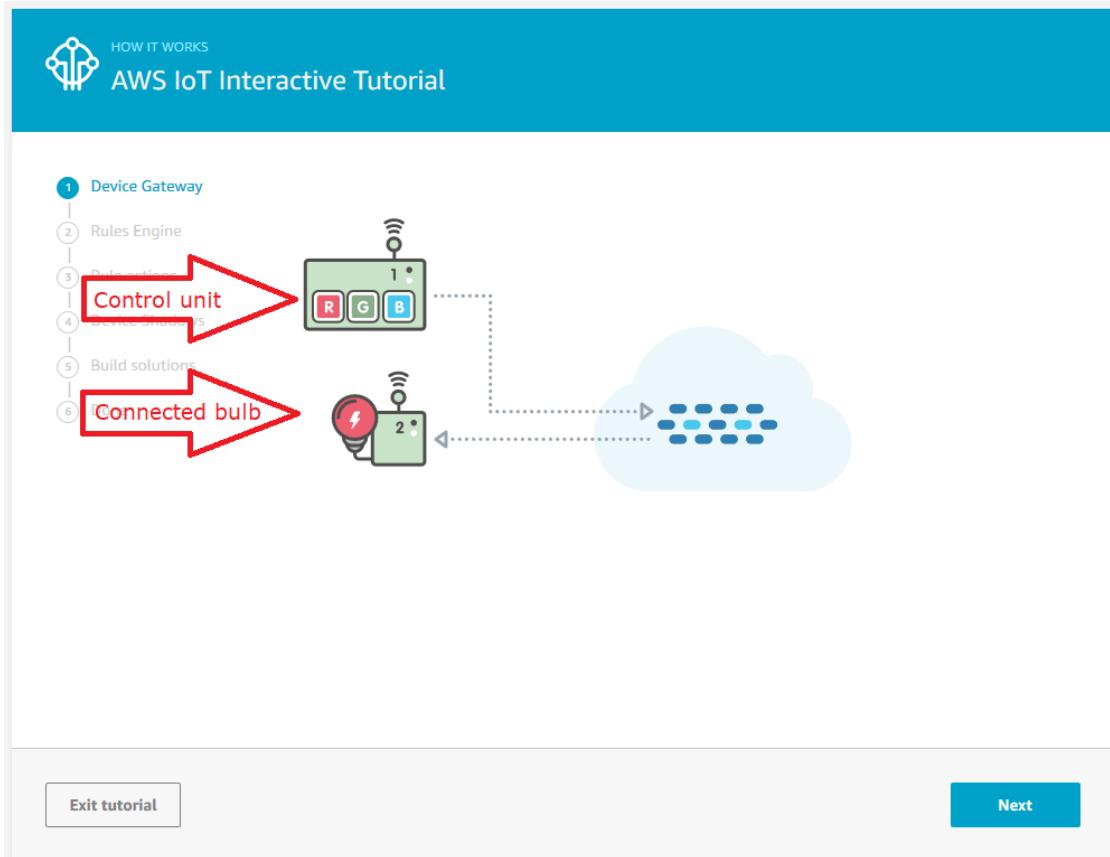
3. On the **AWS IoT Interactive Tutorial** page, review the tutorial overview to get an idea of what you'll learn.

Choose **Start tutorial** when you're ready to continue.

4. **Interactive demo step 1**

Read the instructions in the right panel, which describe the IoT system components.

Choose the different buttons on the control unit and watch the effect on the connected light bulb.



Choose **Next** to continue.

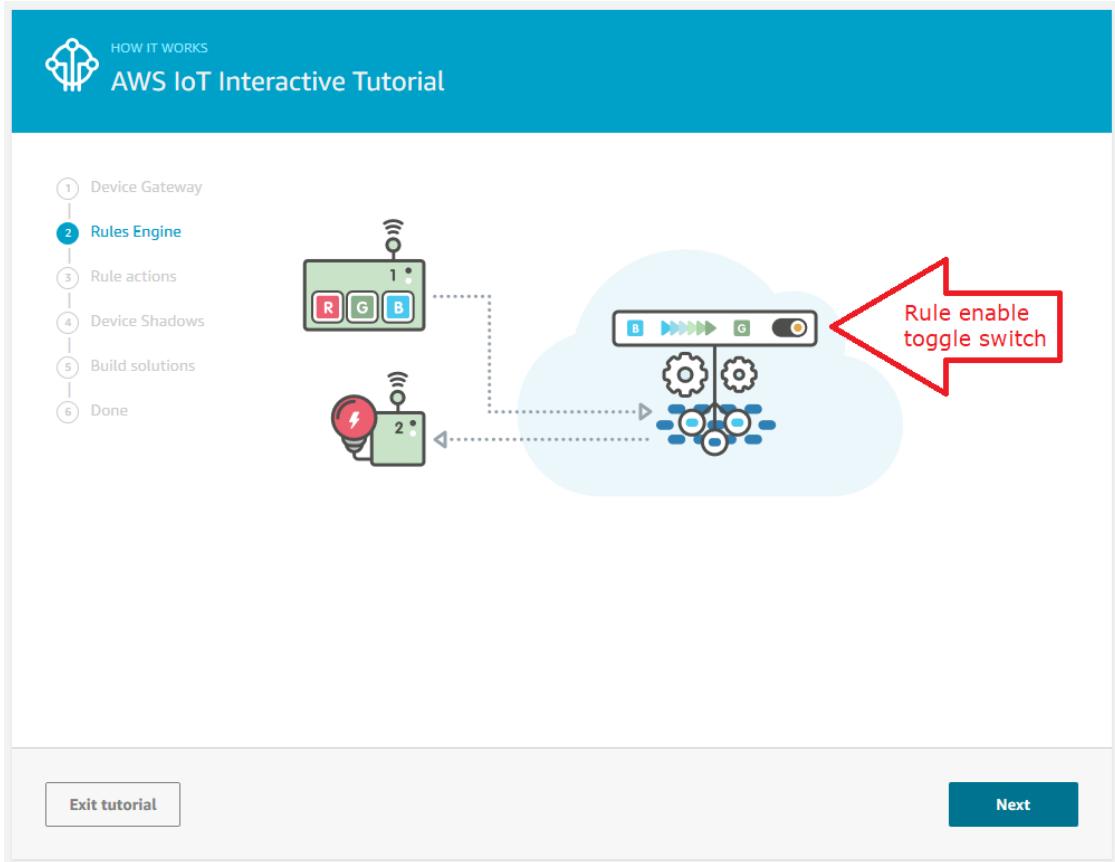
5. Interactive demo step 2

Read the instructions in the right panel, which describe the new components.

Choose the different buttons on the control unit and watch the effect on the connected light bulb.

Move the **Rule enable toggle switch** to disable the rule and choose different buttons on the control unit to see how the bulb behaves differently.

Next



Choose **Next** to continue.

6. Interactive demo step 3

This step adds another rule. Read the instructions in the right panel to understand the new rule, and then try the interactions described in the right panel.

Choose **Next** to continue.

7. Interactive demo step 4

This step adds a device shadow to AWS IoT and a power toggle to the connected bulb.

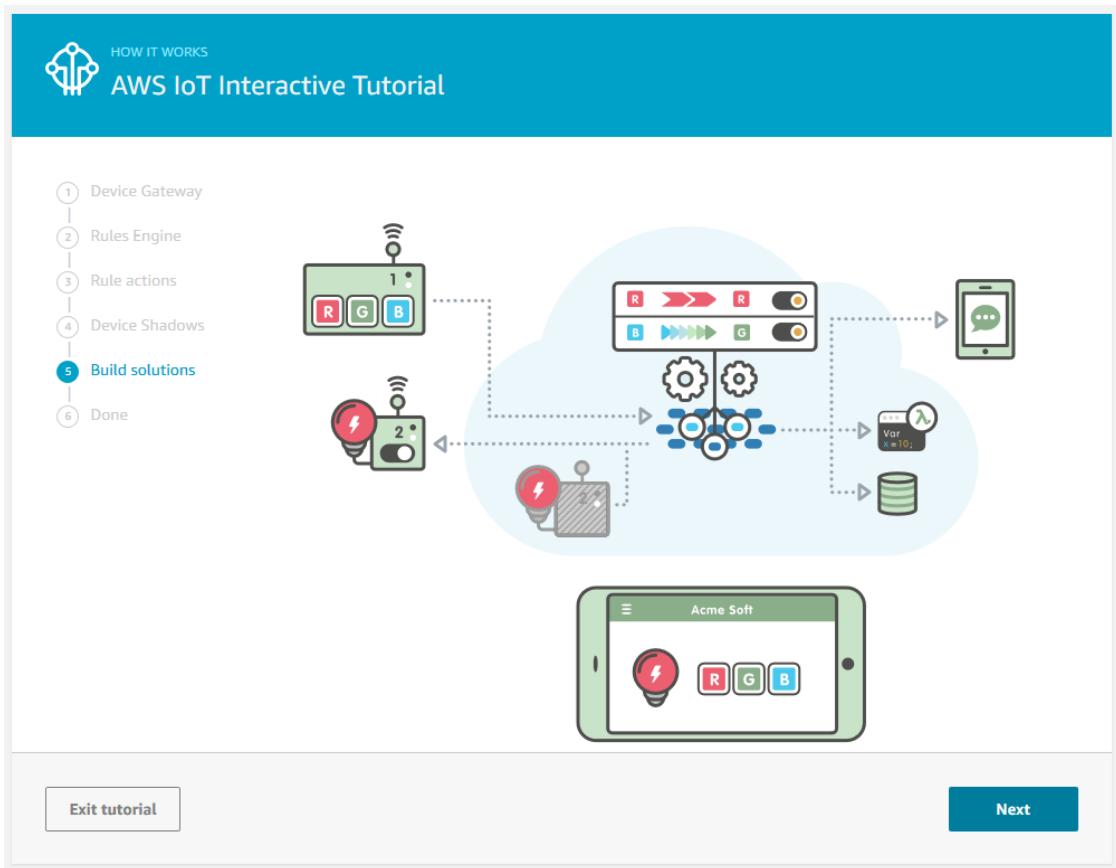
Read the instructions in the right panel to understand how the new components will affect the system operation and then try the interactions described in the right panel.

Choose **Next** to continue.

8. Interactive demo step 5

This step adds a mobile device with an app at the bottom of the screen, which interacts with the IoT solution.

Read the instructions in the right panel to understand how the new components work and then try the interactions described in the right panel.



Choose **Next** to continue.

9. Interactive demo step 6

This is the final step of the demo where you can review and explore how this IoT solution works.

Choose **Get started** to continue.

10. On the Register a thing page, choose **Cancel** to exit the demo.

Try the AWS IoT quick connect

In this tutorial, you'll create your first thing object, connect a device to it, and watch it send MQTT messages.

You can expect to spend 15-20 minutes on this tutorial.

This tutorial is best for people who want to quickly get started with AWS IoT to see how it works in a limited scenario. If you're looking for an example that will get you started so that you can explore more features and services, try [Explore AWS IoT Core services in hands-on tutorial \(p. 33\)](#).

In this tutorial, you'll download and run software on a device that connects to a *thing object* in AWS IoT Core as part of a very small IoT solution. The device can be an IoT device, such as a Raspberry Pi, or it can also be a computer that is running Linux, OS and OSX, or Windows. If you're looking to connect a Long Range WAN (LoRaWAN) device to AWS IoT, refer to the tutorial [Connecting devices and gateways to AWS IoT Core for LoRaWAN \(p. 998\)](#).

If your device supports a browser that can run the [AWS IoT console](#), we recommend you complete this tutorial on that device.

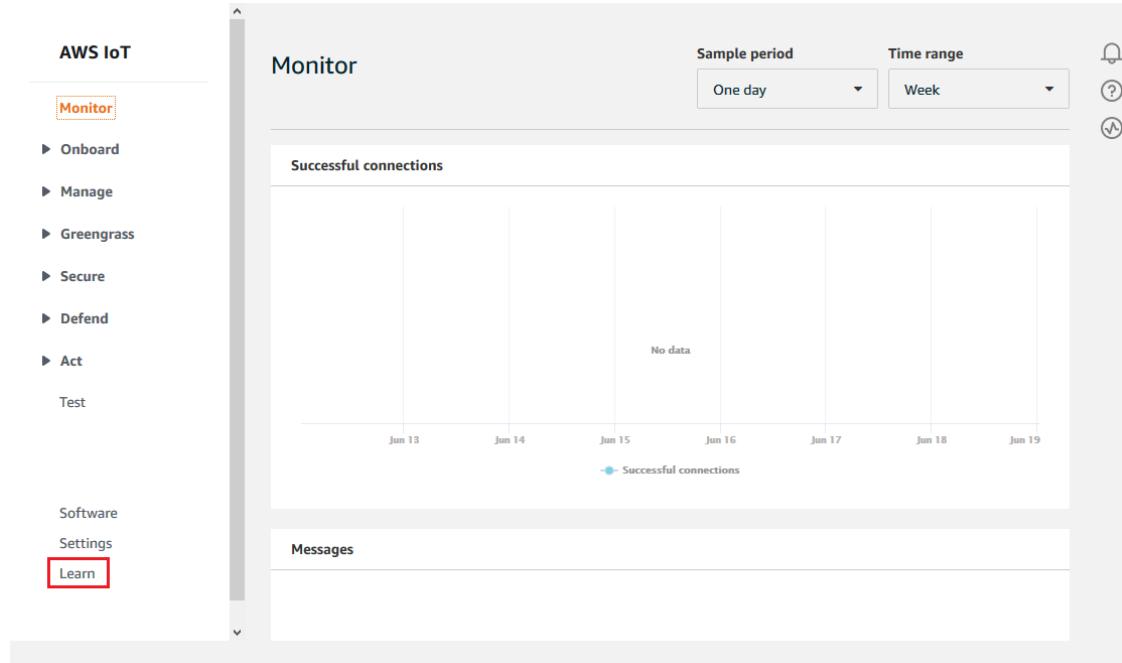
Note

If your device doesn't have a compatible browser, follow this tutorial on a computer. When the procedure asks you to download the file, download it to your computer, and then transfer the downloaded file to your device by using Secure Copy (SCP) or a similar process.

Step 1. Start the tutorial

If possible, complete this procedure on your device; otherwise, be ready to transfer a file to your device later in this procedure.

1. Open your [AWS IoT console](#) and from the left menu, choose **Learn**.



2. On the Connect to AWS IoT tile, choose **View connection options**.

The screenshot shows the AWS IoT Console homepage. On the left, a sidebar menu lists: Monitor, Onboard (selected), Manage, Greengrass, Secure, Defend, Act, Test, Software, Settings, and Learn (highlighted). The main content area is titled "Welcome to the AWS IoT Console" with a sub-instruction: "To get started, you can jump into the recommended starting points below, or explore other learning resources as needed." It features three main tiles: "See how AWS IoT works" (with icons for a speech bubble, car, server, and money), "Connect to AWS IoT" (with a network graph icon), and "Explore documentation" (with icons for a book, laptop, and smartphone). Buttons for "Start the tutorial", "View connection options", and "Go to documentation" are present, with "View connection options" being highlighted with a red box.

3. In the **Onboard a device** tile, choose **Get started**.

The screenshot shows the "Connect to AWS IoT" section of the AWS IoT Console. The sidebar remains the same. The main content area has two tiles: "Onboard a device" (with icons for a car, windmill, and thermometer) and "Onboard many devices" (with a box icon). Buttons for "Get started" and "Create template" are shown, with "Get started" being highlighted with a red box.

4. Review the steps that describe what you'll do in this tutorial. When you're ready to continue, choose **Get started**.

Connect to AWS IoT

Connecting a device (like a development kit or your computer) to AWS IoT requires the completion of the following steps. In this process you will:

-  **Register a device**
A thing is the representation and record of your physical device in the cloud. Any physical device needs a thing record in order to work with AWS IoT.
-  **Download a connection kit**
The connection kit includes some important components: **security credentials, the SDK of your choice, and a sample project**.
-  **Configure and test your device**
Using the connection kit, you will configure your device by **transferring files and running a script**, and **test that it is connected** to AWS IoT correctly.

Want to learn more about the components of AWS IoT?
[Try the interactive overview](#)

[Get started](#)

Step 2. Create a thing object

1. On the **How are you connecting to AWS IoT?** page, choose the platform and the language of the AWS IoT Device SDK that you want to use. This example uses the Linux/OSX platform and the Node.js SDK. If you choose the Python SDK, make sure that you have `python3` and `pip3` installed on your target device before you continue to the next step.

Note

Be sure to check the list of prerequisite software required by your chosen SDK at the bottom of the console page.

You must have the required software installed on your target computer before you continue to the next step.

After you choose the platform and device SDK language, choose **Next**.

How are you connecting to AWS IoT?

Select the platform and SDK that best suits how you are connecting to AWS IoT.

Choose a platform

Linux/OSX



Windows



Choose a AWS IoT Device SDK

Node.js



Python



Java



Some prerequisites to consider:

the device should have **Node.js** and **NPM** installed and a **TCP connection to the public internet on port 8883**.

Looking for AWS IoT Device SDKs and documentation?
[View AWS IoT Device SDKs](#)

Next

2. In the **Name** field, enter the name for your thing object. The thing name used in this example is **MyIotThing**.

Important

Double-check your thing name before you continue.

A thing name can't be changed after the thing object is created. If you want to change a thing name, you must create a new thing object with the correct thing name and then delete the one with the incorrect name.

CONNECT TO AWS IOT

Register a thing

STEP 1/3

A thing is the representation and record of your physical device in the cloud. Any physical device needs a thing to work with AWS IoT. Creating a thing will also create a thing shadow.

[Choose an existing thing instead?](#)

Name

Give your thing a name

Show optional configuration (this can be done later) ▾

Back Next step

3. After you give your thing object a name, choose **Next step**.

CONNECT TO AWS IOT

Register a thing

STEP 1/3

A thing is the representation and record of your physical device in the cloud. Any physical device needs a thing to work with AWS IoT. Creating a thing will also create a thing shadow.

[Choose an existing thing instead?](#)

Name

MylotThing

Show optional configuration (this can be done later) ▾

Back Next step

Step 3. Download files to your device

This page appears after AWS IoT has created the connection kit, which includes the following files and resources that your device requires:

- The thing's certificate files used to authenticate the device
- A policy resource to authorize your thing object to interact with AWS IoT
- The script to download the AWS Device SDK and run the sample program on your device

1. When you're ready to continue, choose the **Download connection kit for** button to download the connection kit for the platform that you chose earlier.

CONNECT TO AWS IOT

Download a connection kit

STEP 2/3

The following AWS IoT resources will be created:

A thing in the AWS IoT registry	MylotThing	
A policy to send and receive messages	MylotThing-Policy	Preview policy

The connection kit contains:

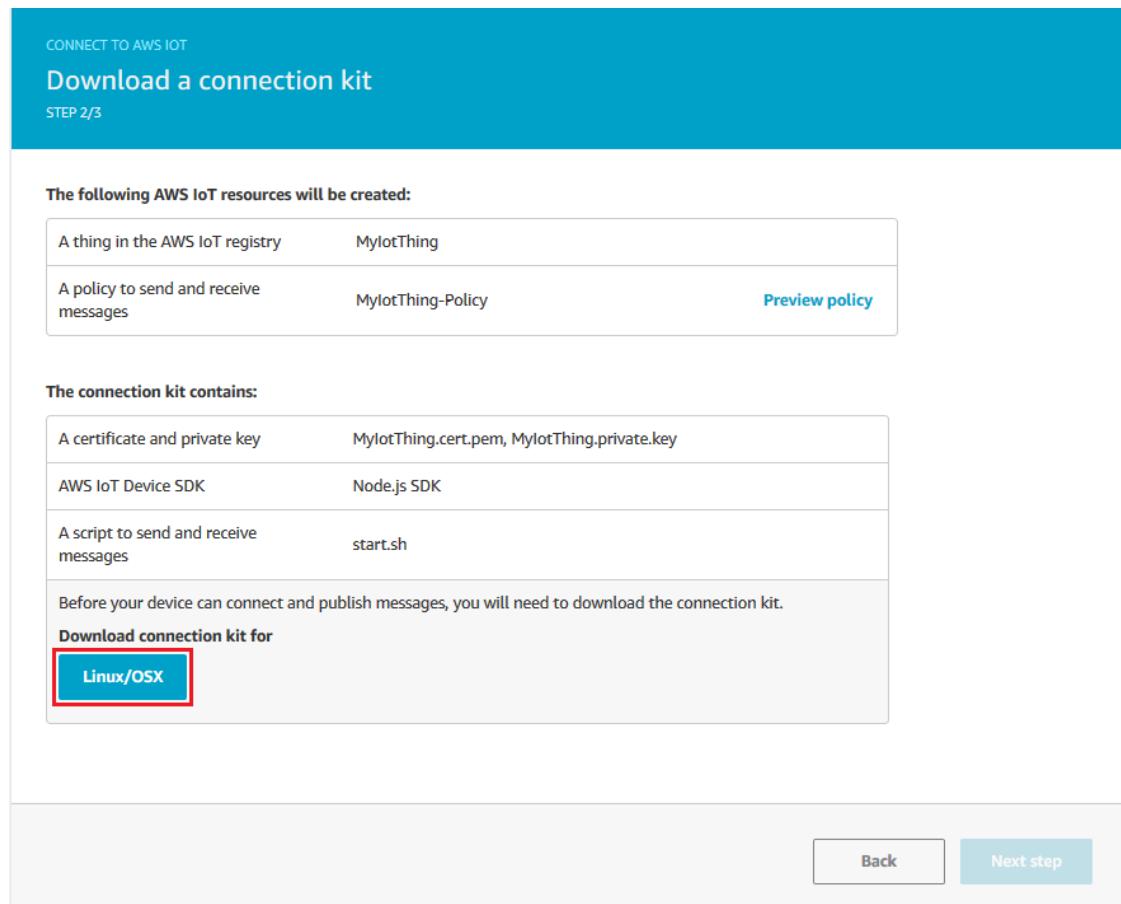
A certificate and private key	MylotThing.cert.pem, MylotThing.private.key
AWS IoT Device SDK	Node.js SDK
A script to send and receive messages	start.sh

Before your device can connect and publish messages, you will need to download the connection kit.

Download connection kit for

[Linux/OSX](#)

[Back](#) [Next step](#)



2. If you're running this procedure on your device, save the connection kit file to a directory from which you can run command line commands.
If you're not running this procedure on your device, save the connection kit file to a local directory and then transfer the file to your device.
3. After you have the connection kit file on the device, continue the tutorial by choosing **Next step**.

CONNECT TO AWS IOT

Download a connection kit

STEP 2/3

The following AWS IoT resources will be created:

A thing in the AWS IoT registry	MylotThing	
A policy to send and receive messages	MylotThing-Policy	Preview policy

The connection kit contains:

A certificate and private key	MylotThing.cert.pem, MylotThing.private.key
AWS IoT Device SDK	Node.js SDK
A script to send and receive messages	start.sh

Before your device can connect and publish messages, you will need to download the connection kit.

Download connection kit for

[Linux/OSX](#)

[Back](#) Next step

Step 4. Run the sample

This procedure is done in a terminal or command window on your device while following the directions displayed in the console. The commands shown in the console are those for the operating system you chose in [the section called “Step 2. Create a thing object” \(p. 26\)](#). Those shown here are for the Linux/OSX operating systems.

1. In a terminal or command window on your device, in the directory with the connection kit file, perform the steps shown in the AWS IoT console.

If you're using a Windows PowerShell command window and the **unzip** command doesn't work, replace **unzip** with **expand-archive** and try the command line again.

The screenshot shows a step-by-step guide for connecting and testing a device. It includes four command-line steps: unzipping the connection kit, adding execution permissions to a shell script, running the script, and finally receiving messages from the device.

```
unzip connect_device_package.zip
chmod +x start.sh
./start.sh
Waiting for messages from your device
```

At the bottom right are 'Back' and 'Done' buttons.

2. In the terminal or command window on your device, after you enter the command from **Step 3** in the console, you should see an output similar to this. This output is from the messages the program is sending to and then receiving back from AWS IoT Core.

While the sample program is running, you can also see some messages in the AWS IoT console. The messages from your device appear in the **Configure and test your device** page, below **Step 3**.

To see more detailed message activity in the console while you run the sample program, see **Step 4** of this procedure. Sometimes, instead of `topic_1`, you might see a message from the terminal that shows it's received from the topic `sdk/test/SDK_programming_language`, where the `SDK_programming_language` can be Python, JavaScript, or Java.

```
Publish received on topic topic_1
{"message":"Hello world!","sequence":1}
Publish received on topic topic_1
 {"message":"Hello world!","sequence":2}
Publish received on topic topic_1
 {"message":"Hello world!","sequence":3}
Publish received on topic topic_1
 {"message":"Hello world!","sequence":4}
Publish received on topic topic_1
 {"message":"Hello world!","sequence":5}
Publish received on topic topic_1
 {"message":"Hello world!","sequence":6}
Publish received on topic topic_1
 {"message":"Hello world!","sequence":7}
Publish received on topic topic_1
 {"message":"Hello world!","sequence":8}
Publish received on topic topic_1
 {"message":"Hello world!","sequence":9}
Publish received on topic topic_1
 {"message":"Hello world!","sequence":10}
```

3. You can repeat the commands from **Step 3/3** in the console of this procedure), to run the sample program again.
4. (Optional) If you want to see the messages from your IoT client in the [AWS IoT console](#), open the [MQTT test client](#) on the **Test** page of the AWS IoT console. In the **MQTT test client**, in **Topic filter**, enter `sdk/test/SDK_programming_language` to subscribe to the messages from your device.

The complete topic filter to enter depends on the programming language of the SDK you chose in **Step 1/1**. The possible topic filters are shown here and are case sensitive.

- For the AWS IoT Device SDK, the topic is **sdk/test/javascript**.
 - For the Python AWS IoT Device SDK, the topic is **sdk/test/Python**.
 - For the Java AWS IoT Device SDK, the topic is **sdk/test/java**.
5. After you subscribe to the test topic, run this program on your device **./start.sh** as described in the previous step. For more information, see [the section called "View MQTT messages with the AWS IoT MQTT client" \(p. 62\)](#) for more information.

After you run **./start.sh**, you'll see messages displayed in the MQTT client similar to the following:

```
{  
  "message": "Hello World!",  
  "sequence": 10  
}
```

The sequence number increments by one each time a new Hello World message is received and stops when you terminate the program.

6. After you've finished running the program on your device, in the AWS IoT console, choose **Done** to finish the tutorial and see this summary.

Connected successfully

A device was connected to AWS IoT by performing some tasks in AWS IoT and on the device.



Registered a thing to represent a device in AWS IoT

[Learn more](#)



Set up security for the device using a certificate and policy

[Learn more](#)



Used a device SDK to connect a device to AWS IoT

[Learn more](#)



Received messages from the device

[Learn more](#)

Want to learn more about the components of AWS IoT?
[Try the interactive overview](#)

[Done](#)

Step 5. Explore further

Here are some ideas to explore AWS IoT further after you complete the quick start.

- [View MQTT messages in the MQTT client](#)

From the [AWS IoT console](#), you can open the [MQTT client](#) on the **Test** page of the AWS IoT console. In the **MQTT client**, subscribe to `#`, and then, on your device, run the program `./start.sh` as described in the previous step. For more information, see [the section called “View MQTT messages with the AWS IoT MQTT client” \(p. 62\)](#).

- [the section called “Try the AWS IoT Core interactive demo” \(p. 19\)](#)

To start the interactive tutorial, from the **Learn** page of the AWS IoT console, in the **See how AWS IoT works** tile, choose **Start the tutorial**.

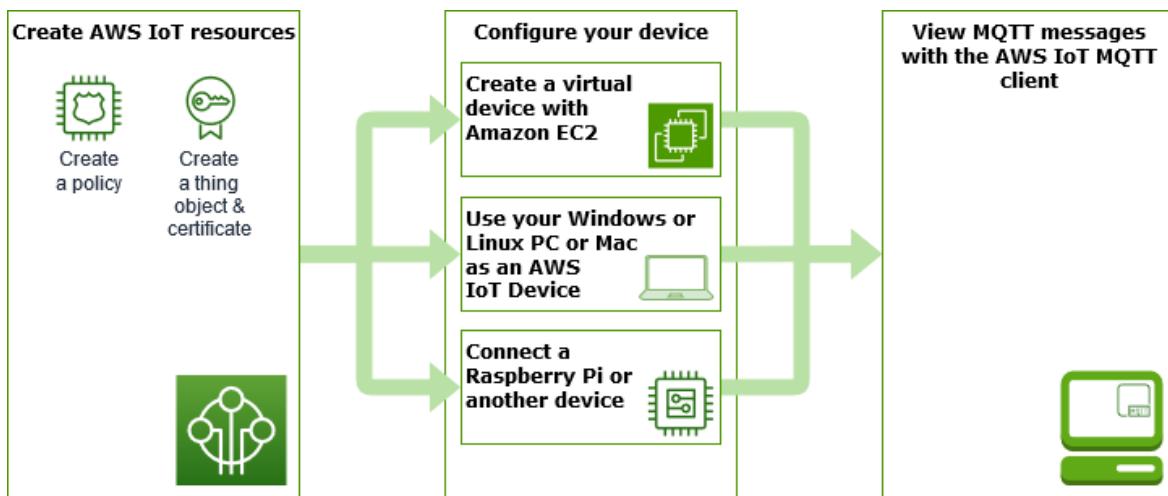
- [Get ready to explore more tutorials \(p. 33\)](#)

This quick start gives you just a sample of AWS IoT. If you want to explore AWS IoT further and learn about the features that make it a powerful IoT solution platform, start preparing your development platform by [Explore AWS IoT Core services in hands-on tutorial \(p. 33\)](#).

Explore AWS IoT Core services in hands-on tutorial

In this tutorial, you'll install the software and create the AWS IoT resources necessary to connect a device to AWS IoT so that it can send and receive MQTT messages with AWS IoT Core. You'll see the messages in the MQTT client in the AWS IoT console.

You can expect to spend 20-30 minutes on this tutorial. If you are using an IoT device or a Raspberry Pi, this tutorial might take longer if, for example, you need to install the operating system and configure the device.



This tutorial is best for developers who want to get started with AWS IoT so they can continue to explore more advanced features, such as the rules engine and shadows. This tutorial prepares you to continue learning about AWS IoT Core and how it interacts with other AWS services by explaining the steps in greater detail than the quick start tutorial. If you are looking for just a quick, *Hello World*, experience, try the [Try the AWS IoT quick connect \(p. 23\)](#).

After setting up your AWS account and AWS IoT console, you'll follow these steps to see how to connect a device and have it send messages to AWS IoT.

Next steps

- [Choose which device option is the best for you \(p. 34\)](#)
- [the section called “Create AWS IoT resources” \(p. 34\) if you are not going to create a virtual device with Amazon EC2.](#)

- the section called “Configure your device” (p. 38)
- the section called “View MQTT messages with the AWS IoT MQTT client” (p. 62)

For more information about AWS IoT Core, see [What Is AWS IoT Core \(p. 1\)?](#)

Which device option is the best for you?

If you're not sure which option to pick, use the following list of each option's advantages and disadvantages to help you decide which one is best for you.

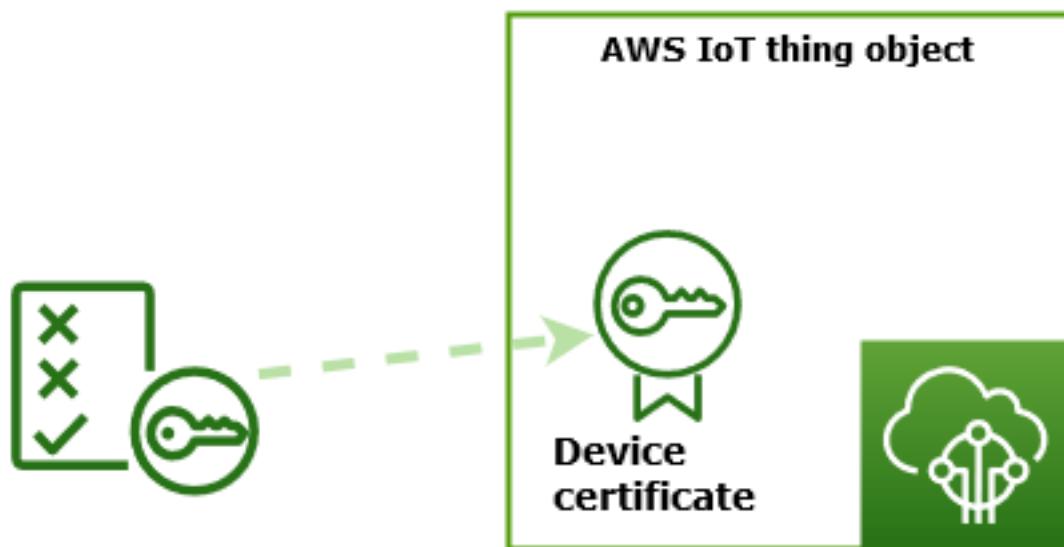
Option	This might be a good option if:	This might not be a good option if:
the section called “Create a virtual device with Amazon EC2” (p. 39)	<ul style="list-style-type: none">• You don't have your own device to test.• You don't want to install any software on your own system.• You want to test on a Linux OS.	<ul style="list-style-type: none">• You're not comfortable using command-line commands.• You don't want to incur any additional AWS charges.• You don't want to test on a Linux OS.
the section called “Use your Windows or Linux PC or Mac as an AWS IoT device” (p. 47)	<ul style="list-style-type: none">• You don't want to incur any additional AWS charges.• You don't want to configure any additional devices.	<ul style="list-style-type: none">• You don't want to install any software on your personal computer.• You want a more representative test platform.
the section called “Connect a Raspberry Pi or another device” (p. 53)	<ul style="list-style-type: none">• You want to test AWS IoT with an actual device.• You already have a device to test with.• You have experience integrating hardware into systems.	<ul style="list-style-type: none">• You don't want to buy or configure a device just to try it out.• You want to test AWS IoT as simply as possible, for now.

Create AWS IoT resources

In this tutorial, you'll create the AWS IoT resources that a device requires to connect to AWS IoT and exchange messages.

Create an AWS IoT Core policy

Create a thing and its certificate



1. Create an AWS IoT policy document, which will authorize your device to interact with AWS IoT services.
2. Create a thing object in AWS IoT and its X.509 device certificate, and then attach the policy document. The thing object is the virtual representation of your device in the AWS IoT registry. The certificate authenticates your device to AWS IoT Core, and the policy document authorizes your device to interact with AWS IoT.

Note

If you are planning to [the section called “Create a virtual device with Amazon EC2” \(p. 39\)](#),

you can skip this page and continue to [the section called “Configure your device” \(p. 38\)](#).

You will create these resources when you create your virtual thing.

This tutorial uses the AWS IoT console to create the AWS IoT resources. If your device supports a web browser, it might be easier to run this procedure on the device's web browser because you will be able to download the certificate files directly to your device. If you run this procedure on another computer, you will need to copy the certificate files to your device before they can be used by the sample app.

Create an AWS IoT policy

X.509 certificates are used to authenticate your device with AWS IoT Core. AWS IoT policies are attached to the certificate that authenticates the device to determine the AWS IoT operations, such as subscribing or publishing to MQTT topics that the device is permitted to perform. Your device presents its certificate when it connects and sends messages to AWS IoT Core.

In this procedure, you will create a policy that allows your device to perform the AWS IoT operations necessary to run the example program. You must create the AWS IoT policy first, so that you can attach it to the device certificate that you will create later.

To create an AWS IoT policy

1. In the left menu, choose **Secure**, and then choose **Policies**. On the **You don't have a policy yet** page, choose **Create a policy**.

If your account has existing policies, choose **Create**.

2. On the **Create a policy** page:

1. In the **Name** field, enter a name for the policy (for example, **My_Iot_Policy**). Do not use personally identifiable information in your policy names.
2. In the **Action** field, enter **iot:Connect, iot:Receive, iot:Publish, iot:Subscribe**. These are the actions that the device will need permission to perform when it runs the example program from the Device SDK.

For more information about IoT policies, see [AWS IoT Core policies \(p. 314\)](#).

3. In the **Resource ARN** field, enter *. This selects any client (device).

Note

In this quick start, the wildcard (*) character is used for simplicity. For higher security, you should restrict which clients (devices) can connect and publish messages by specifying a client ARN (Amazon resource name) instead of the wildcard character as the resource. Client ARNs follow this format:

`arn:aws:iot:your-region:your-aws-account:client/my-client-id`

However, you must first create the resource (client device, thing shadow, etc.) before you can assign its ARN to a policy.

4. Choose the **Allow** check box.

These values allows all clients that have this policy attached to their certificate to perform the actions listed in the **Action** field.

Create a policy

Create a policy to define a set of authorized actions. You can authorize actions on one or more resources (things, topics, topic filters). To learn more about IoT policies go to the [AWS IoT Policies documentation page](#).

Name

My_IoT_Policy

Add statements

Policy statements define the types of actions that can be performed by a resource.

Action

iot:Connect,iot:Receive,iot:Publish,iot:Subscribe

Resource ARN

*

Effect

Allow Deny

Remove

Add statement

Create

The screenshot shows the 'Create a policy' interface. At the top, there's a teal header bar with the title 'Create a policy'. Below it is a main form area. In the 'Name' field, 'My_IoT_Policy' is entered. Under the 'Add statements' section, there's a single statement defined with the following details: Action: 'iot:Connect,iot:Receive,iot:Publish,iot:Subscribe', Resource ARN: '*', Effect: 'Allow' (checked). There's also a 'Remove' button next to the statement. At the bottom right of the form is a large blue 'Create' button.

3. After you have entered the information for your policy, choose **Create**.

For more information, see [IAM policies \(p. 368\)](#).

Create a thing object

Devices connected to AWS IoT are represented by *thing objects* in the AWS IoT registry. A *thing object* represents a specific device or logical entity. It can be a physical device or sensor (for example, a light bulb or a light switch on the wall). It can also be a logical entity, like an instance of an application or physical entity that doesn't connect to AWS IoT, but is related to other devices that do (for example, a car that has engine sensors or a control panel).

To create a thing in the AWS IoT console

1. In the [AWS IoT console](#), in the left menu, choose **Manage**, then choose **Things**.
2. On the **Things** page, choose **Create things**.
3. On the **Create things** page, choose **Create a single thing**, then choose **Next**.
4. On the **Specify thing properties** page, for **Thing name**, enter a name for your thing, such as **MyIotThing**.

When naming things, choose the name carefully, because you can't change a thing name after you create it.

To change a thing's name, you must create a new thing, give it the new name, and then delete the old thing.

Note

Do not use personally identifiable information in your thing name. The thing name can appear in unencrypted communications and reports.

5. Keep the rest of the fields on this page empty. Choose **Next**.
6. On the **Configure device certificate - optional** page, choose **Auto-generate a new certificate (recommended)**. Choose **Next**.
7. On the **Attach policies to certificate - optional** page, select the policy you created in the previous section. In that section, the policy was named, **My_Iot_Policy**. Choose **Create thing**.
8. On the **Download certificates and keys** page:
 1. Download each of the certificate and key files and save them for later. You'll need to install these files on your device.

When you save your certificate files, give them the names in the following table. These are the file names used in later examples.

Certificate file names

File	File path
Private key	<code>private.pem.key</code>
Public key	(not used in these examples)
Device certificate	<code>device.pem.crt</code>
Root CA certificate	<code>Amazon-root-CA-1.pem</code>

2. Download the root CA file for these files by choosing the **Download** link of the root CA certificate file that corresponds to the type of data endpoint and cipher suite you're using. In this tutorial, choose **Download** to the right of **RSA 2048 bit key: Amazon Root CA 1** and download the **RSA 2048 bit key: Amazon Root CA 1** certificate file.

Important

You must save the certificate files before you leave this page. After you leave this page in the console, you will no longer have access to the certificate files.

If you forgot to download the certificate files that you created in this step, you must exit this console screen, go to the list of things in the console, delete the thing object you created, and then restart this procedure from the beginning.

3. Choose **Done**.

After you complete this procedure, you should see the new thing object in your list of things.

Configure your device

This section describes how to configure your device to connect to AWS IoT. If you'd like to get started with AWS IoT but don't have a device yet, you can create a virtual device by using Amazon EC2 or you can use your Windows PC or Mac as an IoT device.

Select the best device option for you to try AWS IoT. Of course, you can try all of them, but try only one at a time. If you're not sure which device option is best for you, read about how to choose [which device option is the best \(p. 34\)](#), and then return to this page.

Device options

- [Create a virtual device with Amazon EC2 \(p. 39\)](#)
- [Use your Windows or Linux PC or Mac as an AWS IoT device \(p. 47\)](#)
- [Connect a Raspberry Pi or another device \(p. 53\)](#)

Create a virtual device with Amazon EC2

In this tutorial, you'll create an Amazon EC2 instance to serve as your virtual device in the cloud.

To complete this tutorial, you need an AWS account. If you don't have one, complete the steps described in [Set up your AWS account \(p. 17\)](#) before you continue.

In this tutorial, you'll:

- [Set up an Amazon EC2 instance \(p. 39\)](#)
- [Install Git, Node.js and configure the AWS CLI \(p. 40\)](#)
- [Create AWS IoT resources for your virtual device \(p. 41\)](#)
- [Install the AWS IoT Device SDK for JavaScript \(p. 44\)](#)
- [Run the sample application \(p. 45\)](#)
- [View messages from the sample app in the AWS IoT console \(p. 46\)](#)

Set up an Amazon EC2 instance

The following steps show you how to create an Amazon EC2 instance that will act as your virtual device in place of a physical device.

To launch an instance

1. Open the Amazon EC2 console at <https://console.aws.amazon.com/ec2/>.
2. From the console dashboard, choose **Launch Instance**.
3. The **Step 1: Choose an Amazon Machine Image (AMI)** page displays a list of basic configurations, called *Amazon Machine Images (AMIs)*, which serve as templates for your instance. Select an HVM version of Amazon Linux 2, such as *Amazon Linux 2 AMI (HVM), SSD Volume Type*. Notice that this AMI is marked "Free tier eligible."
4. On the **Choose an Instance Type** page, you can select the hardware configuration of your instance. Select the *t2.micro* type, which is selected by default. Notice that this instance type is eligible for the free tier.
5. Choose **Review and Launch** to let the wizard complete the other configuration settings for you.
6. On the **Review Instance Launch** page, choose **Launch**.
7. When prompted for a key pair, select **Create a new key pair**, enter a name for the key pair, and then choose **Download Key Pair**. *This is your only chance to save the private key file, so be sure to download it.* Save the private key file in a safe place. You'll need to provide the name of your key pair when you launch an instance and the corresponding private key each time you connect to the instance.

Warning

Don't select the **Proceed without a key pair** option. If you launch your instance without a key pair, then you can't connect to it.

When you are ready, choose **Launch Instances**.

8. A confirmation page lets you know that your instance is launching. Choose **View Instances** to close the confirmation page and return to the console.
9. On the **Instances** screen, you can view the status of the launch. It takes a short time for an instance to launch. When you launch an instance, its initial state is **Pending**. After the instance starts, its state changes to **running** and it receives a public DNS name. (If the **Public DNS (IPv4)** column is hidden, choose **Show/Hide Columns** (the gear-shaped icon) in the top right corner of the page and then select **Public DNS (IPv4)**.)
10. It can take a few minutes for the instance to be ready so that you can connect to it. Check that your instance has passed its status checks; you can view this information in the **Status Checks** column.

After your new instance has passed its status checks, continue to the next procedure and connect to it.

To connect to your instance

You can connect to an instance using the browser-based client by selecting the instance from the Amazon EC2 console and choosing to connect using Amazon EC2 Instance Connect. Instance Connect handles the permissions and provides a successful connection.

1. Open the Amazon EC2 console at <https://console.aws.amazon.com/ec2/>.
2. In the left menu, choose **Instances**.
3. Select the instance and choose **Connect**.
4. Choose **Amazon EC2 Instance Connect (browser-based SSH connection)**, **Connect**.

You should now have an **Amazon EC2 Instance Connect** window that is logged into your new Amazon EC2 instance.

Install Git, Node.js and configure the AWS CLI

In this section, you'll install Git and Node.js, on your Linux instance.

To install Git

1. In your **Amazon EC2 Instance Connect** window, update your instance by using the following command.

```
sudo yum update -y
```

2. In your **Amazon EC2 Instance Connect** window, install Git by using the following command.

```
sudo yum install git -y
```

To install Node.js

1. In your **Amazon EC2 Instance Connect** window, install node version manager (nvm) by using the following command.

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.34.0/install.sh | bash
```

We will use nvm to install Node.js because nvm can install multiple versions of Node.js and allow you to switch between them.

2. In your **Amazon EC2 Instance Connect** window, activate nvm by using this command.

```
. ~/nvm/nvm.sh
```

3. In your **Amazon EC2 Instance Connect** window, use nvm to install the latest version of Node.js by using this command.

```
nvm install node
```

Installing Node.js also installs the Node Package Manager (npm) so you can install additional modules as needed.

4. In your **Amazon EC2 Instance Connect** window, test that Node.js is installed and running correctly by using this command.

```
node -v
```

This tutorial requires Node v10.0 or later.

To configure AWS CLI

Your Amazon EC2 instance comes preloaded with the AWS CLI. However, you must complete your AWS CLI profile. For more information on how to configure your CLI, see [Configuring the AWS CLI](#).

1. The following example shows sample values. Replace them with your own values. You can find these values in your [AWS console in your account info under Security credentials](#).

In your **Amazon EC2 Instance Connect** window, enter this command:

```
aws configure
```

Then enter the values from your account at the prompts displayed.

```
AWS Access Key ID [None]: AKIAIOSFODNN7EXAMPLE
AWS Secret Access Key [None]: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
Default region name [None]: us-west-2
Default output format [None]: json
```

2. You can test your AWS CLI configuration with this command:

```
aws iot describe-endpoint --endpoint-type iot:Data-ATS
```

If your AWS CLI is configured correctly, the command should return an endpoint address from your AWS account.

Create AWS IoT resources for your virtual device

This section describes how to use the AWS CLI to create the thing object and its certificate files directly on the virtual device. This is done directly on the device to avoid the potential complication that might arise from copying them to the device from another computer.

To create an AWS IoT thing object in your Linux instance

Devices connected to AWS IoT are represented by *thing objects* in the AWS IoT registry. A *thing object* represents a specific device or logical entity. In this case, your *thing object* will represent your virtual device, this Amazon EC2 instance.

1. In your **Amazon EC2 Instance Connect** window, run the following command to create your thing object.

```
aws iot create-thing --thing-name "MyIoTThing"
```

2. The JSON response should look like this:

```
{
    "thingArn": "arn:aws:iot:<your-region>:<your-aws-account>:thing/MyIoTThing",
    "thingName": "MyIoTThing",
    "thingId": "6cf922a8-d8ea-4136-f3401EXAMPLE"
}
```

To create and attach AWS IoT keys and certificates in your Linux instance

The **create-keys-and-certificate** command creates client certificates signed by the Amazon Root certificate authority. This certificate is used to authenticate the identity of your virtual device.

1. In your **Amazon EC2 Instance Connect** window, create a directory to store your certificate and key files.

```
mkdir ~/certs
```

2. In your **Amazon EC2 Instance Connect** window, download a copy of the Amazon certificate authority (CA) certificate by using this command.

```
curl -o ~/certs/Amazon-root-CA-1.pem \
      https://www.amazontrust.com/repository/AmazonRootCA1.pem
```

3. In your **Amazon EC2 Instance Connect** window, run the following command to create your private key, public key, and X.509 certificate files. This command also registers and activates the certificate with AWS IoT.

```
aws iot create-keys-and-certificate \
    --set-as-active \
    --certificate-pem-outfile "~/certs/device.pem.crt" \
    --public-key-outfile "~/certs/public.pem.key" \
    --private-key-outfile "~/certs/private.pem.key"
```

The response looks like the following. Save the `certificateArn` so that you can use it in subsequent commands. You'll need it to attach your certificate to your thing and to attach the policy to the certificate in a later steps.

```
{
    "certificateArn": "arn:aws:iot:us-
west-2:123456789012:cert/9894ba17925e663f1d29c23af4582b8e3b7619c31f3fb93adcb51ae54b83dc2",
    "certificateId": "9894ba17925e663f1d29c23af4582b8e3b7619c31f3fb93adcb51ae54b83dc2",
    "certificatePem": "
-----BEGIN CERTIFICATE-----
MIICitCCExAMPLE6m7oRw0uXOjANBgkqhkiG9w0BAQUFADCBiDELMAkGA1UEBhMC
VVMxCzAJBgNVBAgEXAMPEAwDgYDVQQHEwdTZWF0dGxlMQ8wDQYDVQQKEwZBbWF6
b24xFDASBgNVBASTC0lBTSEXAMPLE2xLMRIwEAYDVQQDEwlUZXN0Q2lsYWmxHzAd
BgkqhkiG9w0BCQEWEg5vb25lQGFtYEXAMPEb20wHhcNMTEwNDI1MjA0NTIxWhcN
MTIwNDI0MjA0NTIxWjCBiDELMAkGA1UEBhMCEXAMPLEJBgNVBAgTAldBMRAwDgYD
VQQHEwdTZWF0dGxlMQ8wDQYDVQQKEwZBbWF6b24xFDAEXAMPLESTC0lBTSBdb25z
b2xLMRIwEAYDVQQDEwlUZXN0Q2lsYWmxHzAdBgkqhkiG9w0BCQEEXAMPLE25lQGFt
YXpvbi5jb20wgZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBAMA0dn+aEXAMPLE
-----END CERTIFICATE-----"
```

```

EXAMPLEfEvySWtC2XADZ4nB+BLYgVIk60CpiwsZ3G93vUEIO3IyNoH/f0wYK8m9T
rDHudUZEXAMPLEG5M43q7Wgc/MbQITxOUSQv7c7ugFFDzQGBzZswY6786m86gpE
Ibb3OhjZnzcvcQAEEXAMPLEWimm2nrAgMBAEwDQYJKoZIhvcNAQEFBQADgYEAtCu4
nUhVVxYUntneD9+h8Mg9qEXAMPLEyExzyLwaxlAoo7TJHiibts4J5iNmZgXL0Fkb
FFBjvSfpJ1lJ00zbhNYS5f6GuoEDEXAMPLEBHjJnyp3780D8uTs7fLvjx79LjSTb
NYiytVbZPQU05Yaxu2jXnimvw3rrszlaEXAMPLE=
-----END CERTIFICATE-----\n",
    "keyPair": {
        "PublicKey": "-----BEGIN PUBLIC
KEY-----\nMIIBIjANBgkqhKEEXAMPLEQEFAAOC98AMIIIBCgKCAQEAEEXAMPLE1nnnyJwKSMHw4h
\nMMEXAMPLEuuN/dMAS3fyce8DW/4+EXAMPLEyjmof/YVF/
gHr99VEEXAMPLE5VF13\n59VK7cEXAMPLE67GK+y+jikqXOgHh/xJTw
+sGpWEXAMPLEDz18xDzka4tCzuWEXAMPLEahJbYKCPUBSU8opVkr7qkEXAMPLE1DR6sx2HocliOOLtu6Fkw91swQWEXAMPLE
\GB3ZPrNh0PzQYvJUStZeccyNCx2EXAMPLEvp9mOUXP6plfgxwKRX2fEXAMPLEdA\nhJLXkX3rHU2xbxJSq7D
+XEXAMPLEcw+LyFhI5mgFRL88eGdsAEXAMPLElnI9EesG\nnFQIDAQAB\-----END PUBLIC KEY-----\n",
        "PrivateKey": "-----BEGIN RSA PRIVATE KEY-----\nkey omitted for security
reasons\n-----END RSA PRIVATE KEY-----\n"
    }
}

```

- In your **Amazon EC2 Instance Connect** window, attach your thing object to the certificate you just created by using the following command and the ***certificateArn*** in the response from the previous command.

```
aws iot attach-thing-principal \
--thing-name "MyIoTThing" \
--principal "certificateArn"
```

If successful, this command does not display any output.

To create and attach a policy

- In your **Amazon EC2 Instance Connect** window, create the policy file by copying and pasting this policy document to a file named ***~/policy.json***.

If you don't have a favorite Linux editor, you can open **nano**, by using this command.

```
nano ~/policy.json
```

And pasting the policy document for **policy.json** into it. Enter **ctrl-x** to exit the **nano** editor and save the file.

Contents of the policy document for **policy.json**.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish",
                "iot:Subscribe",
                "iot:Receive",
                "iot:Connect"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

```
}
```

2. In your **Amazon EC2 Instance Connect** window, create your policy by using the following command.

```
aws iot create-policy \
--policy-name "MyIoTThingPolicy" \
--policy-document "file://~/policy.json"
```

Output:

```
{
  "policyName": "MyIoTThingPolicy",
  "policyArn": "arn:aws:iot:<your-region>:<your-aws-account>:policy/MyIoTThingPolicy",
  "policyDocument": "{\n    \"Version\": \"2012-10-17\", \n    \"Statement\": [\n        {\n            \"Effect\": \"Allow\", \n            \"Action\": [\n                \"iot:Publish\", \n                \"iot:Receive\", \n                \"iot:Subscribe\", \n                \"iot:Connect\"\n            ], \n            \"Resource\": [\n                \"*\"\n            ]\n        }\n    ]\n}, \n  \"policyVersionId\": \"1\"\n}
```

3. In your **Amazon EC2 Instance Connect** window, attach the policy to your virtual device's certificate by using the following command.

```
aws iot attach-policy \
--policy-name "MyIoTThingPolicy" \
--target "<certificateArn>"
```

If successful, this command does not display any output.

At this point, you have created for your virtual device:

- A thing object to represent your virtual device in AWS IoT.
- A certificate to authenticate your virtual device.
- A policy document to authorize your virtual device to Connect to AWS IoT, and to Publish, Receive, and Subscribe to messages.

Install the AWS IoT Device SDK for JavaScript

In this section, you'll install the AWS IoT Device SDK for JavaScript, which contains the code that applications can use to communicate with AWS IoT and the sample programs.

To install the AWS IoT Device SDK for JavaScript on your Linux instance

1. In your **Amazon EC2 Instance Connect** window, clone the AWS IoT Device SDK for JavaScript repository into the `aws-iot-device-sdk-js-v2` directory of your home directory by using this command.

```
cd ~  
git clone https://github.com/aws/aws-iot-device-sdk-js-v2.git
```

2. Navigate to the `aws-iot-device-sdk-js-v2` directory that you created in the preceding step.

```
cd aws-iot-device-sdk-js-v2
```

3. Use `npm` to install the SDK.

```
npm install
```

Run the sample application

The commands in the next sections assume that your key and certificate files are stored on your virtual device as shown in this table.

Certificate file names

File	File path
Private key	<code>~/certs/private.pem.key</code>
Device certificate	<code>~/certs/device.pem.crt</code>
Root CA certificate	<code>~/certs/Amazon-root-CA-1.pem</code>

In this section, you'll install and run the `pub-sub.js` sample app found in the `aws-iot-device-sdk-js-v2/samples/node` directory of the AWS IoT Device SDK for JavaScript. This app shows how a device, your Amazon EC2 instance, uses the MQTT library to publish and subscribe to MQTT messages. The `pub-sub.js` sample app subscribes to a topic, `topic_1`, publishes 10 messages to that topic, and displays the messages as they're received from the message broker.

To install and run the sample app

1. In your **Amazon EC2 Instance Connect** window, navigate to the `aws-iot-device-sdk-js-v2/samples/node/pub_sub` directory that the SDK created and install the sample app by using these commands.

```
cd ~/aws-iot-device-sdk-js-v2/samples/node/pub_sub  
npm install
```

2. In your **Amazon EC2 Instance Connect** window, get `your-iot-endpoint` from AWS IoT by using this command.

```
aws iot describe-endpoint --endpoint-type iot:Data-ATS
```

3. In your **Amazon EC2 Instance Connect** window, insert `your-iot-endpoint` as indicated and run this command.

```
node dist/index.js --topic topic_1 --root-ca ~/certs/Amazon-root-CA-1.pem --cert ~/certs/device.pem.crt --key ~/certs/private.pem.key --endpoint your-iot-endpoint
```

The sample app:

1. Connects to the AWS IoT service for your account.
2. Subscribes to the message topic, **topic_1**, and displays the messages it receives on that topic.
3. Publishes 10 messages to the topic, **topic_1**.
4. Displays output similar to the following:

```
Publish received on topic topic_1
{"message":"Hello world!","sequence":1}
Publish received on topic topic_1
{"message":"Hello world!","sequence":2}
Publish received on topic topic_1
{"message":"Hello world!","sequence":3}
Publish received on topic topic_1
{"message":"Hello world!","sequence":4}
Publish received on topic topic_1
{"message":"Hello world!","sequence":5}
Publish received on topic topic_1
{"message":"Hello world!","sequence":6}
Publish received on topic topic_1
{"message":"Hello world!","sequence":7}
Publish received on topic topic_1
{"message":"Hello world!","sequence":8}
Publish received on topic topic_1
{"message":"Hello world!","sequence":9}
Publish received on topic topic_1
{"message":"Hello world!","sequence":10}
```

If you're having trouble running the sample app, review [the section called "Troubleshooting problems with the sample app" \(p. 60\)](#).

You can also add the `--verbosity Debug` parameter to the command line so the sample app displays detailed messages about what it's doing. That information might provide you the help you need to correct the problem.

[View messages from the sample app in the AWS IoT console](#)

You can see the sample app's messages as they pass through the message broker by using the **MQTT client** in the **AWS IoT console**.

To view the MQTT messages published by the sample app

1. Review [View MQTT messages with the AWS IoT MQTT client \(p. 62\)](#). This helps you learn how to use the **MQTT client** in the **AWS IoT console** to view MQTT messages as they pass through the message broker.
2. Open the **MQTT client** in the **AWS IoT console**.
3. Subscribe to the topic, **topic_1**.
4. In your **Amazon EC2 Instance Connect** window, run the sample app again and watch the messages in the **MQTT client** in the **AWS IoT console**.

```
cd ~/aws-iot-device-sdk-js-v2/samples/node/pub_sub
```

```
node dist/index.js --topic topic_1 --root-ca ~/certs/Amazon-root-CA-1.pem --cert ~/certs/device.pem.crt --key ~/certs/private.pem.key --endpoint your-iot-endpoint
```

Use your Windows or Linux PC or Mac as an AWS IoT device

In this tutorial, you'll configure a personal computer for use with AWS IoT. These instructions support Windows and Linux PCs and Macs. To accomplish this, you need to install some software on your computer. If you don't want to install software on your computer, you might try [Create a virtual device with Amazon EC2 \(p. 39\)](#), which installs all software on a virtual machine.

In this tutorial, you'll:

- [Set up your personal computer \(p. 47\)](#)
- [Install Git, Python, and the AWS IoT Device SDK for Python \(p. 47\)](#)
- [Set up the policy and run the sample application \(p. 50\)](#)
- [View messages from the sample app in the AWS IoT console \(p. 52\)](#)

Set up your personal computer

To complete this tutorial, you need a Windows or Linux PC or a Mac with a connection to the internet.

Before you continue to the next step, make sure you can open a command line window on your computer. Use **cmd.exe** on a Windows PC. On a Linux PC or a Mac, use **Terminal**.

Install Git, Python, and the AWS IoT Device SDK for Python

In this section, you'll install Python, and the AWS IoT Device SDK for Python on your computer.

Install the latest version of Git and Python

To download and install Git and Python on your computer

1. Check to see if you have Git installed on your computer. Enter this command in the command line.

```
git --version
```

If the command displays the Git version, Git is installed and you can continue to the next step.

If the command displays an error, open <https://git-scm.com/download> and install Git for your computer.

2. Check to see if you have already installed Python. Enter this command in the command line.

```
python -v
```

Note

If this command gives an error: `Python was not found`, it might be because your operating system calls the Python v3.x executable as `Python3`. In that case, replace all instances of `python` with `python3` and continue the remainder of this tutorial.

If the command displays the Python version, Python is already installed. This tutorial requires Python v3.5 or later.

3. If Python is installed, you can skip the rest of the steps in this section. If not, continue.
4. Open <https://www.python.org/downloads/> and download the installer for your computer.
5. If the download didn't automatically start to install, run the downloaded program to install Python.

6. Verify the installation of Python.

```
python -v
```

Confirm that the command displays the Python version. If the Python version isn't displayed, try downloading and installing Python again.

Install the AWS IoT Device SDK for Python

To install the AWS IoT Device SDK for Python on your computer

1. Install v2 of the AWS IoT Device SDK for Python.

```
python3 -m pip install awsiotsdk
```

2. Clone the AWS IoT Device SDK for Python repository into the `aws-iot-device-sdk-python-v2` directory of your home directory. This procedure refers to the base directory for the files you're installing as `home`.

The actual location of the `home` directory depends on your operating system.

Linux/macOS

In macOS and Linux, the `home` directory is `~`.

```
cd ~  
git clone https://github.com/aws/aws-iot-device-sdk-python-v2.git
```

Windows

In Windows, you can find the `home` directory path by running this command in the `cmd` window.

```
echo %USERPROFILE%  
cd %USERPROFILE%  
git clone https://github.com/aws/aws-iot-device-sdk-python-v2.git
```

Note

If you're using Windows PowerShell as opposed to `cmd.exe`, then use the following command.

```
echo $home
```

Prepare to run the sample applications

To prepare your system to run the sample application

- Create the `certs` directory. Into the `certs` directory, copy the private key, device certificate, and root CA certificate files you saved when you created and registered the thing object in [the section called "Create AWS IoT resources" \(p. 34\)](#). The file names of each file in the destination directory should match those in the table.

The commands in the next section assume that your key and certificate files are stored on your device as shown in this table.

Linux/macOS

Run this command to create the `certs` subdirectory that you'll use when you run the sample applications.

```
mkdir ~/certs
```

Into the new subdirectory, copy the files to the destination file paths shown in the following table.

Certificate file names

File	File path
Private key	<code>~/certs/private.pem.key</code>
Device certificate	<code>~/certs/device.pem.crt</code>
Root CA certificate	<code>~/certs/Amazon-root-CA-1.pem</code>

Run this command to list the files in the `certs` directory and compare them to those listed in the table.

```
ls -l ~/certs
```

Windows

Run this command to create the `certs` subdirectory that you'll use when you run the sample applications.

```
mkdir %USERPROFILE%\certs
```

Into the new subdirectory, copy the files to the destination file paths shown in the following table.

Certificate file names

File	File path
Private key	<code>%USERPROFILE%\certs\private.pem.key</code>
Device certificate	<code>%USERPROFILE%\certs\device.pem.crt</code>
Root CA certificate	<code>%USERPROFILE%\certs\Amazon-root-CA-1.pem</code>

Run this command to list the files in the `certs` directory and compare them to those listed in the table.

```
dir %USERPROFILE%\certs
```

Set up the policy and run the sample application

In this section, you'll set up your policy and run the `pubsub.py` sample script found in the `aws-iot-device-sdk-python-v2/samples` directory of the AWS IoT Device SDK for Python. This script shows how your device uses the MQTT library to publish and subscribe to MQTT messages.

The `pubsub.py` sample app subscribes to a topic, `test/topic`, publishes 10 messages to that topic, and displays the messages as they're received from the message broker.

To run the `pubsub.py` sample script, you need the following information:

Application parameter values

Parameter	Where to find the value
<code>your-iot-endpoint</code>	<ol style="list-style-type: none"> In the AWS IoT console, in the left menu, choose Settings. On the Settings page, your endpoint is displayed in the Device data endpoint section.

The `your-iot-endpoint` value has a format of: `endpoint_id-ats.iot.region.amazonaws.com`, for example, `a3qj468EXAMPLE-ats.iot.us-west-2.amazonaws.com`.

Before running the script, make sure your thing's policy provides permissions for the sample script to connect, subscribe, publish, and receive.

To find and review the policy document for a thing resource

1. In the [AWS console](#), in the **Things** list, find the thing resource that represents your device.
2. Choose the **Name** link of the thing resource that represents your device to open the **Thing details** page.
3. In the **Thing details** page, in the **Certificates** tab, choose the certificate that is attached to the thing resource. There should only be one certificate in the list. If there is more than one, choose the certificate whose files are installed on your device and that will be used to connect to AWS IoT.
4. In the **Certificate** details page, in the **Policies** tab, choose the policy that's attached to the certificate. There should only be one. If there is more than one, repeat the next step for each to make sure that at least one policy grants the required access.
5. In the **Policy** overview page, find the JSON editor and choose **Edit policy document** to review and edit the policy document as required.
6. The policy JSON is displayed in the following example. In the "Resource" element, replace `region:account` with your AWS Region and AWS account in each of the Resource values.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish",
                "iot:Receive"
            ],
            "Resource": [
                "arn:aws:iot:region:account:topic/test/topic"
            ]
        },
        {
            ...
        }
    ]
}
```

```
        "Effect": "Allow",
        "Action": [
            "iot:Subscribe"
        ],
        "Resource": [
            "arn:aws:iot:region:account:topicfilter/test/topic"
        ]
    },
    {
        "Effect": "Allow",
        "Action": [
            "iot:Connect"
        ],
        "Resource": [
            "arn:aws:iot:region:account:client/test-*"
        ]
    }
}
```

Linux/macOS

To run the sample script on Linux/macOS

1. In your command line window, navigate to the `~/aws-iot-device-sdk-python-v2/samples/node/pub_sub` directory that the SDK created by using these commands.

```
cd ~/aws-iot-device-sdk-python-v2/samples
```

2. In your command line window, replace `your-iot-endpoint` as indicated and run this command.

```
python3 pubsub.py --endpoint your-iot-endpoint --root-ca ~/certs/Amazon-root-CA-1.pem --cert ~/certs/device.pem.crt --key ~/certs/private.pem.key
```

Windows

To run the sample app on a Windows PC

1. In your command line window, navigate to the `%USERPROFILE%\aws-iot-device-sdk-python-v2\samples` directory that the SDK created and install the sample app by using these commands.

```
cd %USERPROFILE%\aws-iot-device-sdk-python-v2\samples
```

2. In your command line window, replace `your-iot-endpoint` as indicated and run this command.

```
python3 pubsub.py --endpoint your-iot-endpoint --root-ca %USERPROFILE%\certs\Amazon-root-CA-1.pem --cert %USERPROFILE%\certs\device.pem.crt --key %USERPROFILE%\certs\private.pem.key
```

The sample script:

1. Connects to the AWS IoT service for your account.
2. Subscribes to the message topic, **test/topic**, and displays the messages it receives on that topic.

3. Publishes 10 messages to the topic, **test/topic**.

4. Displays output similar to the following:

```
Publish received on topic test/topic
{"message":"Hello world!","sequence":1}
Publish received on topic test/topic
{"message":"Hello world!","sequence":2}
Publish received on topic test/topic
{"message":"Hello world!","sequence":3}
Publish received on topic test/topic
{"message":"Hello world!","sequence":4}
Publish received on topic test/topic
{"message":"Hello world!","sequence":5}
Publish received on topic test/topic
{"message":"Hello world!","sequence":6}
Publish received on topic test/topic
{"message":"Hello world!","sequence":7}
Publish received on topic test/topic
{"message":"Hello world!","sequence":8}
Publish received on topic test/topic
{"message":"Hello world!","sequence":9}
Publish received on topic test/topic
{"message":"Hello world!","sequence":10}
```

If you're having trouble running the sample app, review [the section called "Troubleshooting problems with the sample app" \(p. 60\)](#).

You can also add the `--verbosity Debug` parameter to the command line so the sample app displays detailed messages about what it's doing. That information might help you correct the problem.

View messages from the sample app in the AWS IoT console

You can see the sample app's messages as they pass through the message broker by using the **MQTT client** in the **AWS IoT console**.

To view the MQTT messages published by the sample app

1. Review [View MQTT messages with the AWS IoT MQTT client \(p. 62\)](#). This helps you learn how to use the **MQTT client** in the **AWS IoT console** to view MQTT messages as they pass through the message broker.
2. Open the **MQTT client** in the **AWS IoT console**.
3. Subscribe to the topic, **test/topic**.
4. In your command line window, run the sample app again and watch the messages in the **MQTT client** in the **AWS IoT console**.

Linux/macOS

```
cd ~/aws-iot-device-sdk-python-v2/samples
python3 pubsub.py --topic test/topic --root-ca ~/certs/Amazon-root-CA-1.pem --cert
~/certs/device.pem.crt --key ~/certs/private.pem.key --endpoint your-iot-endpoint
```

Windows

```
cd %USERPROFILE%\aws-iot-device-sdk-python-v2\samples
python3 pubsub.py --topic test/topic --root-ca %USERPROFILE%\certs\Amazon-root-
CA-1.pem --cert %USERPROFILE%\certs\device.pem.crt --key %USERPROFILE%\certs
\private.pem.key --endpoint your-iot-endpoint
```

Connect a Raspberry Pi or another device

In this section, we'll configure a Raspberry Pi for use with AWS IoT. If you have another device that you'd like to connect, the instructions for the Raspberry Pi include references that can help you adapt these instructions to your device.

This normally takes about 20 minutes, but it can take longer if you have many system software upgrades to install.

In this tutorial, you'll:

- [Set up your device \(p. 53\)](#)
- [Install the required tools and libraries for the AWS IoT Device SDK \(p. 54\)](#)
- [Install AWS IoT Device SDK \(p. 54\)](#)
- [Install and run the sample app \(p. 57\)](#)
- [View messages from the sample app in the AWS IoT console \(p. 60\)](#)

Important

Adapting these instructions to other devices and operating systems can be challenging. You'll need to understand your device well enough to be able to interpret these instructions and apply them to your device.

If you encounter difficulties while configuring your device for AWS IoT, we can't offer any assistance beyond the instructions in this section. However, you might try one of the other device options as an alternative, such as [Create a virtual device with Amazon EC2 \(p. 39\)](#) or [Use your Windows or Linux PC or Mac as an AWS IoT device \(p. 47\)](#).

Set up your device

The goal of this step is to collect what you'll need to configure your device such that it can start the operating system (OS), connect to the Internet, and allow you to interact with it at a command line interface.

To complete this tutorial, you need the following:

- An AWS account. If you don't have one, complete the steps described in [Set up your AWS account \(p. 17\)](#) before you continue.
- A [Raspberry Pi 3 Model B](#) or more recent model. This might work on earlier versions of the Raspberry Pi, but they have not been tested.
- [Raspberry Pi OS \(32-bit\)](#) or later. We recommend using the latest version of the Raspberry Pi OS. Earlier versions of the OS might work, but they have not been tested.

To run this example, you do not need to install the desktop with the graphical user interface (GUI); however, if you're new to the Raspberry Pi and your Raspberry Pi hardware supports it, using the desktop with the GUI might be easier.

- An Ethernet or Wi-Fi connection.
- Keyboard, mouse, monitor, cables, power supplies, and other hardware required by your device.

Important

Before you continue to the next step, your device must have its operating system installed, configured, and running. The device must be connected to the Internet and you will need to be able to access the device by using its command line interface. Command line access can be through a directly-connected keyboard, mouse, and monitor, or by using an SSH terminal remote interface.

If you are running an operating system on your Raspberry Pi that has a graphical user interface (GUI), open a terminal window on the device and perform the following instructions in that window. Otherwise,

if you are connecting to your device by using a remote terminal, such as PuTTY, open a remote terminal to your device and use that.

Install the required tools and libraries for the AWS IoT Device SDK

Before you install the AWS IoT Device SDK and sample code, make sure your system is up-to-date and has the required tools and libraries to install the SDKs.

1. Update the operating system and install required libraries

Before you install an AWS IoT Device SDK, run these commands in a terminal window on your device to update the operating system and install the required libraries.

```
sudo apt-get update
```

```
sudo apt-get upgrade
```

```
sudo apt-get install cmake
```

```
sudo apt-get install libssl-dev
```

2. Install Git

If your device's operating system doesn't come with Git installed, you'll need to install it to install the AWS IoT Device SDK for JavaScript.

- Test to see if Git is already installed by running this command.

```
git --version
```

- If the previous command returns the Git version, Git is already installed and you can skip to Step 3.
- If an error is displayed when you run the `git` command, install Git by running this command.

```
sudo apt-get install git
```

- Test again to see if Git is installed by running this command.

```
git --version
```

- If Git is installed, continue to the next section. If not, troubleshoot and correct the error before continuing. You need Git to install the AWS IoT Device SDK for JavaScript.

Install AWS IoT Device SDK

Install the AWS IoT Device SDK.

Python

In this section, you'll install Python, its development tools, and the AWS IoT Device SDK for Python on your device. These instructions are for a Raspberry Pi running the latest Raspberry Pi OS. If you have another device or are using another operating system, you might need to adapt these instructions for your device.

1. Install Python and its development tools

The AWS IoT Device SDK for Python requires Python v3.5 or later to be installed on your Raspberry Pi.

In a terminal window to your device, run these commands.

1. Run this command to determine the version of Python installed on your device.

```
python3 --version
```

If Python is installed, it will display its version.

2. If the version displayed is Python 3.5 or greater, you can skip to Step 2.
3. If the version displayed is less than Python 3.5, you can install the correct version by running this command.

```
sudo apt install python3
```

4. Run this command to confirm that the correct version of Python is now installed.

```
python3 --version
```

2. Test for pip3

In a terminal window to your device, run these commands.

1. Run this command to see if **pip3** is installed.

```
pip3 --version
```

2. If the command returns a version number, **pip3** is installed and you can skip to Step 3.
3. If the previous command returns an error, run this command to install **pip3**.

```
sudo apt install python3-pip
```

4. Run this command to see if **pip3** is installed.

```
pip3 --version
```

3. Install the current AWS IoT Device SDK for Python

Install the AWS IoT Device SDK for Python and download the sample apps to your device.

On your device, run these commands.

```
cd ~  
python3 -m pip install awsiotsdk
```

```
git clone https://github.com/aws/aws-iot-device-sdk-python-v2.git
```

JavaScript

In this section, you'll install Node.js, the npm package manager, and the AWS IoT Device SDK for JavaScript on your device. These instructions are for a Raspberry Pi running the Raspberry Pi OS. If you have another device or are using another operating system, you might need to adapt these instructions for your device.

1. Install the latest version of Node.js

The AWS IoT Device SDK for JavaScript requires Node.js and the npm package manager to be installed on your Raspberry Pi.

- Download the latest version of the Node repository by entering this command.

```
cd ~  
curl -sL https://deb.nodesource.com/setup_12.x | sudo -E bash -
```

- Install Node and npm.

```
sudo apt-get install -y nodejs
```

- Verify the installation of Node.

```
node -v
```

Confirm that the command displays the Node version. This tutorial requires Node v10.0 or later. If the Node version isn't displayed, try downloading the Node repository again.

- Verify the installation of npm.

```
npm -v
```

Confirm that the command displays the npm version. If the npm version isn't displayed, try installing Node and npm again.

- Restart the device.

```
sudo shutdown -r 0
```

Continue after the device restarts.

2. Install the AWS IoT Device SDK for JavaScript

Install the AWS IoT Device SDK for JavaScript on your Raspberry Pi.

- Install aws-crt, the common runtime library.

```
cd ~  
npm install aws-crt
```

- Install v2 of the AWS IoT Device SDK for JavaScript.

```
npm install aws-iot-device-sdk-v2
```

- Clone the AWS IoT Device SDK for JavaScript repository into the `aws-iot-device-sdk-js-v2` directory of your `home` directory. On the Raspberry Pi, the `home` directory is `~/`, which is used as the `home` directory in the following commands. If your device uses a different path for the `home` directory, you must replace `~/` with the correct path for your device in the following commands.

These commands create the `~/aws-iot-device-sdk-js-v2` directory and copy the SDK code into it.

```
cd ~  
git clone https://github.com/aws/aws-iot-device-sdk-js-v2.git
```

- d. Change to the `aws-iot-device-sdk-js-v2` directory that you created in the preceding step and install the SDK.

```
cd ~/aws-iot-device-sdk-js-v2
npm install
```

Install and run the sample app

In this section, you'll install and run the pubsub sample app found in the AWS IoT Device SDK. This app shows how your device uses the MQTT library to publish and subscribe to MQTT messages. The sample app subscribes to a topic, `topic_1`, publishes 10 messages to that topic, and displays the messages as they're received from the message broker.

Install the certificate files

The sample app requires the certificate files that authenticate the device to be installed on the device.

To install the device certificate files for the sample app

1. Create a `certs` subdirectory in your `home` directory by running these commands.

```
cd ~
mkdir certs
```

2. Into the `~/certs` directory, copy the private key, device certificate, and root CA certificate that you created earlier in [the section called "Create AWS IoT resources" \(p. 34\)](#).

How you copy the certificate files to your device depends on the device and operating system and isn't described here. However, if your device supports a graphical user interface (GUI) and has a web browser, you can perform the procedure described in [the section called "Create AWS IoT resources" \(p. 34\)](#) from your device's web browser to download the resulting files directly to your device.

The commands in the next section assume that your key and certificate files are stored on the device as shown in this table.

Certificate file names

File	File path
Root CA certificate	<code>~/certs/Amazon-root-CA-1.pem</code>
Device certificate	<code>~/certs/device.pem.crt</code>
Private key	<code>~/certs/private.pem.key</code>

To run the sample app, you need the following information:

Application parameter values

Parameter	Where to find the value
<code>your-iot-endpoint</code>	In the AWS IoT console , choose Manage , and then choose Things .

Parameter	Where to find the value
	<p>Choose the IoT thing you created for your device, MyIoTThing was the name used earlier, and then choose Interact.</p> <p>On the thing details page, your endpoint is displayed in the HTTPS section.</p>

The *your-iot-endpoint* value has a format of: *endpoint_id*-ats.iot.*region*.amazonaws.com, for example, a3qj468EXAMPLE-ats.iot.us-west-2.amazonaws.com.

Python

To install and run the sample app

1. Navigate to the sample app directory.

```
cd ~/aws-iot-device-sdk-python-v2/samples
```

2. In the command line window, replace *your-iot-endpoint* as indicated and run this command.

```
python3 pubsub.py --topic topic_1 --root-ca ~/certs/Amazon-root-CA-1.pem --cert ~/certs/device.pem.crt --key ~/certs/private.pem.key --endpoint your-iot-endpoint
```

3. Observe that the sample app:

1. Connects to the AWS IoT service for your account.
2. Subscribes to the message topic, **topic_1**, and displays the messages it receives on that topic.
3. Publishes 10 messages to the topic, **topic_1**.
4. Displays output similar to the following:

```
Connecting to a3qj468EXAMPLE-ats.iot.us-west-2.amazonaws.com with client ID
'test-0c8ae2ff-cc87-49d2-a82a-ae7ba1d0ca5a'...
Connected!
Subscribing to topic 'topic_1'...
Subscribed with QoS.AT_LEAST_ONCE
Sending 10 message(s)
Publishing message to topic 'topic_1': Hello World! [1]
Received message from topic 'topic_1': b'Hello World! [1]'
Publishing message to topic 'topic_1': Hello World! [2]
Received message from topic 'topic_1': b'Hello World! [2]'
Publishing message to topic 'topic_1': Hello World! [3]
Received message from topic 'topic_1': b'Hello World! [3]'
Publishing message to topic 'topic_1': Hello World! [4]
Received message from topic 'topic_1': b'Hello World! [4]'
Publishing message to topic 'topic_1': Hello World! [5]
Received message from topic 'topic_1': b'Hello World! [5]'
Publishing message to topic 'topic_1': Hello World! [6]
Received message from topic 'topic_1': b'Hello World! [6]'
Publishing message to topic 'topic_1': Hello World! [7]
Received message from topic 'topic_1': b'Hello World! [7]'
Publishing message to topic 'topic_1': Hello World! [8]
Received message from topic 'topic_1': b'Hello World! [8]'
Publishing message to topic 'topic_1': Hello World! [9]
Received message from topic 'topic_1': b'Hello World! [9]'
Publishing message to topic 'topic_1': Hello World! [10]
```

```
Received message from topic 'topic_1': b'Hello World! [10]'  
10 message(s) received.  
Disconnecting...  
Disconnected!
```

If you're having trouble running the sample app, review [the section called "Troubleshooting problems with the sample app" \(p. 60\)](#).

You can also add the `--verbosity Debug` parameter to the command line so the sample app displays detailed messages about what it's doing. That information might provide you the help you need to correct the problem.

JavaScript

To install and run the sample app

1. In your command line window, navigate to the `~/aws-iot-device-sdk-js-v2/samples/node/pub_sub` directory that the SDK created and install the sample app by using these commands.

```
cd ~/aws-iot-device-sdk-js-v2/samples/node/pub_sub  
npm install
```

2. In the command line window, replace `your-iot-endpoint` as indicated and run this command.

```
node dist/index.js --topic topic_1 --root-ca ~/certs/Amazon-root-CA-1.pem --cert ~/certs/device.pem.crt --key ~/certs/private.pem.key --endpoint your-iot-endpoint
```

3. Observe that the sample app:

1. Connects to the AWS IoT service for your account.
2. Subscribes to the message topic, `topic_1`, and displays the messages it receives on that topic.
3. Publishes 10 messages to the topic, `topic_1`.
4. Displays output similar to the following:

```
Publish received on topic topic_1  
{ "message": "Hello world!", "sequence": 1 }  
Publish received on topic topic_1  
{ "message": "Hello world!", "sequence": 2 }  
Publish received on topic topic_1  
{ "message": "Hello world!", "sequence": 3 }  
Publish received on topic topic_1  
{ "message": "Hello world!", "sequence": 4 }  
Publish received on topic topic_1  
{ "message": "Hello world!", "sequence": 5 }  
Publish received on topic topic_1  
{ "message": "Hello world!", "sequence": 6 }  
Publish received on topic topic_1  
{ "message": "Hello world!", "sequence": 7 }  
Publish received on topic topic_1  
{ "message": "Hello world!", "sequence": 8 }  
Publish received on topic topic_1  
{ "message": "Hello world!", "sequence": 9 }  
Publish received on topic topic_1  
{ "message": "Hello world!", "sequence": 10 }
```

If you're having trouble running the sample app, review [the section called "Troubleshooting problems with the sample app" \(p. 60\)](#).

You can also add the `--verbosity Debug` parameter to the command line so the sample app displays detailed messages about what it's doing. That information might provide you the help you need to correct the problem.

View messages from the sample app in the AWS IoT console

You can see the sample app's messages as they pass through the message broker by using the **MQTT client** in the **AWS IoT console**.

To view the MQTT messages published by the sample app

1. Review [View MQTT messages with the AWS IoT MQTT client \(p. 62\)](#). This helps you learn how to use the **MQTT client** in the **AWS IoT console** to view MQTT messages as they pass through the message broker.
2. Open the **MQTT client** in the **AWS IoT console**.
3. Subscribe to the topic, **topic_1**.
4. In your command line window, run the sample app again and watch the messages in the **MQTT client** in the **AWS IoT console**.

Python

```
cd ~/aws-iot-device-sdk-python-v2/samples
python3 pubsub.py --topic topic_1 --root-ca ~/certs/Amazon-root-CA-1.pem --cert ~/certs/device.pem.crt --key ~/certs/private.pem.key --endpoint your-iot-endpoint
```

JavaScript

```
cd ~/aws-iot-device-sdk-js-v2/samples/node/pub_sub
node dist/index.js --topic topic_1 --root-ca ~/certs/Amazon-root-CA-1.pem --cert ~/certs/device.pem.crt --key ~/certs/private.pem.key --endpoint your-iot-endpoint
```

Troubleshooting problems with the sample app

If you encounter an error when you try to run the sample app, here are some things to check.

Check the certificate

If the certificate is not active, AWS IoT will not accept any connection attempts that use it for authorization. When creating your certificate, it's easy to overlook the **Activate** button. Fortunately, you can activate your certificate from the [AWS IoT console](#).

To check your certificate's activation

1. In the [AWS IoT console](#), in the left menu, choose **Secure**, and then choose **Certificates**.
2. In the list of certificates, find the certificate you created for the exercise and check its status in the **Status** column.

If you don't remember the certificate's name, check for any that are **Inactive** to see if they might be the one you're using.

Choose the certificate in the list to open its detail page. In the detail page, you can see its **Create date** to help you identify the certificate.

3. **To activate an inactive certificate**, from the certificate's detail page, choose **Actions** and then choose **Activate**.

If you found the correct certificate and its active, but you're still having problems running the sample app, check its policy as the next step describes.

You can also try to create a new thing and a new certificate by following the steps in [the section called "Create a thing object" \(p. 37\)](#). If you create a new thing, you will need to give it new thing name and download the new certificate files to your device.

Check the policy attached to the certificate

Policies authorize actions in AWS IoT. If the certificate used to connect to AWS IoT does not have a policy, or does not have a policy that allows it to connect, the connection will be refused, even if the certificate is active.

To check the policies attached to a certificate

1. Find the certificate as described in the previous item and open its details page.
2. In the left menu of the certificate's details page, choose **Policies** to see the policies attached to the certificate
3. If there are no policies attached to the certificate, add one by choosing the **Actions** menu, and then choosing **Attach policy**.

Choose the policy that you created earlier in [the section called "Create AWS IoT resources" \(p. 34\)](#).

4. If there is a policy attached, choose the policy tile to open its details page.

In the details page, review the **Policy document** to make sure it contains the same information as the one you created in [the section called "Create an AWS IoT policy" \(p. 35\)](#).

Check the command line

Make sure you used the correct command line for your system. The commands used on Linux/macOS systems are often different from those used on Windows systems.

Check the endpoint address

Review the command you entered and double-check the endpoint address in your command to the one in your [AWS IoT console](#).

Check the filenames of the certificate files

Compare the filenames in the command you entered to the filenames of the certificate files in the `certs` directory.

Some systems might require the filenames to be in quotes to work correctly.

Check the SDK installation

Make sure that your SDK installation is complete and correct.

If in doubt, reinstall the SDK on your device. In most cases, that's a matter of finding the section of the tutorial titled **Install the AWS IoT Device SDK for *SDK Language*** and following the procedure again.

If you are using the **AWS IoT Device SDK for JavaScript**, remember to install the sample apps before you try to run them. Installing the SDK doesn't automatically install the sample apps. The sample apps must be installed manually after the SDK has been installed.

View MQTT messages with the AWS IoT MQTT client

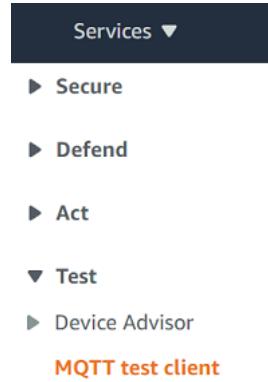
This section describes how to use the AWS IoT MQTT client in the [AWS IoT console](#) to watch the MQTT messages sent and received by AWS IoT. The example used in this section relates to the examples used in [Getting started with AWS IoT Core \(p. 16\)](#); however, you can replace the *topicName* used in the examples with any [topic name or topic filter \(p. 93\)](#) used by your IoT solution.

Devices publish MQTT messages that are identified by [topics \(p. 93\)](#) to communicate their state to AWS IoT, and AWS IoT publishes MQTT messages to inform the devices and apps of changes and events. You can use the MQTT client to subscribe to these topics and watch the messages as they occur. You can also use the MQTT client to publish MQTT messages to subscribed devices and services in your AWS account.

Viewing MQTT messages in the MQTT client

To view MQTT messages in the MQTT client

1. In the [AWS IoT console](#), in the left menu, choose **Test** and then choose **MQTT test client**.



2. In the **Subscribe to a topic** tab, enter the *topicName* to subscribe to the topic on which your device publishes. For the getting started sample app, subscribe to `#`, which subscribes to all message topics.

Continuing with the getting started example, on the **Subscribe to a topic** tab, in the **Topic filter** field, enter `#`, and then choose **Subscribe**.

The screenshot shows the AWS IoT MQTT client interface. At the top, there are two tabs: "Subscribe to a topic" (highlighted in orange) and "Publish to a topic". Below the tabs, there is a "Topic filter" field with the value "#". A link "Additional configuration" is shown below the filter. At the bottom is a large orange "Subscribe" button.

The topic message log page, # opens and # appears in the **Subscriptions** list. If the device that you configured in [the section called “Configure your device” \(p. 38\)](#) is running the example program, you should see the messages it sends to AWS IoT in the # message log. The message log entries will appear below the **Publish** section when messages with the subscribed topic are received by AWS IoT.

Subscriptions	#	Pause	Clear	Export	Edit
#	♡ X				

3. On the # message log page, you can also publish messages to a topic, but you'll need to specify the topic name. You cannot publish to the # topic.

Messages published to subscribed topics appear in the message log as they are received, with the most recent message first.

Troubleshooting MQTT messages

Use the wild card topic filter

If your messages are not showing up in the message log as you expect, try subscribing to a wild card topic filter as described in [Topic filters \(p. 94\)](#). The MQTT multi-level wild card topic filter is the hash or pound sign (#) and can be used as the topic filter in the **Subscription topic** field.

Subscribing to the # topic filter subscribes to every topic received by the message broker. You can narrow the filter down by replacing elements of the topic filter path with a # multi-level wild card character or the '+' single-level wild-card character.

When using wild cards in a topic filter

- The multi-level wild card character must be the last character in the topic filter.
- The topic filter path can have only one single-level wild card character per topic level.

For example:

Topic filter	Displays messages with
#	Any topic name
topic_1/#	A topic name that starts with topic_1/
topic_1/level_2/#	A topic name that starts with topic_1/level_2/
topic_1/+/level_3	A topic name that starts with topic_1/, ends with /level_3, and has one element of any value in between.

For more information on topic filters, see [Topic filters \(p. 94\)](#).

Check for topic name errors

MQTT topic names and topic filters are case sensitive. If, for example, your device is publishing messages to Topic_1 (with a capital T) instead of topic_1, the topic to which you subscribed, its messages would not appear in the MQTT client. Subscribing to the wild card topic filter, however would show that the device is publishing messages and you could see that it was using a topic name that was not the one you expected.

Publishing MQTT messages from the MQTT client

To publish a message to an MQTT topic

1. On the MQTT client page, in the **Publish to a topic** tab, in the **Topic name** field, enter the **topicName** of your message. In this example, use **my/topic**.

Note

Do not use personally identifiable information in topic names, whether using them in the MQTT client or in your system implementation. Topic names can appear in unencrypted communications and reports.

2. In the message payload window, enter the following JSON:

```
{  
    "message": "Hello, world",  
    "clientType": "MQTT client"  
}
```

3. Choose **Publish** to publish your message to AWS IoT.

Note

Make sure you are subscribed to the **my/topic** topic before publishing your message.

Subscribe to a topic **Publish to a topic**

Topic name
The topic name identifies the message. The message payload will be published to this topic with a Quality of Service (QoS) of 0.

X

Message payload

```
{  
  "message": "Hello, world",  
  "clientType": "MQTT client"  
}
```

► Additional configuration

Publish

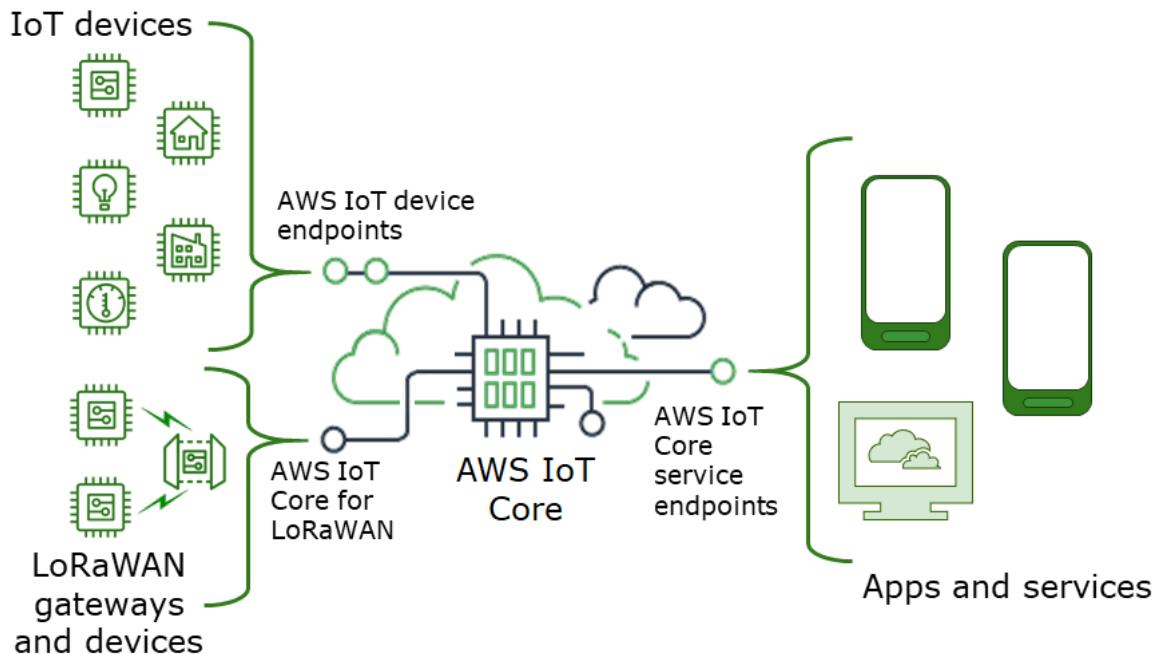
4. In the **Subscriptions** list, choose **my/topic** to see the message. You should see the message appear in the MQTT client below the publish message payload window.

Subscriptions	#	Pause	Clear	Export	Edit	
#	Heart	X				
	▼ my/topic		November 02, 2021, 11:55:22 (UTC-0700)			
	{ "message": "Hello, world", "clientType": "MQTT client" }					

You can publish MQTT messages to other topics by changing the **topicName** in the **Topic name** field and choosing the **Publish** button.

Connecting to AWS IoT Core

AWS IoT Core supports connections with IoT devices, wireless gateways, services, and apps. Devices connect to the AWS IoT Core so they can send data to and receive data from AWS IoT services and other devices. Apps and other services also connect to AWS IoT Core to control and manage the IoT devices and process the data from your IoT solution. This section describes how to choose the best way to connect and communicate with AWS IoT Core for each aspect of your IoT solution.



There are several ways to interact with AWS IoT. Apps and services can use the [AWS IoT Core - control plane endpoints \(p. 66\)](#) and devices can connect to AWS IoT Core by using the [AWS IoT device endpoints \(p. 67\)](#) or [AWS IoT Core for LoRaWAN gateways and devices \(p. 68\)](#).

AWS IoT Core - control plane endpoints

The **AWS IoT Core - control plane** endpoints provide access to functions that control and manage your AWS IoT solution.

- **Endpoints**

The **AWS IoT Core - control plane** endpoints are Region specific and are listed in [AWS IoT Core Endpoints and Quotas](#). The formats of the **AWS IoT Core - control plane** endpoints are as follows.

Endpoint purpose	Endpoint format	Serves
AWS IoT Core - control plane	<code>iot.aws- region.amazonaws.com</code>	AWS IoT Control Plane API

- **SDKs and tools**

The [AWS SDKs](#) provide language-specific support for the AWS IoT Core APIs, and the APIs of other AWS services. The [AWS Mobile SDKs](#) provide app developers with platform-specific support for the AWS IoT Core API, and other AWS services on mobile devices.

The [AWS CLI](#) provides command-line access to the functions provided by the AWS IoT service endpoints. [AWS Tools for PowerShell](#) provides tools to manage AWS services and resources in the PowerShell scripting environment.

- **Authentication**

The service endpoints use IAM users and AWS credentials to authenticate users.

- **Learn more**

For more information and links to SDK references, see [the section called “Connecting to AWS IoT Core service endpoints” \(p. 68\)](#).

AWS IoT device endpoints

The AWS IoT device endpoints support communication between your IoT devices and AWS IoT.

- **Endpoints**

The device endpoints support AWS IoT Core and AWS IoT Device Management functions. They are specific to your AWS account and you can see what they are by using the [describe-endpoint](#) command.

Endpoint purpose	Endpoint format	Serves
AWS IoT Core - data plane	See ??? (p. 74) .	AWS IoT Data Plane API
AWS IoT Device Management - jobs data	See ??? (p. 74) .	AWS IoT Jobs Data Plane API
AWS IoT Device Management - Fleet Hub		
AWS IoT Device Management - secure tunneling	<code>api.tunneling.iot.aws- region.amazonaws.com</code>	AWS IoT Secure Tunneling API

For more information about these endpoints and the functions that they support, see [the section called “AWS IoT device data and service endpoints” \(p. 74\)](#).

- **SDKs**

The [AWS IoT Device SDKs \(p. 76\)](#) provide language-specific support for the Message Queueing Telemetry Transport (MQTT) and WebSocket Secure (WSS) protocols, which devices use to communicate with AWS IoT. [AWS Mobile SDKs \(p. 73\)](#) also provide support for MQTT device communications, AWS IoT APIs, and the APIs of other AWS services on mobile devices.

- **Authentication**

The device endpoints use X.509 certificates or AWS IAM users with credentials to authenticate users.

- **Learn more**

For more information and links to SDK references, see [the section called “AWS IoT Device SDKs” \(p. 76\)](#).

AWS IoT Core for LoRaWAN gateways and devices

AWS IoT Core for LoRaWAN connects wireless gateways and devices to AWS IoT Core.

- **Endpoints**

AWS IoT Core for LoRaWAN manages the gateway connections to account and Region-specific AWS IoT Core endpoints. Gateways can connect to your account's Configuration and Update Server (CUPS) endpoint that AWS IoT Core for LoRaWAN provides.

Endpoint purpose	Endpoint format	Serves
Configuration and Update Server (CUPS)	<i>account-specific-prefix</i> .cups.lorawan. <i>aws-region</i> .amazonaws.com:443	Gateway communication with the Configuration and Update Server provided by AWS IoT Core for LoRaWAN
LoRaWAN Network Server (LNS)	<i>account-specific-prefix</i> .gateway.lorawan. <i>aws-region</i> .amazonaws.com:443	Gateway communication with the LoRaWAN Network Server provided by AWS IoT Core for LoRaWAN

- **SDKs**

The AWS IoT Wireless API that AWS IoT Core for LoRaWAN is built on is supported by the AWS SDK. For more information, see [AWS SDKs and Toolkits](#).

- **Authentication**

AWS IoT Core for LoRaWAN device communications use X.509 certificates to secure communications with AWS IoT.

- **Learn more**

For more information about configuring and connecting wireless devices, see [AWS IoT Core for LoRaWAN \(p. 998\)](#).

Connecting to AWS IoT Core service endpoints

You can access the features of the **AWS IoT Core - control plane** by using the AWS CLI, the AWS SDK for your preferred language, or by calling the REST API directly. We recommend using the AWS CLI or an AWS SDK to interact with AWS IoT Core because they incorporate the best practices for calling AWS services. Calling the REST APIs directly is an option, but you must provide [the necessary security credentials](#) that enable access to the API.

Note

IoT devices should use [AWS IoT Device SDKs \(p. 76\)](#). The Device SDKs are optimized for use on devices, support MQTT communication with AWS IoT, and support the AWS IoT APIs most used by devices. For more information about the Device SDKs and the features they provide, see [AWS IoT Device SDKs \(p. 76\)](#).

Mobile devices should use [AWS Mobile SDKs \(p. 73\)](#). The Mobile SDKs provide support for AWS IoT APIs, MQTT device communications, and the APIs of other AWS services on mobile devices. For more information about the Mobile SDKs and the features they provide, see [AWS Mobile SDKs \(p. 73\)](#).

You can use AWS Amplify tools and resources in web and mobile applications to connect more easily to AWS IoT Core. For more information about connecting to AWS IoT Core by using Amplify, see [Pub Sub Getting Started](#) in the Amplify documentation.

The following sections describe the tools and SDKs that you can use to develop and interact with AWS IoT and other AWS services. For the complete list of AWS tools and development kits that are available to build and manage apps on AWS, see [Tools to Build on AWS](#).

AWS CLI for AWS IoT Core

The AWS CLI provides command-line access to AWS APIs.

- **Installation**

For information about how to install the AWS CLI, see [Installing the AWS CLI](#).

- **Authentication**

The AWS CLI uses credentials from your AWS account.

- **Reference**

For information about the AWS CLI commands for these AWS IoT Core services, see:

- [AWS CLI Command Reference for IoT](#)
- [AWS CLI Command Reference for IoT data](#)
- [AWS CLI Command Reference for IoT jobs data](#)
- [AWS CLI Command Reference for IoT secure tunneling](#)

For tools to manage AWS services and resources in the PowerShell scripting environment, see [AWS Tools for PowerShell](#).

AWS SDKs

With AWS SDKs, your apps and compatible devices can call AWS IoT APIs and the APIs of other AWS services. This section provides links to the AWS SDKs and to the API reference documentation for the APIs of the AWS IoT Core services.

The AWS SDKs support these AWS IoT Core APIs

- [AWS IoT](#)
- [AWS IoT Data Plane](#)
- [AWS IoT Jobs Data Plane](#)
- [AWS IoT Secure Tunneling](#)
- [AWS IoT Wireless](#)

C++

To install the AWS SDK for C++ and use it to connect to AWS IoT:

1. Follow the instructions in [Getting Started Using the AWS SDK for C++](#)

These instructions describe how to:

- Install and build the SDK from source files
- Provide credentials to use the SDK with your AWS account
- Initialize and shutdown the SDK in your app or service
- Create a CMake project to build your app or service

2. Create and run a sample app. For sample apps that use the AWS SDK for C++, see [AWS SDK for C++ + Code Examples](#).

Documentation for the AWS IoT Core services that the AWS SDK for C++ supports

- [Aws::IoT::IoTClient reference documentation](#)
- [Aws::IoTDataPlane::IoTDataPlaneClient reference documentation](#)
- [Aws::IoTJobsDataPlane::IoTJobsDataPlaneClient reference documentation](#)
- [Aws::IoTSecureTunneling::IoTSecureTunnelingClient reference documentation](#)

Go

To install the AWS SDK for Go and use it to connect to AWS IoT:

1. Follow the instructions in [Getting Started with the AWS SDK for Go](#)

These instructions describe how to:

- Install the AWS SDK for Go
- Get access keys for the SDK to access your AWS account
- Import packages into the source code of our apps or services

2. Create and run a sample app. For sample apps that use the AWS SDK for Go, see [AWS SDK for Go Code Examples](#).

Documentation for the AWS IoT Core services that the AWS SDK for Go supports

- [IoT reference documentation](#)
- [IoTDataPlane reference documentation](#)
- [IoTJobsDataPlane reference documentation](#)
- [IoTSecureTunneling reference documentation](#)

Java

To install the AWS SDK for Java and use it to connect to AWS IoT:

1. Follow the instructions in [Getting Started with AWS SDK for Java 2.x](#)

These instructions describe how to:

- Sign up for AWS and Create an IAM User
- Download the SDK
- Set up AWS Credentials and Region
- Use the SDK with Apache Maven
- Use the SDK with Gradle

2. Create and run a sample app using one of the [AWS SDK for Java 2.x Code Examples](#).

3. Review the [SDK API reference documentation](#)

Documentation for the AWS IoT Core services that the AWS SDK for Java supports

- [IoTClient reference documentation](#)
- [IoTDataPlaneClient reference documentation](#)
- [IoTJobsDataPlaneClient reference documentation](#)

- [IoTSecureTunnelingClient reference documentation](#)

JavaScript

To install the AWS SDK for JavaScript and use it to connect to AWS IoT:

1. Follow the instructions in [Setting Up the AWS SDK for JavaScript](#). These instructions apply to using the AWS SDK for JavaScript in the browser and with Node.js. Make sure you follow the directions that apply to your installation.

These instructions describe how to:

- Check for the prerequisites
 - Install the SDK for JavaScript
 - Load the SDK for JavaScript
2. Create and run a sample app to get started with the SDK as the getting started option for your environment describes.
 - Get started with the [AWS SDK for JavaScript in the Browser](#), or
 - Get started with the [AWS SDK for JavaScript in Node.js](#)

Documentation for the AWS IoT Core services that the AWS SDK for JavaScript supports

- [AWS.Iot reference documentation](#)
- [AWS.IotData reference documentation](#)
- [AWS.IotJobsDataPlane reference documentation](#)
- [AWS.IotSecureTunneling reference documentation](#)

.NET

To install the AWS SDK for .NET and use it to connect to AWS IoT:

1. Follow the instructions in [Setting up your AWS SDK for .NET environment](#)
2. Follow the instructions in [Setting up your AWS SDK for .NET project](#)

These instructions describe how to:

- Start a new project
- Obtain and configure AWS credentials
- Install AWS SDK packages

3. Create and run one of the sample programs in [Working with AWS services in the AWS SDK for .NET](#)
4. Review the [SDK API reference documentation](#)

Documentation for the AWS IoT Core services that the AWS SDK for .NET supports

- [Amazon.IoT.Model reference documentation](#)
- [Amazon.IotData.Model reference documentation](#)
- [Amazon.IoTJobsDataPlane.Model reference documentation](#)
- [Amazon.IoTSecureTunneling.Model reference documentation](#)

PHP

To install the AWS SDK for PHP and use it to connect to AWS IoT:

1. Follow the instructions in [Getting Started with the AWS SDK for PHP Version 3](#)

These instructions describe how to:

- Check for the prerequisites
- Install the SDK
- Apply the SDK to a PHP script

2. Create and run a sample app using one of the [AWS SDK for PHP Version 3 Code Examples](#)

Documentation for the AWS IoT Core services that the AWS SDK for PHP supports

- [IoTClient reference documentation](#)
- [IoTDataPlaneClient reference documentation](#)
- [IoTJobsDataPlaneClient reference documentation](#)
- [IoTSecureTunnelingClient reference documentation](#)

Python

To install the AWS SDK for Python (Boto3) and use it to connect to AWS IoT:

1. Follow the instructions in the [AWS SDK for Python \(Boto3\) Quickstart](#)

These instructions describe how to:

- Install the SDK
- Configure the SDK
- Use the SDK in your code

2. Create and run a sample program that uses the AWS SDK for Python (Boto3)

This program displays the account's currently configured logging options. After you install the SDK and configure it for your account, you should be able to run this program.

```
import boto3
import json

# initialize client
iot = boto3.client('iot')

# get current logging levels, format them as JSON, and write them to stdout
response = iot.get_v2_logging_options()
print(json.dumps(response, indent=4))
```

For more information about the function used in this example, see [the section called "Configure AWS IoT logging" \(p. 400\)](#).

Documentation for the AWS IoT Core services that the AWS SDK for Python (Boto3) supports

- [IoT reference documentation](#)
- [IoTDataPlane reference documentation](#)
- [IoTJobsDataPlane reference documentation](#)

- [IoTSecureTunneling reference documentation](#)

Ruby

To install the AWS SDK for Ruby and use it to connect to AWS IoT:

- Follow the instructions in [Getting Started with the AWS SDK for Ruby](#)
These instructions describe how to:
 - Install the SDK
 - Configure the SDK
 - Create and run the [Hello World Tutorial](#)

Documentation for the AWS IoT Core services that the AWS SDK for Ruby supports

- [Aws::IoT::Client reference documentation](#)
- [Aws::IoTDataPlane::Client reference documentation](#)
- [Aws::IoTJobsDataPlane::Client reference documentation](#)
- [Aws::IoTSecureTunneling::Client reference documentation](#)

AWS Mobile SDKs

The AWS Mobile SDKs provide mobile app developers platform-specific support for the APIs of the AWS IoT Core services, IoT device communication using MQTT, and the APIs of other AWS services.

Android

AWS Mobile SDK for Android

The AWS Mobile SDK for Android contains a library, samples, and documentation for developers to build connected mobile applications using AWS. This SDK also includes support for MQTT device communications and calling the APIs of the AWS IoT Core services. For more information, see the following:

- [AWS Mobile SDK for Android on GitHub](#)
- [AWS Mobile SDK for Android Readme](#)
- [AWS Mobile SDK for Android Samples](#)
- [AWS SDK for Android API reference](#)
- [AWSIoTClient Class reference documentation](#)

iOS

AWS Mobile SDK for iOS

The AWS Mobile SDK for iOS is an open-source software development kit, distributed under an Apache Open Source license. The SDK for iOS provides a library, code samples, and documentation to help developers build connected mobile applications using AWS. This SDK also includes support for MQTT device communications and calling the APIs of the AWS IoT Core services. For more information, see the following:

- [AWS Mobile SDK for iOS on GitHub](#)
- [AWS SDK for iOS Readme](#)
- [AWS SDK for iOS Samples](#)

- [AWS IoT Class reference docs in the AWS SDK for iOS](#)

REST APIs of the AWS IoT Core services

The REST APIs of the AWS IoT Core services can be called directly by using HTTP requests.

- **Endpoint URL**

The service endpoints that expose the REST APIs of the AWS IoT Core services vary by Region and are listed in [AWS IoT Core Endpoints and Quotas](#). You must use the endpoint for the Region that has the AWS IoT resources that you want to access, because AWS IoT resources are Region specific.

- **Authentication**

The REST APIs of the AWS IoT Core services use AWS IAM credentials for authentication. For more information, see [Signing AWS API requests](#) in the AWS General Reference.

- **API reference**

For information about the specific functions provided by the REST APIs of the AWS IoT Core services, see:

- [API reference for IoT](#).
- [API reference for IoT data](#).
- [API reference for IoT jobs data](#).
- [API reference for IoT secure tunneling](#).

Connecting devices to AWS IoT

Devices connect to AWS IoT and other services through AWS IoT Core. Through AWS IoT Core, devices send and receive messages using device endpoints that are specific to your account. The [the section called “AWS IoT Device SDKs” \(p. 76\)](#) support device communications using the MQTT and WSS protocols. For more information about the protocols that devices can use, see [the section called “Device communication protocols” \(p. 77\)](#).

The message broker

AWS IoT manages device communication through a message broker. Devices and clients publish messages to the message broker and also subscribe to messages that the message broker publishes. Messages are identified by an application-defined [topic \(p. 93\)](#). When the message broker receives a message published by a device or client, it republishes that message to the devices and clients that have subscribed to the message's topic. The message broker also forwards messages to the AWS IoT [rules \(p. 443\)](#) engine, which can act on the content of the message.

AWS IoT message security

Device connections to AWS IoT use [the section called “X.509 client certificates” \(p. 282\)](#) and [AWS Signature V4](#) for authentication. Device communications are secured by TLS version 1.2 and AWS IoT requires devices to send the [Server Name Indication \(SNI\) extension](#) when they connect. For more information, see [Transport Security in AWS IoT](#).

AWS IoT device data and service endpoints

Each account has several device endpoints that are unique to the account and support specific IoT functions. The AWS IoT device data endpoints support a publish/subscribe protocol that is designed for the communication needs of IoT devices; however, other clients, such as apps and services, can also use

this interface if their application requires the specialized features that these endpoints provide. The AWS IoT device service endpoints support device-centric access to security and management services.

To learn your account's device data endpoint, you can find it in the [Settings](#) page of your AWS IoT Core console.

To learn your account's device endpoint for a specific purpose, including the device data endpoint, use the **describe-endpoint** CLI command shown here, or the `DescribeEndpoint` REST API, and provide the `endpointType` parameter value from the following table.

```
aws iot describe-endpoint --endpoint-type endpointType
```

This command returns an `iot-endpoint` in the following format: `account-specific-prefix.ion.aws-region.amazonaws.com`.

The `DescribeEndpoint` API does not have to be queried every time a new device is connected. The endpoints that you create persist forever and do not change once they are created.

Every customer has an `iot:Data-ATS` and an `iot:Data` endpoint. Each endpoint uses an X.509 certificate to authenticate the client. We strongly recommend that customers use the newer `iot:Data-ATS` endpoint type to avoid issues related to the widespread distrust of Symantec certificate authorities. We provide the `iot:Data` endpoint for devices to retrieve data from old endpoints that use VeriSign certificates for backward compatibility. For more information, see [Server Authentication](#).

AWS IoT endpoints for devices

Endpoint purpose	<code>endpointType</code> value	Description
AWS IoT Core - data plane operations	<code>iot:Data-ATS</code>	Used to send and receive data to and from the message broker, Device Shadow (p. 591) , and Rules Engine (p. 443) components of AWS IoT. <code>iot:Data-ATS</code> returns an ATS signed data endpoint.
AWS IoT Core - data plane operations (legacy)	<code>iot:Data</code>	<code>iot:Data</code> returns a VeriSign signed data endpoint provided for backward compatibility.
AWS IoT Core credential access	<code>iot:CredentialProvider</code>	Used to exchange a device's built-in X.509 certificate for temporary credentials to connect directly with other AWS services. For more information about connecting to other AWS services, see Authorizing Direct Calls to AWS Services (p. 355) .
AWS IoT Device Management - jobs data operations	<code>iot:Jobs</code>	Used to enable devices to interact with the AWS IoT Jobs service using the Jobs Device HTTPS APIs (p. 680) .
AWS IoT operations (preview)	<code>iot:DeviceAdvisor</code>	A test endpoint type used for testing devices with Device Advisor. For more information, see ??? (p. 948).

Endpoint purpose	<i>endpointType</i> value	Description
AWS IoT Core data beta (preview)	iot:Data-Beta	An endpoint type reserved for beta releases. For information about its current use, see ??? (p. 109).

You can also use your own fully-qualified domain name (FQDN), such as [example.com](#), and the associated server certificate to connect devices to AWS IoT by using the section called “Configurable endpoints” (p. 109).

AWS IoT Device SDKs

The AWS IoT Device SDKs help you connect your IoT devices to AWS IoT Core and they support MQTT and MQTT over WSS protocols.

The AWS IoT Device SDKs differ from the AWS SDKs in that the AWS IoT Device SDKs support the specialized communications needs of IoT devices, but don't support all of the services supported by the AWS SDKs. The AWS IoT Device SDKs are compatible with the AWS SDKs that support all of the AWS services; however, they use different authentication methods and connect to different endpoints, which could make using the AWS SDKs impractical on an IoT device.

Mobile devices

The [the section called “AWS Mobile SDKs” \(p. 73\)](#) support both MQTT device communications, some of the AWS IoT service APIs, and the APIs of other AWS services. If you're developing on a supported mobile device, review its SDK to see if it's the best option for developing your IoT solution.

C++

AWS IoT C++ Device SDK

The AWS IoT C++ Device SDK allows developers to build connected applications using AWS and the APIs of the AWS IoT Core services. Specifically, this SDK was designed for devices that are not resource constrained and require advanced features such as message queuing, multi-threading support, and the latest language features. For more information, see the following:

- [AWS IoT Device SDK C++ v2 on GitHub](#)
- [AWS IoT Device SDK C++ v2 Readme](#)
- [AWS IoT Device SDK C++ v2 Samples](#)
- [AWS IoT Device SDK C++ v2 API documentation](#)

Python

AWS IoT Device SDK for Python

The AWS IoT Device SDK for Python makes it possible for developers to write Python scripts to use their devices to access the AWS IoT platform through MQTT or MQTT over the WebSocket Secure (WSS) protocol. By connecting their devices to the APIs of the AWS IoT Core services, users can securely work with the message broker, rules, and Device Shadow service that AWS IoT Core provides and with other AWS services like AWS Lambda, Amazon Kinesis, and Amazon S3, and more.

- [AWS IoT Device SDK for Python v2 on GitHub](#)
- [AWS IoT Device SDK for Python v2 Readme](#)
- [AWS IoT Device SDK for Python v2 Samples](#)
- [AWS IoT Device SDK for Python v2 API documentation](#)

JavaScript

AWS IoT Device SDK for JavaScript

The AWS IoT Device SDK for JavaScript makes it possible for developers to write JavaScript applications that access APIs of the AWS IoT Core using MQTT or MQTT over the WebSocket protocol. It can be used in Node.js environments and browser applications. For more information, see the following:

- [AWS IoT Device SDK for JavaScript v2 on GitHub](#)
- [AWS IoT Device SDK for JavaScript v2 Readme](#)
- [AWS IoT Device SDK for JavaScript v2 Samples](#)
- [AWS IoT Device SDK for JavaScript v2 API documentation](#)

Java

AWS IoT Device SDK for Java

The AWS IoT Device SDK for Java makes it possible for Java developers to access the APIs of the AWS IoT Core through MQTT or MQTT over the WebSocket protocol. The SDK supports the Device Shadow service. You can access shadows by using HTTP methods, including GET, UPDATE, and DELETE. The SDK also supports a simplified shadow access model, which allows developers to exchange data with shadows by using getter and setter methods, without having to serialize or deserialize any JSON documents. For more information, see the following:

- [AWS IoT Device SDK for Java v2 on GitHub](#)
- [AWS IoT Device SDK for Java v2 Readme](#)
- [AWS IoT Device SDK for Java v2 Samples](#)
- [AWS IoT Device SDK for Java v2 API documentation](#)

Embedded C

AWS IoT Device SDK for Embedded C

Important

This SDK is intended for use by experienced embedded-software developers.

The AWS IoT Device SDK for Embedded C (C-SDK) is a collection of C source files under the MIT open source license that can be used in embedded applications to securely connect IoT devices to AWS IoT Core. It includes MQTT, JSON Parser, and AWS IoT Device Shadow libraries and others. It is distributed in source form and intended to be built into customer firmware along with application code, other libraries and, optionally, an RTOS (Real Time Operating System).

The AWS IoT Device SDK for Embedded C is generally targeted at resource constrained devices that require an optimized C language runtime. You can use the SDK on any operating system and host it on any processor type (for example, MCUs and MPUs). If your device has sufficient memory and processing resources available, we recommend that you use one of the other AWS IoT Device and Mobile SDKs, such as the AWS IoT Device SDK for C++, Java, JavaScript, or Python.

For more information, see the following:

- [AWS IoT Device SDK for Embedded C on GitHub](#)
- [AWS IoT Device SDK for Embedded C Readme](#)
- [AWS IoT Device SDK for Embedded C Samples](#)

Device communication protocols

AWS IoT Core supports devices and clients that use the MQTT and the MQTT over WebSocket Secure (WSS) protocols to publish and subscribe to messages, and devices and clients that use the HTTPS protocol to publish messages. All protocols support IPv4 and IPv6. This section describes the different connection options for devices and clients.

TLS v1.2

AWS IoT Core uses [TLS version 1.2](#) to encrypt all communication. Clients must also send the [Server Name Indication \(SNI\)](#) TLS extension. Connection attempts that don't include the SNI are refused. For more information, see [Transport Security in AWS IoT](#).

The [AWS IoT Device SDKs \(p. 76\)](#) support MQTT and MQTT over WSS and support the security requirements of client connections. We recommend using the [AWS IoT Device SDKs \(p. 76\)](#) to connect clients to AWS IoT.

Protocols, port mappings, and authentication

How a device or client connects to the message broker by using a device endpoint depends on the protocol it uses. The following table lists the protocols that the AWS IoT device endpoints support and the authentication methods and ports they use.

Protocols, authentication, and port mappings

Protocol	Operations supported	Authentication	Port	ALPN protocol name
MQTT over WebSocket	Publish, Subscribe	Signature Version 4	443	N/A
MQTT over WebSocket	Publish, Subscribe	Custom authentication	443	N/A
MQTT	Publish, Subscribe	X.509 client certificate	443 [†]	x-amzn-mqtt-ca
MQTT	Publish, Subscribe	X.509 client certificate	8883	N/A
MQTT	Publish, Subscribe	Custom authentication	443 [†]	mqtt
HTTPS	Publish only	Signature Version 4	443	N/A
HTTPS	Publish only	X.509 client certificate	443 [†]	x-amzn-http-ca
HTTPS	Publish only	X.509 client certificate	8443	N/A
HTTPS	Publish only	Custom authentication	443	N/A

Application Layer Protocol Negotiation (ALPN)

[†]Clients that connect on port 443 with X.509 client certificate authentication must implement the [Application Layer Protocol Negotiation \(ALPN\)](#) TLS extension and use the [ALPN protocol name](#) listed in the ALPN ProtocolNameList sent by the client as part of the `ClientHello` message.

On port 443, the [IoT:Data-ATS \(p. 75\)](#) endpoint supports ALPN x-amzn-http-ca HTTP, but the [IoT:Jobs \(p. 75\)](#) endpoint does not.

On port 8443 HTTPS and port 443 MQTT with ALPN x-amzn-mqtt-ca, [custom authentication \(p. 301\)](#) can't be used.

Clients connect to their AWS account's device endpoints. See [the section called "AWS IoT device data and service endpoints" \(p. 74\)](#) for information about how to find your account's device endpoints.

Note

AWS SDKs don't require the entire URL. They only require the endpoint hostname such as the [pubsub.py sample for AWS IoT Device SDK for Python on GitHub](#). Passing the entire URL as provided in the following table can generate an error such as invalid hostname.

Connecting to AWS IoT Core

Protocol	Endpoint or URL
MQTT	<code>iot-endpoint</code>
MQTT over WSS	<code>wss://iot-endpoint/mqtt</code>
HTTPS	<code>https://iot-endpoint/topics</code>

Choosing a protocol for your device communication

For most IoT device communication through the device endpoints, you'll want to use the MQTT or MQTT over WSS protocols; however, the device endpoints also support HTTPS. The following table compares how AWS IoT Core uses the two protocols for device communication.

AWS IoT device protocols side-by-side

Feature	MQTT (p. 80)	HTTPS (p. 90)
Publish/Subscribe support	Publish and subscribe	Publish only
SDK support	AWS Device SDKs (p. 76) support MQTT and WSS protocols	No SDK support, but you can use language-specific methods to make HTTPS requests
Quality of Service support	MQTT QoS levels 0 and 1 (p. 82)	QoS is supported by passing a query string parameter <code>?qos=qos</code> where the value can be 0 or 1. You can add this query string to publish a message with the QoS value you want.
Can receive messages be missed while device was offline	Yes	No
<code>clientId</code> field support	Yes	No
Device disconnection detection	Yes	No
Secure communications	Yes. See Protocols, port mappings, and authentication (p. 78)	Yes. See Protocols, port mappings, and authentication (p. 78)
Topic definitions	Application defined	Application defined
Message data format	Application defined	Application defined
Protocol overhead	Lower	Higher

Feature	MQTT (p. 80)	HTTPS (p. 90)
Power consumption	Lower	Higher

Connection duration limits

HTTPS connections aren't guaranteed to last any longer than the time it takes to receive and respond to requests.

MQTT connection duration depends on the authentication feature that you use. The following table lists the maximum connection duration under ideal conditions for each feature.

MQTT connection duration by authentication feature

Feature	Maximum duration *
X.509 client certificate	1–2 weeks
Custom authentication	1–2 weeks
Signature Version 4	Up to 24 hours

* Not guaranteed

With X.509 certificates and custom authentication, connection duration has no hard limit, but it can be as short as a few minutes. Connection interruptions can occur for various reasons. The following list contains some of the most common reasons.

- Wi-Fi availability interruptions
- Internet service provider (ISP) connection interruptions
- Service patches
- Service deployments
- Service auto scaling
- Unavailable service host
- Load balancer issues and updates
- Client-side errors

Your devices must implement strategies for detecting disconnections and reconnecting. For information about disconnect events and guidance on how to handle them, see [??? \(p. 993\)](#) in [??? \(p. 993\)](#).

MQTT

MQTT is a lightweight and widely adopted messaging protocol that is designed for constrained devices. AWS IoT support for MQTT is based on the [MQTT v3.1.1 specification](#), with some differences. For information about how AWS IoT differs from the MQTT v3.1.1 specification, see [the section called "AWS IoT differences from MQTT version 3.1.1 specification" \(p. 89\)](#).

AWS IoT Core supports device connections that use the MQTT protocol and MQTT over WSS protocol and that are identified by a *client ID*. The [AWS IoT Device SDKs \(p. 76\)](#) support both protocols and are the recommended ways to connect devices to AWS IoT. The AWS IoT Device SDKs support the functions necessary for devices and clients to connect to and access AWS IoT Core services. The Device SDKs support the authentication protocols that the AWS IoT services require and the connection ID requirements that the MQTT protocol and MQTT over WSS protocols require. For information about how

to connect to AWS IoT using the AWS Device SDKs and links to examples of AWS IoT in the supported languages, see [the section called “Connecting with MQTT using the AWS IoT Device SDKs” \(p. 81\)](#). For more information about authentication methods and the port mappings for MQTT messages, see [Protocols, port mappings, and authentication \(p. 78\)](#).

While we recommend using the AWS IoT Device SDKs to connect to AWS IoT, they are not required. If you do not use the AWS IoT Device SDKs, however, you must provide the necessary connection and communication security. Clients must send the [Server Name Indication \(SNI\) TLS extension](#) in the connection request. Connection attempts that don't include the SNI are refused. For more information, see [Transport Security in AWS IoT](#). Clients that use IAM users and AWS credentials to authenticate clients must provide the correct [Signature Version 4](#) authentication.

Connecting with MQTT using the AWS IoT Device SDKs

This section contains links to the AWS IoT Device SDKs and to the source code of sample programs that illustrate how to connect a device to AWS IoT. The sample apps linked here show how to connect to AWS IoT using the MQTT protocol and MQTT over WSS.

C++

Using the AWS IoT C++ Device SDK to connect devices

- [Source code of a sample app that shows an MQTT connection example in C++](#)
- [AWS IoT C++ Device SDK v2 on GitHub](#)

Python

Using the AWS IoT Device SDK for Python to connect devices

- [Source code of a sample app that shows an MQTT connection example in Python](#)
- [AWS IoT Device SDK for Python v2 on GitHub](#)

JavaScript

Using the AWS IoT Device SDK for JavaScript to connect devices

- [Source code of a sample app that shows an MQTT connection example in JavaScript](#)
- [AWS IoT Device SDK for JavaScript v2 on GitHub](#)

Java

Using the AWS IoT Device SDK for Java to connect devices

- [Source code of a sample app that shows an MQTT connection example in Java](#)
- [AWS IoT Device SDK for Java v2 on GitHub](#)

Embedded C

Using the AWS IoT Device SDK for Embedded C to connect devices

Important

This SDK is intended for use by experienced embedded-software developers.

- [Source code of a sample app that shows an MQTT connection example in Embedded C](#)
- [AWS IoT Device SDK for Embedded C on GitHub](#)

MQTT Quality of Service (QoS) options

AWS IoT and the AWS IoT Device SDKs support the [MQTT Quality of Service \(QoS\) levels 0 and 1](#). The MQTT protocol defines a third level of QoS, level 2, but AWS IoT does not support it. Only the MQTT protocol supports the QoS feature. HTTPS does not support QoS.

This table describes how each QoS level affects messages published to and by the message broker.

With a QoS level of...	The message is...	Comments
QoS level 0	Sent zero or more times	This level should be used for messages that are sent over reliable communication links or that can be missed without a problem.
QoS level 1	Sent at least one time, and then repeatedly until a <code>PUBACK</code> response is received	The message is not considered complete until the sender receives a <code>PUBACK</code> response to indicate successful delivery.

Using MQTT persistent sessions

Persistent sessions store a client's subscriptions and messages, with a quality of service (QoS) of 1, that have not been acknowledged by the client. When a disconnected device reconnects to a persistent session, the session resumes, its subscriptions are reinstated, and subscribed messages received prior to the reconnection and that have not been acknowledged by the client are sent to the client.

Creating a persistent session

You create an MQTT persistent session by sending a `CONNECT` message and setting the `cleanSession` flag to 0. If no session exists for the client sending the `CONNECT` message, a new persistent session is created. If a session already exists for the client, the client resumes the existing session.

Operations during a persistent session

Clients use the `sessionPresent` attribute in the connection acknowledged (`CONNACK`) message to determine if a persistent session is present. If `sessionPresent` is 1, a persistent session is present and any stored messages for the client are delivered to the client immediately after the client receives the `CONNACK`, as described in [Message traffic after reconnection to a persistent session \(p. 82\)](#). If `sessionPresent` is 1, there is no need for the client to resubscribe. However, if `sessionPresent` is 0, no persistent session is present and the client must resubscribe to its topic filters.

After the client joins a persistent session, it can publish messages and subscribe to topic filters without any additional flags on each operation.

Message traffic after reconnection to a persistent session

A persistent session represents an ongoing connection between a client and an MQTT message broker. When a client connects to the message broker using a persistent session, the message broker saves all subscriptions that the client makes during the connection. When the client disconnects, the message broker stores unacknowledged QoS 1 messages and new QoS 1 messages published to topics to which the client is subscribed. Messages are stored according to account limit, messages that exceed that limit will be dropped. For more information about persistent message limits, see [AWS IoT Core endpoints and quotas](#). When the client reconnects to its persistent session, all subscriptions are reinstated and all stored messages are sent to the client at a maximum rate of 10 messages per second.

After reconnection, the stored messages are sent to the client, at a rate that is limited to 10 stored messages per second, along with any current message traffic until the [Publish requests per second per connection](#) limit is reached. Because the delivery rate of the stored messages is limited it will take several seconds to deliver all stored messages if a session has more than 10 stored messages to deliver after reconnection.

Ending a persistent session

The following conditions describe how persistent sessions can end.

- When the persistent session expiration time elapses. The persistent session expiration timer starts when the message broker detects that a client has disconnected, either by the client disconnecting or the connection timing out.
- When the client sends a CONNECT message that sets the cleanSession flag to 1.

Note

The stored messages waiting to be sent to the client when a session ends are discarded; however, they are still billed at the standard messaging rate, even though they could not be sent. For more information about message pricing, see [AWS IoT Core Pricing](#). You can configure the expiration time interval.

Reconnection after a persistent session has expired

If a client doesn't reconnect to its persistent session before it expires, the session ends and its stored messages are discarded. When a client reconnects after the session has expired with a cleanSession flag to 0, the service creates a new persistent session. Any subscriptions or messages from the previous session are not available to this session because they were discarded when the previous session expired.

Persistent session message charges

Messages are charged to your AWS account when the message broker sends a message to a client or an offline persistent session. When an offline device with a persistent session reconnects and resumes its session, the stored messages are delivered to the device and charged to your account again. For more information about message pricing, see [AWS IoT Core pricing - Messaging](#).

The default persistent session expiration time of one hour can be increased by using the standard limit increase process. Note that increasing the session expiration time might increase your message charges because the additional time could allow for more messages to be stored for the offline device and those additional messages would be charged to your account at the standard messaging rate. The session expiration time is approximate and a session could persist for up to 30 minutes longer than the account limit; however, a session will not be shorter than the account limit. For more information about session limits, see [AWS Service Quotas](#).

Using MQTT retained messages

AWS IoT Core supports the RETAIN flag described in [MQTT v3.1.1](#). When a client sets the RETAIN flag on an MQTT message that it publishes, AWS IoT Core saves the message. It can then be sent to new subscribers, retrieved by calling the [GetRetainedMessage](#) operation, and viewed in the [AWS IoT console](#). AWS IoT Core stores retained messages for three years after the last time they were updated or accessed. After three years, the messages are deleted.

Examples of using MQTT retained messages

- As an initial configuration message

MQTT retained messages are sent to a client after the client subscribes to a topic. If you want all clients that subscribe to a topic to receive the MQTT retained message immediately after their

subscription, you can publish a configuration message with the RETAIN flag set. Subscribing clients also receive updates to that configuration whenever a new configuration message is published.

- **As a last-known state message**

Devices can set the RETAIN flag on current-state messages so that AWS IoT Core will save them. When applications connect or reconnect, they can subscribe to this topic and get the last reported state immediately after subscribing to the retained message topic. This way they can avoid having to wait until the next message from the device to see the current state.

In this section:

- [Common tasks with MQTT retained messages in AWS IoT Core \(p. 84\)](#)
- [Billing and retained messages \(p. 86\)](#)
- [Comparing MQTT retained messages and MQTT persistent sessions \(p. 86\)](#)
- [MQTT retained messages and AWS IoT Device Shadows \(p. 88\)](#)

Common tasks with MQTT retained messages in AWS IoT Core

AWS IoT Core saves MQTT messages with the RETAIN flag set. These *retained messages* are sent to all clients that have subscribed to the topic, as a normal MQTT message, and they are also stored to be sent to new subscribers to the topic.

MQTT retained messages require specific policy actions to authorize clients to access them. For examples of using retained message policies, see [Retained message policy examples \(p. 346\)](#).

This section describes common operations that involve retained messages.

- **Creating a retained message**

The client determines whether a message is retained when it publishes an MQTT message. Clients can set the RETAIN flag when they publish a message by using a [Device SDK \(p. 1127\)](#). Applications and services can set the RETAIN flag when they use the [Publish action](#) to publish an MQTT message.

Only one message per topic name is retained. A new message with the RETAIN flag set published to a topic replaces any existing retained message that was sent to the topic earlier.

NOTE: You can't publish to a [reserved topic \(p. 95\)](#) with the RETAIN flag set.

- **Subscribing to a retained message topic**

Clients subscribe to retained message topics as they would any other MQTT message topic. Retained messages received by subscribing to a retained message topic have the RETAIN flag set.

Retained messages are deleted from AWS IoT Core when a client publishes a retained message with a 0-byte message payload to the retained message topic. Clients that have subscribed to the retained message topic will also receive the 0-byte message.

Subscribing to a wild card topic filter that includes a retained message topic lets the client receive subsequent messages published to the retained message's topic, but it doesn't deliver the retained message upon subscription.

NOTE: To receive a retained message upon subscription, the topic filter in the subscription request must match the retained message topic exactly.

Retained messages received upon subscribing to a retained message topic have the RETAIN flag set. Retained messages that are received by a subscribing client after subscription, don't.

- **Retrieving a retained message**

Retained messages are delivered to clients automatically when they subscribe to the topic with the retained message. For a client to receive the retained message upon subscription, it must subscribe to the exact topic name of the retained message. Subscribing to a wild card topic filter that includes a retained message topic lets the client receive subsequent messages published to the retained message's topic, but it does not deliver the retained message upon subscription.

Services and apps can list and retrieve retained messages by calling [ListRetainedMessages](#) and [GetRetainedMessage](#).

A client is not prevented from publishing messages to a retained message topic *without* setting the RETAIN flag. This could cause unexpected results, such as the retained message not matching the message received by subscribing to the topic.

- **Listing retained message topics**

You can list retained messages by calling [ListRetainedMessages](#) and the retained messages can be viewed in the [AWS IoT console](#).

- **Getting retained message details**

You can get retained message details by calling [GetRetainedMessage](#) and they can be viewed in the [AWS IoT console](#).

- **Retaining a Will message**

MQTT [Will messages](#) that are created when a device connects can be retained by setting the Will Retain flag in the Connect Flag bits field.

- **Deleting a retained message**

Devices, applications, and services can delete a retained message by publishing a message with the RETAIN flag set and an empty (0-byte) message payload to the topic name of the retained message to delete. Such messages delete the retained message from AWS IoT Core, are sent to clients with a subscription to the topic, but they are not retained by AWS IoT Core.

Retained messages can also be deleted interactively by accessing the retained message in the [AWS IoT console](#). Retained messages that are deleted by using the [AWS IoT console](#) also send a 0-byte message to clients that have subscribed to the retained message's topic.

Retained messages can't be restored after they are deleted. A client would need to publish a new retained message to take the place of the deleted message.

- **Debugging and troubleshooting retained messages**

The [AWS IoT console](#) provides several tools to help you troubleshoot retained messages:

- **The Retained messages page**

The **Retained messages** page in the AWS IoT console provides a paginated list of the retained messages that have been stored by your Account in the current Region. From this page, you can:

- See the details of each retained message, such as the message payload, QoS, the time it was received.
- Update the contents of a retained message.
- Delete a retained message.

- **The MQTT test client**

The **MQTT test client** page in the AWS IoT console can subscribe and publish to MQTT topics. The publish option lets you set the RETAIN flag on the messages that you publish to simulate how your devices might behave.

Some unexpected results might be the result of these aspects of how retained messages are implemented in AWS IoT Core.

- **Retained message limits**

When an account has stored the maximum number of retained messages, AWS IoT Core returns a throttled response to messages published with RETAIN set and payloads greater than 0 bytes until some retained messages are deleted and the retained message count falls below the limit.

- **Retained message delivery order**

The sequence of retained message and subscribed message delivery is not guaranteed.

Billing and retained messages

Publishing messages with the RETAIN flag set from a client, by using AWS IoT console, or by calling [Publish](#) incurs additional messaging charges described in [AWS IoT Core pricing - Messaging](#).

Retrieving retained messages by a client, by using AWS IoT console, or by calling [GetRetainedMessage](#) incurs messaging charges in addition to the normal API usage charges. The additional charges are described in [AWS IoT Core pricing - Messaging](#).

MQTT [Will messages](#) that are published when a device disconnects unexpectedly incur messaging charges described in [AWS IoT Core pricing - Messaging](#).

For more information about messaging costs, see [AWS IoT Core pricing - Messaging](#).

Comparing MQTT retained messages and MQTT persistent sessions

Retained messages and persistent sessions are standard features of MQTT 3.1.1 that make it possible for devices to receive messages that were published while they were offline. Retained messages can be published from persistent sessions. This section describes key aspects of these features and how they work together.

	Retained messages	Persistent sessions	Retained messages in persistent sessions
Key features	<p>Retained messages can be used to configure or notify large groups of devices after they connect.</p> <p>Retained messages can also be used where you want devices to receive only the last message published to a topic after a reconnection.</p>	<p>Persistent sessions are useful for devices that have intermittent connectivity and could miss several important messages.</p> <p>Devices can connect with a persistent session to receive messages sent while they are offline.</p>	Retained messages can be used in both regular and persistent sessions.
Examples	Retained messages can give devices configuration information about their environment when they come online. The initial configuration could include a list of other message topics to which it should subscribe or	Devices that connect over a cellular network with intermittent connectivity could use persistent sessions to avoid missing important messages that are sent while a device is out of network coverage or needs to	The cellular device in the persistent session sample could use a retained message to receive its initial configuration on its initial connection.

	Retained messages	Persistent sessions	Retained messages in persistent sessions
	information about how it should configure its local time zone.	turn off its cellular radio.	
Messages received on initial subscription to a topic	After subscribing to a topic with a retained message, the most recent retained message is received.	After subscribing to a topic without a retained message, no message is received until one is published to the topic.	After subscribing to a topic with a retained message, the most recent retained message is received.
Subscribed topics after reconnection	Without a persistent session, the client must subscribe to topics after reconnection.	Subscribed topics are restored after reconnection.	Subscribed topics are restored after reconnection.
Messages received after reconnection	After subscribing to a topic with a retained message, the most recent retained message is received.	All messages published with a QOS = 1 and subscribed to with a QOS =1 while the device was disconnected are sent after the device reconnects.	All messages published with a QOS = 1 and subscribed to with a QOS =1 that were sent while the device was disconnected are sent after the device reconnects. Updated retained messages from topics to which the client was subscribed are also sent to the client. If more than one retained message was published to a topic while the client was offline, it can receive multiple stored retained messages to that topic after it reconnects.

	Retained messages	Persistent sessions	Retained messages in persistent sessions
Data/session expiration	<p>Retained messages do not expire. They are stored until they are replaced or deleted.</p> <p>For more information about session expirations with persistent sessions, see the section called "Using MQTT persistent sessions" (p. 82).</p>	<p>Persistent sessions expire if the client doesn't reconnect within the timeout period. After a persistent session expires, the client's subscriptions and saved messages that were published with a QOS = 1 and subscribed to with a QOS = 1 while the device was disconnected are deleted.</p> <p>For more information about session expirations with persistent sessions, see the section called "Using MQTT persistent sessions" (p. 82).</p>	<p>Retained messages do not expire. They are stored until they are replaced or deleted even if they are sent from within a persistent session that has expired. After a persistent session expires, the client's subscriptions and saved messages that were published with a QOS = 1 and subscribed to with a QOS = 1 while the device was disconnected are deleted.</p>

For information about persistent sessions, see [the section called "Using MQTT persistent sessions" \(p. 82\)](#).

With Retained Messages, the publishing client determines whether a message should be retained and delivered to a device after it connects, whether it had a previous session or not. The choice to store a message is made by the publisher and the stored message is delivered to all current and future clients that subscribe with a QoS 0 or QoS 1 subscriptions. Retained messages keep only one message on a given topic at a time.

When an account has stored the maximum number of retained messages, AWS IoT Core returns a throttled response to messages published with RETAIN set and payloads greater than 0 bytes until some retained messages are deleted and the retained message count falls below the limit.

MQTT retained messages and AWS IoT Device Shadows

Retained messages and Device Shadows both retain data from a device, but they behave differently and serve different purposes. This section describes their similarities and differences.

	Retained messages	Device Shadows
Message payload has a pre-defined structure or schema	As defined by the implementation. MQTT does not specify a structure or schema for its message payload.	AWS IoT supports a specific data structure.
Updating the message payload generates event messages	Publishing a retained message sends the message to subscribed clients, but doesn't generate additional update messages.	Updating a Device Shadow produces update messages that describe the change .

	Retained messages	Device Shadows
Message updates are numbered	Retained messages are not numbered automatically.	Device Shadow documents have automatic version numbers and timestamps.
Message payload is attached to a thing resource	Retained messages are not attached to a thing resource.	Device Shadows are attached to a thing resource.
Updating individual elements of the message payload	Individual elements of the message can't be changed without updating the entire message payload.	Individual elements of a Device Shadow document can be updated without the need to update the entire Device Shadow document.
Client receives message data upon subscription	Client automatically receives a retained message after it subscribes to a topic with a retained message.	Clients can subscribe to Device Shadow updates, but they must request the current state deliberately.
Indexing and searchability	Retained messages are not indexed for search.	Fleet indexing indexes Device Shadow data for search and aggregation.

Using connectAttributes

ConnectAttributes allow you to specify what attributes you want to use in your connect message in your IAM policies such as `PersistentConnect` and `LastWill`. With ConnectAttributes, you can build policies that don't give devices access to new features by default, which can be helpful if a device is compromised.

`connectAttributes` supports the following features:

`PersistentConnect`

Use the `PersistentConnect` feature to save all subscriptions the client makes during the connection when the connection between the client and broker is interrupted.

`LastWill`

Use the `LastWill` feature to publish a message to the `LastWillTopic` when a client unexpectedly disconnects.

By default, your policy has non-persistent connection and there are no attributes passed for this connection. You must specify a persistent connection in your IAM policy if you want to have one.

For `ConnectAttributes` examples, see [Connect Policy Examples \(p. 325\)](#).

AWS IoT differences from MQTT version 3.1.1 specification

The message broker implementation is based on the [MQTT v3.1.1 specification](#), but it differs from the specification in these ways:

- AWS IoT supports MQTT quality of service (QoS) levels 0 and 1 only. AWS IoT doesn't support publishing or subscribing with QoS level 2. When QoS level 2 is requested, the message broker doesn't send a PUBACK or SUBACK.

- In AWS IoT, subscribing to a topic with QoS level 0 means that a message is delivered zero or more times. A message might be delivered more than once. Messages delivered more than once might be sent with a different packet ID. In these cases, the DUP flag is not set.
- When responding to a connection request, the message broker sends a CONNACK message. This message contains a flag to indicate if the connection is resuming a previous session.
- Before sending additional control packets or a disconnect request, the client must wait for the CONNACK message to be received on their device from the AWS IoT message broker.
- When a client subscribes to a topic, there might be a delay between the time the message broker sends a SUBACK and the time the client starts receiving new matching messages.
- When a client uses the wildcard character # in the topic filter to subscribe to a topic, all strings at and below its level in the topic hierarchy are matched. However, the parent topic is not matched. For example, a subscription to the topic sensor/# receives messages published to the topics sensor/, sensor/temperature, sensor/temperature/room1, but not messages published to sensor. For more information about wildcards, see [Topic filters \(p. 94\)](#).
- The message broker uses the client ID to identify each client. The client ID is passed in from the client to the message broker as part of the MQTT payload. Two clients with the same client ID can't be connected concurrently to the message broker. When a client connects to the message broker using a client ID that another client is using, the new client connection is accepted and the previously connected client is disconnected.
- On rare occasions, the message broker might resend the same logical PUBLISH message with a different packet ID.
- Subscription to topic filters that contain a wildcard character can't receive retained messages. To receive a retained message, the subscribe request must contain a topic filter that matches the retained message topic exactly.
- The message broker doesn't guarantee the order in which messages and ACK are received.

HTTPS

Clients can publish messages by making requests to the REST API using the HTTP 1.0 or 1.1 protocols. For the authentication and port mappings used by HTTP requests, see [Protocols, port mappings, and authentication \(p. 78\)](#).

Note

Unlike MQTT, HTTPS does not support a `clientId` value. So, while a `clientId` is available when using MQTT, it's not available when using HTTPS.

HTTPS message URL

Devices and clients publish their messages by making POST requests to a client-specific endpoint and a topic-specific URL:

```
https://IoT_data_endpoint/topics/url_encoded_topic_name?qos=1"
```

- *IoT_data_endpoint* is the [AWS IoT device data endpoint \(p. 74\)](#). You can find the endpoint in the AWS IoT console on the thing's details page or on the client by using the AWS CLI command:

```
aws iot describe-endpoint --endpoint-type  
    iot:Data-ATS
```

The endpoint should look something like this: a3qjEXAMPLEffp-ats.iot.us-west-2.amazonaws.com

- *url_encoded_topic_name* is the full [topic name \(p. 93\)](#) of the message being sent.

HTTPS message code examples

These are some examples of how to send an HTTPS message to AWS IoT.

Python

```
import requests
import argparse

# define command-line parameters
parser = argparse.ArgumentParser(description="Send messages through an HTTPS connection.")
parser.add_argument('--endpoint', required=True, help="Your AWS IoT data custom endpoint, not including a port. " +
                                         "Ex: \"abcdEXAMPLExyz-ats.iot.us-east-1.amazonaws.com\"")
parser.add_argument('--cert', required=True, help="File path to your client certificate, in PEM format.")
parser.add_argument('--key', required=True, help="File path to your private key, in PEM format.")
parser.add_argument('--topic', required=True, default="test/topic", help="Topic to publish messages to.")
parser.add_argument('--message', default="Hello World!", help="Message to publish. " +
                                         "Specify empty string to publish nothing.")

# parse and load command-line parameter values
args = parser.parse_args()

# create and format values for HTTPS request
publish_url = 'https://' + args.endpoint + ':8443/topics/' + args.topic + '?qos=1'
publish_msg = args.message.encode('utf-8')

# make request
publish = requests.request('POST',
                           publish_url,
                           data=publish_msg,
                           cert=[args.cert, args.key])

# print results
print("Response status: ", str(publish.status_code))
if publish.status_code == 200:
    print("Response body:", publish.text)
```

CURL

You can use [curl](#) from a client or device to send a message to AWS IoT.

To use curl to send a message from an AWS IoT client device

1. Check the [curl](#) version.

- a. On your client, run this command at a command prompt.

```
curl --help
```

In the help text, look for the TLS options. You should see the `--tlsv1.2` option.

- b. If you see the `--tlsv1.2` option, continue.
- c. If you don't see the `--tlsv1.2` option or you get a `command not found` error, you might need to update or install curl on your client or install openssl before you continue.

2. Install the certificates on your client.

Copy the certificate files that you created when you registered your client (thing) in the AWS IoT console. Make sure you have these three certificate files on your client before you continue.

- The CA certificate file ([Amazon-root-CA-1.pem](#) in this example).
 - The client's certificate file ([device.pem.crt](#) in this example).
 - The client's private key file ([private.pem.key](#) in this example).
3. Create the **curl** command line, replacing the replaceable values for those of your account and system.

```
curl --tlsv1.2 \
--cacert Amazon-root-CA-1.pem \
--cert device.pem.crt \
--key private.pem.key \
--request POST \
--data "{ \"message\": \"Hello, world\" }" \
"https://IoT\_data\_endpoint:8443/topics/topic?qos=1"
```

--tlsv1.2

Use TLS 1.2 (SSL).

--cacert [Amazon-root-CA-1.pem](#)

The file name and path, if necessary, of the CA certificate to verify the peer.

--cert [device.pem.crt](#)

The client's certificate file name and path, if necessary.

--key [private.pem.key](#)

The client's private key file name and path, if necessary.

--request POST

The type of HTTP request (in this case, POST).

--data "{ \"message\": \"Hello, world\" }"

The HTTP POST data you want to publish. In this case, it's a JSON string, with the internal quotation marks escaped with the backslash character (\).

"[https://IoT_data_endpoint](#):8443/topics/[topic](#)?qos=1"

The URL of your client's AWS IoT device data endpoint, followed by the HTTPS port, :8443, which is then followed by the keyword, /topics/ and the topic name, [topic](#), in this case. Specify the Quality of Service as the query parameter, ?qos=1.

4. Open the MQTT test client in the AWS IoT console.

Follow the instructions in [View MQTT messages with the AWS IoT MQTT client \(p. 62\)](#) and configure the console to subscribe to messages with the topic name of [topic](#) used in your **curl** command, or use the wildcard topic filter of #.

5. Test the command.

While monitoring the topic in the test client of the AWS IoT console, go to your client and issue the curl command line that you created in step 3. You should see your client's messages in the console.

MQTT topics

MQTT topics identify AWS IoT messages. AWS IoT clients identify the messages they publish by giving the messages topic names. Clients identify the messages to which they want to subscribe (receive) by registering a topic filter with AWS IoT Core. The message broker uses topic names and topic filters to route messages from publishing clients to subscribing clients.

The message broker uses topics to identify messages sent using MQTT and sent using HTTP to the [HTTPS message URL \(p. 90\)](#).

While AWS IoT supports some [reserved system topics \(p. 95\)](#), most MQTT topics are created and managed by you, the system designer. AWS IoT uses topics to identify messages received from publishing clients and select messages to send to subscribing clients, as described in the following sections. Before you create a topic namespace for your system, review the characteristics of MQTT topics to create the hierarchy of topic names that works best for your IoT system.

Topic names

Topic names and topic filters are UTF-8 encoded strings. They can represent a hierarchy of information by using the forward slash (/) character to separate the levels of the hierarchy. For example, this topic name could refer to a temperature sensor in room 1:

- `sensor/temperature/room1`

In this example, there might also be other types of sensors in other rooms with topic names such as:

- `sensor/temperature/room2`
- `sensor/humidity/room1`
- `sensor/humidity/room2`

Note

As you consider topic names for the messages in your system, keep in mind:

- Topic names and topic filters are case sensitive.
- Topic names must not contain personally identifiable information.
- Topic names that begin with a \$ are [reserved topics \(p. 95\)](#) to be used only by AWS IoT Core.
- AWS IoT Core can't send or receive messages between AWS accounts or Regions.

For more information on designing your topic names and namespace, see our whitepaper, [Designing MQTT Topics for AWS IoT Core](#).

For examples of how apps can publish and subscribe to messages, start with [Getting started with AWS IoT Core \(p. 16\)](#) and [AWS IoT Device SDKs, Mobile SDKs, and AWS IoT Device Client \(p. 1127\)](#).

Important

The topic namespace is limited to an AWS account and Region. For example, the `sensor/temp/room1` topic used by an AWS account in one Region is distinct from the `sensor/temp/room1` topic used by the same AWS account in another Region or used by any other AWS account in any Region.

Topic ARN

All topic ARNs (Amazon Resource Names) have the following form:

```
arn:aws:iot:aws-region:AWS-account-ID:topic/Topic
```

For example, `arn:aws:iot:us-west-2:123EXAMPLE456:topic/application/topic/device/sensor` is an ARN for the topic `application/topic/device/sensor`.

Topic filters

Subscribing clients register topic filters with the message broker to specify the message topics that the message broker should send to them. A topic filter can be a single topic name to subscribe to a single topic name or it can include wildcard characters to subscribe to multiple topic names at the same time.

Publishing clients can't use wildcard characters in the topic names they publish.

The following table lists the wildcard characters that can be used in a topic filter.

Topic wildcards

Wildcard character	Matches	Notes
#	All strings at and below its level in the topic hierarchy.	Must be the last character in the topic filter. Must be the only character in its level of the topic hierarchy. Can be used in a topic filter that also contains the + wildcard character.
+	Any string in the level that contains the character.	Must be the only character in its level of the topic hierarchy. Can be used in multiple levels of a topic filter.

Using wildcards with the previous sensor topic name examples:

- A subscription to `sensor/#` receives messages published to `sensor/`, `sensor/temperature`, `sensor/temperature/room1`, but not messages published to `Sensor`.
- A subscription to `sensor/+/room1` receives messages published to `sensor/temperature/room1` and `sensor/humidity/room1`, but not messages sent to `sensor/temperature/room2` or `sensor/humidity/room2`.

Topic filter ARN

All topic filter ARNs (Amazon Resource Names) have the following form:

```
arn:aws:iot:aws-region:AWS-account-ID:topicfilter/TopicFilter
```

For example, `arn:aws:iot:us-west-2:123EXAMPLE456:topicfilter/application/topic/+/_sensor` is an ARN for the topic filter `application/topic/+/_sensor`.

MQTT message payload

The message payload sent in your MQTT messages is not specified by AWS IoT, unless the message payload is for one of the [the section called "Reserved topics" \(p. 95\)](#). Rather, you define the message

payload for your topics to best accommodate your application's needs, within the constraints of the [AWS IoT Core Service Quotas for Protocols](#).

Using a JSON format for your message payload enables the AWS IoT rules engine to parse your messages and apply SQL queries to it. If your application doesn't require the Rules engine to apply SQL queries to your message payloads, you can use any data format that your application requires. For information about limitations and reserved characters in a JSON document used in SQL queries, see [JSON extensions \(p. 584\)](#).

For more information about designing your MQTT topics and their corresponding message payloads, see [Designing MQTT Topics for AWS IoT Core](#).

Reserved topics

Topics that begin with a dollar sign (\$) are reserved for use by AWS IoT. You can subscribe and publish to these reserved topics as they allow; however, you can't create new topics that begin with a dollar sign. Unsupported publish or subscribe operations to reserved topics can result in a terminated connection.

Asset model topics

Topic	Client operations allowed	Description
\$aws/sitewise/asset-models/ <i>assetModelId</i> /assets/ <i>assetId</i> /properties/ <i>propertyId</i>	Subscribe	AWS IoT SiteWise publishes asset property notifications to this topic. For more information, see Interacting with other AWS services in the AWS IoT SiteWise User Guide .

Device Defender topics

These messages support response buffers in Concise Binary Object Representation (CBOR) format and JavaScript Object Notation (JSON), depending on the [*payload-format*](#) of the topic.

<i>payload-format</i>	Response format data type
cbor	Concise Binary Object Representation (CBOR)
json	JavaScript Object Notation (JSON)

For more information, see [Sending metrics from devices \(p. 912\)](#).

Topic	Allowed operations	Description
\$aws/things/ <i>thingName</i> /defender/metrics/ <i>payload-format</i>	Publish	Device Defender agents publish metrics to this topic. For more information, see Sending metrics from devices (p. 912) .
\$aws/things/ <i>thingName</i> /defender/metrics/ <i>payload-format</i> /accepted	Subscribe	AWS IoT publishes to this topic after a Device Defender agent publishes a successful message to \$aws/

Topic	Allowed operations	Description
		things/ <i>thingName</i> /defender/metrics/ <i>payload-format</i> . For more information, see Sending metrics from devices (p. 912) .
\$aws/things/ <i>thingName</i> /defender/metrics/ <i>payload-format</i> /rejected	Subscribe	AWS IoT publishes to this topic after a Device Defender agent publishes an unsuccessful message to \$aws/things/ <i>thingName</i> /defender/metrics/ <i>payload-format</i> . For more information, see Sending metrics from devices (p. 912) .

Event topics

Topic	Client operations allowed	Description
\$aws/events/certificates/registered/ <i>caCertificateId</i>	Subscribe	AWS IoT publishes this message when AWS IoT automatically registers a certificate and when a client presents a certificate with the PENDING_ACTIVATION status. For more information, see the section called "Configure the first connection by a client for automatic registration" (p. 293).
\$aws/events/job/ <i>jobID</i> /canceled	Subscribe	AWS IoT publishes this message when a job is canceled. For more information, see Jobs events (p. 990) .
\$aws/events/job/ <i>jobID</i> /cancellation_in_progress	Subscribe	AWS IoT publishes this message when a job is being canceled. For more information, see Jobs events (p. 990) .
\$aws/events/job/ <i>jobID</i> /completed	Subscribe	AWS IoT publishes this message when a job has completed. For more information, see Jobs events (p. 990) .
\$aws/events/job/ <i>jobID</i> /deleted	Subscribe	AWS IoT publishes this message when a job is

Topic	Client operations allowed	Description
		deleted For more information, see Jobs events (p. 990) .
\$aws/events/job/ <i>jobID</i> /deletion_in_progress	Subscribe	AWS IoT publishes this message when a job is being deleted. For more information, see Jobs events (p. 990) .
\$aws/events/jobExecution/ <i>jobID</i> /canceled	Subscribe	AWS IoT publishes this message when a job execution is canceled. For more information, see Jobs events (p. 990) .
\$aws/events/jobExecution/ <i>jobID</i> /deleted	Subscribe	AWS IoT publishes this message when a job execution is deleted. For more information, see Jobs events (p. 990) .
\$aws/events/jobExecution/ <i>jobID</i> /failed	Subscribe	AWS IoT publishes this message when a job execution has failed. For more information, see Jobs events (p. 990) .
\$aws/events/jobExecution/ <i>jobID</i> /rejected	Subscribe	AWS IoT publishes this message when a job execution was rejected. For more information, see Jobs events (p. 990) .
\$aws/events/jobExecution/ <i>jobID</i> /removed	Subscribe	AWS IoT publishes this message when a job execution was removed. For more information, see Jobs events (p. 990) .
\$aws/events/jobExecution/ <i>jobID</i> /succeeded	Subscribe	AWS IoT publishes this message when a job execution succeeded. For more information, see Jobs events (p. 990) .
\$aws/events/jobExecution/ <i>jobID</i> /timed_out	Subscribe	AWS IoT publishes this message when a job execution timed out. For more information, see Jobs events (p. 990) .

Topic	Client operations allowed	Description
\$aws/events/presence/connected/ <i>clientId</i>	Subscribe	AWS IoT publishes to this topic when an MQTT client with the specified client ID connects to AWS IoT. For more information, see Connect/Disconnect events (p. 993) .
\$aws/events/presence/disconnected/ <i>clientId</i>	Subscribe	AWS IoT publishes to this topic when an MQTT client with the specified client ID disconnects to AWS IoT. For more information, see Connect/Disconnect events (p. 993) .
\$aws/events/subscriptions/subscribed/ <i>clientId</i>	Subscribe	AWS IoT publishes to this topic when an MQTT client with the specified client ID subscribes to an MQTT topic. For more information, see Subscribe/Unsubscribe events (p. 996) .
\$aws/events/subscriptions/unsubscribed/ <i>clientId</i>	Subscribe	AWS IoT publishes to this topic when an MQTT client with the specified client ID unsubscribes to an MQTT topic. For more information, see Subscribe/Unsubscribe events (p. 996) .
\$aws/events/thing/ <i>thingName</i> /created	Subscribe	AWS IoT publishes to this topic when the <i>thingName</i> thing is created. For more information, see the section called "Registry events" (p. 983) .
\$aws/events/thing/ <i>thingName</i> /updated	Subscribe	AWS IoT publishes to this topic when the <i>thingName</i> thing is updated. For more information, see the section called "Registry events" (p. 983) .
\$aws/events/thing/ <i>thingName</i> /deleted	Subscribe	AWS IoT publishes to this topic when the <i>thingName</i> thing is deleted. For more information, see the section called "Registry events" (p. 983) .

Topic	Client operations allowed	Description
\$aws/events/thingGroup/ <i>thingGroupName</i> /created	Subscribe	AWS IoT publishes to this topic when thing group, <i>thingGroupName</i> , is created. For more information, see the section called "Registry events" (p. 983) .
\$aws/events/thingGroup/ <i>thingGroupName</i> /updated	Subscribe	AWS IoT publishes to this topic when thing group, <i>thingGroupName</i> , is updated. For more information, see the section called "Registry events" (p. 983) .
\$aws/events/thingGroup/ <i>thingGroupName</i> /deleted	Subscribe	AWS IoT publishes to this topic when thing group, <i>thingGroupName</i> , is deleted. For more information, see the section called "Registry events" (p. 983) .
\$aws/events/thingType/ <i>thingTypeName</i> /created	Subscribe	AWS IoT publishes to this topic when the <i>thingTypeName</i> thing type is created. For more information, see the section called "Registry events" (p. 983) .
\$aws/events/thingType/ <i>thingTypeName</i> /updated	Subscribe	AWS IoT publishes to this topic when the <i>thingTypeName</i> thing type is updated. For more information, see the section called "Registry events" (p. 983) .
\$aws/events/thingType/ <i>thingTypeName</i> /deleted	Subscribe	AWS IoT publishes to this topic when the <i>thingTypeName</i> thing type is deleted. For more information, see the section called "Registry events" (p. 983) .
\$aws/events/thingTypeAssociation/thing/ <i>thingName</i> / <i>thingTypeName</i>	Subscribe	AWS IoT publishes to this topic when thing, <i>thingName</i> , is associated with or disassociated from thing type, <i>thingTypeName</i> . For more information, see the section called "Registry events" (p. 983) .

Topic	Client operations allowed	Description
\$aws/events/thingGroupMembership/thingGroup/ <i>thingGroupName</i> /thing/ <i>thingName</i> /added	Subscribe	AWS IoT publishes to this topic when thing, <i>thingName</i> , is added to thing group, <i>thingGroupName</i> . For more information, see the section called "Registry events" (p. 983).
\$aws/events/thingGroupMembership/thingGroup/ <i>thingGroupName</i> /thing/ <i>thingName</i> /removed	Subscribe	AWS IoT publishes to this topic when thing, <i>thingName</i> , is removed from thing group, <i>thingGroupName</i> . For more information, see the section called "Registry events" (p. 983).
\$aws/events/thingGroupHierarchy/thingGroup/ <i>parentThingGroupName</i> /childThingGroup/ <i>childThingGroupName</i> /added	Subscribe	AWS IoT publishes to this topic when thing group, <i>childThingGroupName</i> , is added to thing group, <i>parentThingGroupName</i> . For more information, see the section called "Registry events" (p. 983).
\$aws/events/thingGroupHierarchy/thingGroup/ <i>parentThingGroupName</i> /childThingGroup/ <i>childThingGroupName</i> /removed	Subscribe	AWS IoT publishes to this topic when thing group, <i>childThingGroupName</i> , is removed from thing group, <i>parentThingGroupName</i> . For more information, see the section called "Registry events" (p. 983).

Fleet provisioning topics

These messages support response buffers in Concise Binary Object Representation (CBOR) format and JavaScript Object Notation (JSON), depending on the *payload-format* of the topic.

<i>payload-format</i>	Response format data type
cbor	Concise Binary Object Representation (CBOR)
json	JavaScript Object Notation (JSON)

For more information, see [Device provisioning MQTT API \(p. 726\)](#).

Topic	Client operations allowed	Description
\$aws/certificates/create/ <i>payload-format</i>	Publish	Publish to this topic to create a certificate from a certificate signing request (CSR).

Topic	Client operations allowed	Description
\$aws/certificates/create/ <i>payload-format</i> /accepted	Subscribe	AWS IoT publishes to this topic after a successful call to \$aws/certificates/create/ <i>payload-format</i> .
\$aws/certificates/create/ <i>payload-format</i> /rejected	Subscribe	AWS IoT publishes to this topic after an unsuccessful call to \$aws/certificates/create/ <i>payload-format</i> .
\$aws/certificates/create-from-csr/ <i>payload-format</i>	Publish	Publishes to this topic to create a certificate from a CSR.
\$aws/certificates/create-from-csr/ <i>payload-format</i> /accepted	Subscribe	AWS IoT publishes to this topic a successful call to \$aws/certificates/create-from-csr/ <i>payload-format</i> .
\$aws/certificates/create-from-csr/ <i>payload-format</i> /rejected	Subscribe	AWS IoT publishes to this topic an unsuccessful call to \$aws/certificates/create-from-csr/ <i>payload-format</i> .
\$aws/events/presence/connected/ <i>clientId</i>	Subscribe	AWS IoT publishes to this topic when an MQTT client with the specified client ID connects to AWS IoT. For more information, see Connect/Disconnect events (p. 993) .
\$aws/provisioning-templates/ <i>templateName</i> /provision/ <i>payload-format</i>	Publish	Publish to this topic to register a thing.
\$aws/provisioning-templates/ <i>templateName</i> /provision/ <i>payload-format</i> /accepted	Subscribe	AWS IoT publishes to this topic after a successful call to \$aws/provisioning-templates/ <i>templateName</i> /provision/ <i>payload-format</i> .
\$aws/provisioning-templates/ <i>templateName</i> /provision/ <i>payload-format</i> /rejected	Subscribe	AWS IoT publishes to this topic after an unsuccessful call to \$aws/provisioning-templates/ <i>templateName</i> /provision/ <i>payload-format</i> .

Job topics

Note

The client operations noted as **Receive** in this table indicate topics that AWS IoT publishes directly to the client that requested it, whether the client has subscribed to the topic or not. Clients should expect to receive these response messages even if they haven't subscribed to them.

These response messages don't pass through the message broker and they can't be subscribed to by other clients or rules. To subscribe to job activity related messages, use the `notify` and `notify-next` topics.

When subscribing to the job and `jobExecution` event topics for your fleet-monitoring solution, you must first enable [job and job execution events \(p. 982\)](#) to receive any events on the cloud side.

For more information, see [Jobs device MQTT and HTTPS APIs \(p. 680\)](#).

Topic	Client operations allowed	Description
\$aws/things/ <i>thingName</i> /jobs/get	Publish	Devices publish a message to this topic to make a <code>GetPendingJobExecutions</code> request. For more information, see Jobs device MQTT and HTTPS APIs (p. 680) .
\$aws/things/ <i>thingName</i> /jobs/get/accepted	Subscribe, Receive	Devices subscribe to this topic to receive successful responses from a <code>GetPendingJobExecutions</code> request. For more information, see Jobs device MQTT and HTTPS APIs (p. 680) .
\$aws/things/ <i>thingName</i> /jobs/get/rejected	Subscribe, Receive	Devices subscribe to this topic when a <code>GetPendingJobExecutions</code> request is rejected. For more information, see Jobs device MQTT and HTTPS APIs (p. 680) .
\$aws/things/ <i>thingName</i> /jobs/start-next	Publish	Devices publish a message to this topic to make a <code>StartNextPendingJobExecution</code> request. For more information, see Jobs device MQTT and HTTPS APIs (p. 680) .
\$aws/things/ <i>thingName</i> /jobs/start-next/accepted	Subscribe, Receive	Devices subscribe to this topic to receive successful responses to a <code>StartNextPendingJobExecution</code> request. For more information, see Jobs device MQTT and HTTPS APIs (p. 680) .
\$aws/things/ <i>thingName</i> /jobs/start-next/rejected	Subscribe, Receive	Devices subscribe to this topic when a <code>StartNextPendingJobExecution</code> request is rejected. For more information, see Jobs device MQTT and HTTPS APIs (p. 680) .

Topic	Client operations allowed	Description
		device MQTT and HTTPS APIs (p. 680).
\$aws/things/ <i>thingName</i> /jobs/ <i>jobId</i> /get	Publish	Devices publish a message to this topic to make a <code>DescribeJobExecution</code> request. For more information, see Jobs device MQTT and HTTPS APIs (p. 680) .
\$aws/things/ <i>thingName</i> /jobs/ <i>jobId</i> /get/accepted	Subscribe, Receive	Devices subscribe to this topic to receive successful responses to a <code>DescribeJobExecution</code> request. For more information, see Jobs device MQTT and HTTPS APIs (p. 680) .
\$aws/things/ <i>thingName</i> /jobs/ <i>jobId</i> /get/rejected	Subscribe, Receive	Devices subscribe to this topic when a <code>DescribeJobExecution</code> request is rejected. For more information, see Jobs device MQTT and HTTPS APIs (p. 680) .
\$aws/things/ <i>thingName</i> /jobs/ <i>jobId</i> /update	Publish	Devices publish a message to this topic to make an <code>UpdateJobExecution</code> request. For more information, see Jobs device MQTT and HTTPS APIs (p. 680) .
\$aws/things/ <i>thingName</i> /jobs/ <i>jobId</i> /update/accepted	Subscribe, Receive	Devices subscribe to this topic to receive successful responses to an <code>UpdateJobExecution</code> request. For more information, see Jobs device MQTT and HTTPS APIs (p. 680) .
		<p>Note Only the device that publishes to \$aws/things/<i>thingName</i>/jobs/<i>jobId</i>/update receives messages on this topic.</p>

Topic	Client operations allowed	Description
\$aws/things/ <i>thingName</i> /jobs/ <i>jobId</i> /update/rejected	Subscribe, Receive	<p>Devices subscribe to this topic when an <code>UpdateJobExecution</code> request is rejected. For more information, see Jobs device MQTT and HTTPS APIs (p. 680).</p> <p>Note Only the device that publishes to <code>\$aws/things/<i>thingName</i>/jobs/<i>jobId</i>/update</code> receives messages on this topic.</p>
\$aws/things/ <i>thingName</i> /jobs/notify	Subscribe	<p>Devices subscribe to this topic to receive notifications when a job execution is added or removed to the list of pending executions for a thing. For more information, see Jobs device MQTT and HTTPS APIs (p. 680).</p>
\$aws/things/ <i>thingName</i> /jobs/notify-next	Subscribe	<p>Devices subscribe to this topic to receive notifications when the next pending job execution for the thing is changed. For more information, see Jobs device MQTT and HTTPS APIs (p. 680).</p>
\$aws/events/job/ <i>jobId</i> /completed	Subscribe	<p>The Jobs service publishes an event on this topic when a job completes. For more information, see Jobs events (p. 990).</p>
\$aws/events/job/ <i>jobId</i> /canceled	Subscribe	<p>The Jobs service publishes an event on this topic when a job is canceled. For more information, see Jobs events (p. 990).</p>
\$aws/events/job/ <i>jobId</i> /deleted	Subscribe	<p>The Jobs service publishes an event on this topic when a job is deleted. For more information, see Jobs events (p. 990).</p>

Topic	Client operations allowed	Description
\$aws/events/job/ <i>jobId</i> /cancellation_in_progress	Subscribe	The Jobs service publishes an event on this topic when a job cancellation begins. For more information, see Jobs events (p. 990) .
\$aws/events/job/ <i>jobId</i> /deletion_in_progress	Subscribe	The Jobs service publishes an event on this topic when a job deletion begins. For more information, see Jobs events (p. 990) .
\$aws/events/jobExecution/ <i>jobId</i> /succeeded	Subscribe	The Jobs service publishes an event on this topic when job execution succeeds. For more information, see Jobs events (p. 990) .
\$aws/events/jobExecution/ <i>jobId</i> /failed	Subscribe	The Jobs service publishes an event on this topic when a job execution fails. For more information, see Jobs events (p. 990) .
\$aws/events/jobExecution/ <i>jobId</i> /rejected	Subscribe	The Jobs service publishes an event on this topic when a job execution is rejected. For more information, see Jobs events (p. 990) .
\$aws/events/jobExecution/ <i>jobId</i> /canceled	Subscribe	The Jobs service publishes an event on this topic when a job execution is canceled. For more information, see Jobs events (p. 990) .
\$aws/events/jobExecution/ <i>jobId</i> /timed_out	Subscribe	The Jobs service publishes an event on this topic when a job execution times out. For more information, see Jobs events (p. 990) .
\$aws/events/jobExecution/ <i>jobId</i> /removed	Subscribe	The Jobs service publishes an event on this topic when a job execution is removed. For more information, see Jobs events (p. 990) .
\$aws/events/jobExecution/ <i>jobId</i> /deleted	Subscribe	The Jobs service publishes an event on this topic when a job execution is deleted. For more information, see Jobs events (p. 990) .

Rule topics

Topic	Client operations allowed	Description
\$aws/rules/ <i>ruleName</i>	Publish	A device or an application publishes to this topic to trigger rules directly. For more information, see Reducing messaging costs with basic ingest (p. 528) .

Secure tunneling topics

Topic	Client operations allowed	Description
\$aws/things/ <i>thing-name</i> /tunnels/notify	Subscribe	AWS IoT publishes this message for an IoT agent to start a local proxy on the remote device. For more information, see the section called "IoT agent snippet" (p. 701) .

Shadow topics

The topics in this section are used by named and unnamed shadows. The topics used by each differ only in the topic prefix. This table shows the topic prefix used by each shadow type.

ShadowTopicPrefix value	Shadow type
\$aws/things/ <i>thingName</i> /shadow	Unnamed (classic) shadow
\$aws/things/ <i>thingName</i> /shadow/name/ <i>shadowName</i>	Named shadow

To create a complete topic, select the **ShadowTopicPrefix** for the type of shadow to which you want to refer, replace *thingName* and if applicable, *shadowName*, with their corresponding values, and then append that with the topic stub as shown in the following table. Remember that topics are case sensitive.

Topic	Client operations allowed	Description
<i>ShadowTopicPrefix</i> /delete	Publish/Subscribe	A device or an application publishes to this topic to delete a shadow. For more information, see / delete (p. 626) .
<i>ShadowTopicPrefix</i> /delete/accepted	Subscribe	The Device Shadow service sends messages to this topic when a shadow is deleted.

Topic	Client operations allowed	Description
		For more information, see /delete/accepted (p. 626) .
<i>ShadowTopicPrefix/</i> delete/rejected	Subscribe	The Device Shadow service sends messages to this topic when a request to delete a shadow is rejected. For more information, see /delete/rejected (p. 627) .
<i>ShadowTopicPrefix/get</i>	Publish/Subscribe	An application or a thing publishes an empty message to this topic to get a shadow. For more information, see Device Shadow MQTT topics (p. 619) .
<i>ShadowTopicPrefix/get/</i> accepted	Subscribe	The Device Shadow service sends messages to this topic when a request for a shadow is made successfully. For more information, see /get/accepted (p. 621) .
<i>ShadowTopicPrefix/get/</i> rejected	Subscribe	The Device Shadow service sends messages to this topic when a request for a shadow is rejected. For more information, see /get/rejected (p. 621) .
<i>ShadowTopicPrefix/</i> update	Publish/Subscribe	A thing or application publishes to this topic to update a shadow. For more information, see /update (p. 622) .
<i>ShadowTopicPrefix/</i> update/accepted	Subscribe	The Device Shadow service sends messages to this topic when an update is successfully made to a shadow. For more information, see /update/accepted (p. 624) .
<i>ShadowTopicPrefix/</i> update/rejected	Subscribe	The Device Shadow service sends messages to this topic when an update to a shadow is rejected. For more information, see /update/rejected (p. 625) .

Topic	Client operations allowed	Description
<code>ShadowTopicPrefix/</code> update/delta	Subscribe	The Device Shadow service sends messages to this topic when a difference is detected between the reported and desired sections of a shadow. For more information, see /update/delta (p. 623) .
<code>ShadowTopicPrefix/</code> update/documents	Subscribe	AWS IoT publishes a state document to this topic whenever an update to the shadow is successfully performed. For more information, see /update/documents (p. 624) .

MQTT-based file delivery topics

These messages support response buffers in Concise Binary Object Representation (CBOR) format and JavaScript Object Notation (JSON), depending on the `payload-format` of the topic.

<code>payload-format</code>	Response format data type
cbor	Concise Binary Object Representation (CBOR)
json	JavaScript Object Notation (JSON)

Topic	Client operations allowed	Description
<code>\$aws/things/<i>ThingName</i>/streams/<i>StreamId</i>/data/<code>payload-format</code></code>	Subscribe	AWS MQTT-based file delivery publishes to this topic if the "GetStream" request from a device is accepted. The payload contains the stream data. For more information, see Using AWS IoT MQTT-based file delivery in devices (p. 764) .
<code>\$aws/things/<i>ThingName</i>/streams/<i>StreamId</i>/get/<code>payload-format</code></code>	Publish	A device publishes to this topic to perform a "GetStream" request. For more information, see Using AWS IoT MQTT-based file delivery in devices (p. 764) .
<code>\$aws/things/<i>ThingName</i>/streams/<i>StreamId</i>/description/<code>payload-format</code></code>	Subscribe	AWS MQTT-based file delivery publishes to this topic if the "DescribeStream" request from a device is accepted. The payload contains the stream description. For more information, see Using AWS

Topic	Client operations allowed	Description
		IoT MQTT-based file delivery in devices (p. 764).
\$aws/things/ <i>ThingName</i> /streams/ <i>StreamId</i> /describe/ <i>payload-format</i>	Publish	A device publishes to this topic to perform a "DescribeStream" request. For more information, see Using AWS IoT MQTT-based file delivery in devices (p. 764) .
\$aws/things/ <i>ThingName</i> /streams/ <i>StreamId</i> /rejected/ <i>payload-format</i>	Subscribe	AWS MQTT-based file delivery publishes to this topic if a "DescribeStream" or "GetStream" request from a device is rejected. For more information, see Using AWS IoT MQTT-based file delivery in devices (p. 764) .

Reserved topic ARN

All reserved topic ARNs (Amazon Resource Names) have the following form:

```
arn:aws:iot:aws-region:AWS-account-ID:topic/Topic
```

For example, arn:aws:iot:us-west-2:123EXAMPLE456:topic/\$aws/things/thingName/jobs/get/accepted is an ARN for the reserved topic \$aws/things/thingName/jobs/get/accepted.

Configurable endpoints

Note

This feature is not available in GovCloud AWS Regions.

With AWS IoT Core, you can configure and manage the behaviors of your data endpoints by using domain configurations. You can generate multiple AWS IoT Core data endpoints, and also customize data endpoints with your own fully qualified domain names (and associated server certificates) and authorizers. For more information about custom authorizers, see [the section called "Custom authentication" \(p. 301\)](#).

AWS IoT Core uses the server name indication (SNI) TLS extension to apply domain configurations. Devices must use this extension when they connect. They also must pass a server name that is identical to the domain name that you specify in the domain configuration.* To test this service, use the v2 version of the [AWS IoT Device SDKs](#) in GitHub.

Note

If you create multiple data endpoints in your AWS account, they will share AWS IoT Core resources such as MQTT topics, device shadows, and rules.

You can use domain configurations to simplify tasks such as the following.

- Migrate devices to AWS IoT.
- Support heterogeneous device fleets by maintaining separate domain configurations for separate device types.
- Maintain brand identity (for example, through domain name) while migrating application infrastructure to AWS IoT.

You can configure a fully qualified domain name (FQDN) and the associated server certificate too. You can also associate a custom authorizer. For more information, see [Custom authentication \(p. 301\)](#).

Note

AWS IoT uses the server name indication (SNI) TLS extension to apply domain configurations. Devices must use this extension when connecting and pass a server name that is identical to the domain name that is specified in the domain configuration. To test this service, use the v2 version of each [AWS IoT Device SDK in GitHub](#).

Note

When you provide the server certificates for AWS IoT custom domain configuration, the certificates have a maximum of four domain names. For more information, see [AWS IoT Core endpoints and quotas](#).

Topics

- [Creating and configuring AWS-managed domains \(p. 110\)](#)
- [Creating and configuring custom domains \(p. 110\)](#)
- [Managing domain configurations \(p. 113\)](#)

Creating and configuring AWS-managed domains

You create a configurable endpoint on an AWS-managed domain by using the [CreateDomainConfiguration](#) API. A domain configuration for an AWS-managed domain consists of the following:

- `domainConfigurationName` – A user-defined name that identifies the domain configuration.
- Note**
Domain configuration names that start with `IoT:` are reserved for default endpoints and can't be used. Also, this value must be unique to your AWS Region.
- `defaultAuthorizerName` (optional) – The name of the custom authorizer to use on the endpoint.
 - `allowAuthorizerOverride` – A Boolean value that specifies whether devices can override the default authorizer by specifying a different authorizer in the HTTP header of the request. This value is required if a value for `defaultAuthorizerName` is specified.
 - `serviceType` – AWS IoT currently supports only the `DATA` service type. When you specify `DATA`, AWS IoT returns an endpoint with an endpoint type of `iot:Data-ATS`. You can't create a configurable `iot:Data (VeriSign)` endpoint.

The following AWS CLI command creates domain configuration for a Data endpoint.

```
aws iot create-domain-configuration --domain-configuration-name "myDomainConfigurationName"  
--service-type "DATA"
```

Creating and configuring custom domains

Domain configurations let you specify a custom fully qualified domain name (FQDN) to connect to AWS IoT. Custom domains enable you to manage your own server certificates so that you can manage details, such as the root certificate authority (CA) used to sign the certificate, the signature algorithm, the certificate chain depth, and the lifecycle of the certificate.

The workflow to set up a domain configuration with a custom domain consists of the following three stages.

1. [Registering Server Certificates in AWS Certificate Manager \(p. 111\)](#)
2. [Creating a Domain Configuration \(p. 112\)](#)

3. Creating DNS Records (p. 112)

Registering server certificates in AWS certificate manager

Before you create a domain configuration with a custom domain, you must register your server certificate chain in [AWS Certificate Manager \(ACM\)](#). You can use three types of server certificates.

- [ACM Generated Public Certificates \(p. 111\)](#)
- [External Certificates Signed by a Public CA \(p. 111\)](#)
- [External Certificates Signed by a Private CA \(p. 111\)](#)

Note

AWS IoT considers a certificate to be signed by a public CA if it's included in [Mozilla's trusted ca-bundle](#).

Certificate requirements

See [Prerequisites for Importing Certificates](#) for the requirements for importing certificates into ACM. In addition to these requirements, AWS IoT Core adds the following requirements.

- The leaf certificate must include contain the **Extended Key Usage** x509 v3 extension with a value of **serverAuth** (TLS Web Server Authentication). If you request the certificate from ACM, this extension is automatically added.
- The maximum certificate chain depth is 5 certificates.
- The maximum certificate chain size is 16KB.

Using one certificate for multiple domains

If you plan to use one certificate to cover multiple subdomains, use a wildcard domain in the common name (CN) or Subject Alternative Names (SAN) field. For example, use `*.iot.example.com` to cover `dev.iot.example.com`, `qa.iot.example.com`, and `prod.iot.example.com`. Each FQDN requires its own domain configuration, but more than one domain configuration can use the same wildcard value. Either the CN or the SAN must cover the FQDN that you want to use as a custom domain. If SANs are present, the CN is ignored and a SAN must cover the FQDN that you want to use as a custom domain. This coverage can be an exact match or a wildcard match.

The following sections describe how to get each type of certificate. Every certificate resource requires an Amazon Resource Name (ARN) registered with ACM that you use when you create your domain configuration.

ACM-generated public certificates

You can generate a public certificate for your custom domain by using the [RequestCertificate API](#). When you generate a certificate in this way, ACM validates your ownership of the custom domain. For more information, see [Request a Public Certificate](#) in the [AWS Certificate Manager User Guide](#).

External certificates signed by a public CA

If you already have a server certificate that is signed by a public CA (a CA that is included in Mozilla's trusted ca-bundle), you can import the certificate chain directly into ACM by using the [ImportCertificate API](#). To learn more about this task and the prerequisites and certificate format requirements, see [Importing Certificates](#).

External certificates signed by a private CA

If you already have a server certificate that is signed by a private CA or self-signed, you can use the certificate to create your domain configuration, but you also must create an extra public certificate in

ACM to validate ownership of your domain. To do this, register your server certificate chain in ACM using the [ImportCertificate API](#). To learn more about this task and the prerequisites and certificate format requirements, see [Importing Certificates](#).

After you import your certificate to ACM, generate a public certificate for your custom domain by using the [RequestCertificate API](#). When you generate a certificate in this way, ACM validates your ownership of the custom domain. For more information, see [Request a Public Certificate](#). When you create your domain configuration, use this public certificate as your validation certificate.

Creating a domain configuration

You create a configurable endpoint on a custom domain by using the [CreateDomainConfiguration API](#). A domain configuration for a custom domain consists of the following:

- `domainConfigurationName` – A user-defined name that identifies the domain configuration.

Note

Domain configuration names starting with `IoT`: are reserved for default endpoints and can't be used. Also, this value must be unique to your AWS Region.

- `domainName` – The FQDN that your devices use to connect to AWS IoT.

Note

AWS IoT leverages the server name indication (SNI) TLS extension to apply domain configurations. Devices must use this extension when connecting and pass a server name that is identical to the domain name that is specified in the domain configuration.

- `serverCertificateArns` – The ARN of the server certificate chain that you registered with ACM. AWS IoT Core currently supports only one server certificate.
- `validationCertificateArn` – The ARN of the public certificate that you generated in ACM to validate ownership of your custom domain. This argument isn't required if you use a publicly signed or ACM-generated server certificate.
- `defaultAuthorizerName` (optional) – The name of the custom authorizer to use on the endpoint.
- `allowAuthorizerOverride` – A Boolean value that specifies whether devices can override the default authorizer by specifying a different authorizer in the HTTP header of the request. This value is required if a value for `defaultAuthorizerName` is specified.
- `serviceType` – AWS IoT currently supports only the `DATA` service type. When you specify `DATA`, AWS IoT returns an endpoint with an endpoint type of `iot:Data-ATS`.

The following AWS CLI command creates a domain configuration for `iot.example.com`.

```
aws iot create-domain-configuration --domain-configuration-name "myDomainConfigurationName"
--service-type "DATA"
--domain-name "iot.example.com" --server-certificate-arns serverCertARN --validation-
certificate-arn validationCertArn
```

Note

After you create your domain configuration, it might take up to 60 minutes until AWS IoT serves your custom server certificates.

Creating DNS records

After you register your server certificate chain and create your domain configuration, create a DNS record so that your custom domain points to an AWS IoT domain. This record must point to an AWS IoT endpoint of type `iot:Data-ATS`. You can get your endpoint by using the [DescribeEndpoint API](#).

The following AWS CLI command shows how to get your endpoint.

```
aws iot describe-endpoint --endpoint-type iot:Data-ATS
```

After you get your `iot:Data-ATS` endpoint, create a CNAME record from your custom domain to this AWS IoT endpoint. If you create multiple custom domains in the same AWS account, alias them to this same `iot:Data-ATS` endpoint.

Troubleshooting

If you have trouble connecting devices to a custom domain, make sure that AWS IoT Core has accepted and applied your server certificate. You can verify that AWS IoT Core has accepted your certificate by using either the AWS IoT Core console or the AWS CLI.

To use the AWS IoT Core console, navigate to the **Settings** page and select the domain configuration name. In the **Server certificate details** section, check the status and status details. If the certificate is invalid, replace it in ACM with a certificate that meets the [certificate requirements \(p. 111\)](#) listed in the previous section. If the certificate has the same ARN, AWS IoT Core will pick it up and apply it automatically.

To check the certificate status by using the AWS CLI, call the [DescribeDomainConfiguration](#) API and specify your domain configuration name.

Note

If your certificate is invalid, AWS IoT Core will continue to serve the last valid certificate.

You can check which certificate is being served on your endpoint by using the following openssl command.

```
openssl s_client -connect custom-domain-name:8883 -showcerts -servername custom-domain-name
```

Managing domain configurations

You can manage the lifecycles of existing configurations by using the following APIs.

- [ListDomainConfigurations](#)
- [DescribeDomainConfiguration](#)
- [UpdateDomainConfiguration](#)
- [DeleteDomainConfiguration](#)

Viewing domain configurations

Use the [ListDomainConfigurations](#) API to return a paginated list of all domain configurations in your AWS account. You can see the details of a particular domain configuration using the [DescribeDomainConfiguration](#) API. This API takes a single `domainConfigurationName` parameter and returns the details of the specified configuration.

Updating domain configurations

To update the status or the custom authorizer of your domain configuration, use the [UpdateDomainConfiguration](#) API. You can set the status to `ENABLED` or `DISABLED`. If you disable the domain configuration, devices connected to that domain receive an authentication error.

Note

Currently you can't update the server certificate in your domain configuration. To change the certificate of a domain configuration, you must delete and recreate it.

Deleting domain configurations

Before you delete a domain configuration, use the [UpdateDomainConfiguration](#) API to set the status to **DISABLED**. This helps you avoid accidentally deleting the endpoint. After you disable the domain configuration, delete it by using the [DeleteDomainConfiguration](#) API.

Note

You must place AWS-managed domains in **DISABLED** status for 7 days before you can delete them. You can place custom domains in **DISABLED** status and then immediately delete them.

After you delete a domain configuration, AWS IoT Core no longer serves the server certificate associated with that custom domain.

Rotating certificates in custom domains

You may need to periodically replace your server certificate with an updated certificate. The rate at which you do this depends on the validity period of your certificate. If you generated your server certificate by using AWS Certificate Manager (ACM), you can set the certificate to renew automatically. When ACM renews your certificate, AWS IoT Core automatically picks up the new certificate. You don't have to perform any additional action. If you imported your server certificate from a different source, you can rotate it by reimporting it to ACM. For information about reimporting certificates, see [Reimport a certificate](#).

Note

AWS IoT Core only picks up certificate updates under the following conditions.

- The new certificate has the same ARN as the old one.
- The new certificate has the same signing algorithm, common name, or subject alternative name as the old one.

Connecting to AWS IoT FIPS endpoints

AWS IoT provides endpoints that support the [Federal Information Processing Standard \(FIPS\) 140-2](#). FIPS compliant endpoints are different from standard AWS endpoints. To interact with AWS IoT in a FIPS-compliant manner, you must use the endpoints described below with your FIPS compliant client. The AWS IoT console is not FIPS compliant.

The following sections describe how to access the FIPS compliant AWS IoT endpoints by using the REST API, an SDK, or the .AWS CLI.

AWS IoT Core - control plane endpoints

The FIPS compliant **AWS IoT Core - control plane** endpoints that support the [AWS IoT](#) operations and their related [CLI commands](#) are listed in [FIPS Endpoints by Service](#). In [FIPS Endpoints by Service](#), find the **AWS IoT Core - control plane** service, and look up the endpoint for your AWS Region.

To use the FIPS compliant endpoint when you access the [AWS IoT](#) operations, use the AWS SDK or the REST API with the endpoint that is appropriate for your AWS Region.

To use the FIPS compliant endpoint when you run [aws iot CLI commands](#), add the **--endpoint** parameter with the appropriate endpoint for your AWS Region to the command.

AWS IoT Core - data plane endpoints

The FIPS compliant **AWS IoT Core - data plane** endpoints are listed in [FIPS Endpoints by Service](#). In [FIPS Endpoints by Service](#), find the **AWS IoT Core - data plane** service, and look up the endpoint for your AWS Region.

You can use the FIPS compliant endpoint for your AWS Region with a FIPS compliant client by using the AWS IoT Device SDK and providing the endpoint to the SDK's connection function in place of your account's default **AWS IoT Core - data plane** endpoint. The connection function is specific to the AWS IoT Device SDK. For an example of a connection function, see the [Connection function in the AWS IoT Device SDK for Python](#).

Note

AWS IoT doesn't support AWS account-specific **AWS IoT Core - data plane** endpoints that are FIPS-compliant. Service features that require an AWS account-specific endpoint in the [Server Name Indication \(SNI\) \(p. 362\)](#) can't be used. FIPS-compliant **AWS IoT Core - data plane** endpoints can't support [Multi-Account Registration Certificates \(p. 283\)](#), [Custom Domains \(p. 110\)](#), and [Custom Authorizers \(p. 301\)](#).

AWS IoT Device Management - jobs data endpoints

The FIPS compliant **AWS IoT Device Management - jobs data** endpoints are listed in [FIPS Endpoints by Service](#). In [FIPS Endpoints by Service](#), find the **AWS IoT Device Management - jobs data** service, and look up the endpoint for your AWS Region.

To use the FIPS compliant **AWS IoT Device Management - jobs data** endpoint when you run [aws iot-jobs-data CLI commands](#), add the **--endpoint** parameter with the appropriate endpoint for your AWS Region to the command. You can also use the REST API with this endpoint.

You can use the FIPS compliant endpoint for your AWS Region with a FIPS compliant client by using the AWS IoT Device SDK and providing the endpoint to the SDK's connection function in place of your account's default **AWS IoT Device Management - jobs data** endpoint. The connection function is specific to the AWS IoT Device SDK. For an example of a connection function, see the [Connection function in the AWS IoT Device SDK for Python](#).

AWS IoT Device Management - Fleet Hub endpoints

The FIPS compliant **AWS IoT Device Management - Fleet Hub** endpoints to use with [Fleet Hub for AWS IoT Device Management CLI commands](#) are listed in [FIPS Endpoints by Service](#). In [FIPS Endpoints by Service](#), find the **AWS IoT Device Management - Fleet Hub** service, and look up the endpoint for your AWS Region.

To use the FIPS compliant **AWS IoT Device Management - Fleet Hub** endpoint when you run [aws iotfleethub CLI commands](#), add the **--endpoint** parameter with the appropriate endpoint for your AWS Region to the command. You can also use the REST API with this endpoint.

AWS IoT Device Management - secure tunneling endpoints

The FIPS compliant **AWS IoT Device Management - secure tunneling** endpoints for the [AWS IoT secure tunneling API](#) and the corresponding [CLI commands](#) are listed in [FIPS Endpoints by Service](#). In [FIPS Endpoints by Service](#), find the **AWS IoT Device Management - secure tunneling** service, and look up the endpoint for your AWS Region.

To use the FIPS compliant **AWS IoT Device Management - secure tunneling** endpoint when you run [aws iotsecuretunneling CLI commands](#), add the **--endpoint** parameter with the appropriate endpoint for your AWS Region to the command. You can also use the REST API with this endpoint.

AWS IoT tutorials

The AWS IoT tutorials are divided into two learning paths to support two different goals. Choose the best learning path for your goal.

- **You want to build a proof-of-concept to test or demonstrate an AWS IoT solution idea**

To demonstrate common IoT tasks and applications using the AWS IoT Device Client on your devices, follow the [the section called "Building demos with the AWS IoT Device Client" \(p. 116\)](#) learning path. The AWS IoT Device Client provides device software with which you can apply your own cloud resources to demonstrate an end-to-end solution with minimum development.

For information about the AWS IoT Device Client, see the [AWS IoT Device Client](#).

- **You want to learn how to build production software to deploy your solution**

To create your own solution software that meets your specific requirements using an AWS IoT Device SDK, follow the [the section called "Building solutions with the AWS IoT Device SDKs" \(p. 166\)](#) learning path.

For information about the available AWS IoT Device SDKs, see [???](#) (p. 1127). For information about the AWS SDKs, see [Tools to Build on AWS](#).

AWS IoT tutorial learning path options

- [Building demos with the AWS IoT Device Client \(p. 116\)](#)
- [Building solutions with the AWS IoT Device SDKs \(p. 166\)](#)

Building demos with the AWS IoT Device Client

The tutorials in this learning path walk you through the steps to develop demonstration software by using the AWS IoT Device Client. The AWS IoT Device Client provides software that runs on your IoT device to test and demonstrate aspects of an IoT solution that's built on AWS IoT.

The goal of these tutorials is to facilitate exploration and experimentation so you can feel confident that AWS IoT supports your solution before you develop your device software.

What you'll learn in these tutorials:

- How to prepare a Raspberry Pi for use as an IoT device with AWS IoT
- How to demonstrate AWS IoT features by using the AWS IoT Device Client on your device

In this learning path, you'll install the AWS IoT Device Client on your own Raspberry Pi and create the AWS IoT resources in the cloud to demonstrate IoT solution ideas. While the tutorials in this learning path demonstrate features by using a Raspberry Pi, they explain the goals and procedures to help you adapt them to other devices.

Prerequisites to building demos with the AWS IoT Device Client

This section describes what you'll need to have before you start the tutorials in this learning path.

To complete the tutorials in this learning path, you'll need:

- **An AWS account**

You can use your existing AWS account, if you have one, but you might need to add additional roles or permissions to use the AWS IoT features these tutorials use.

If you need to create a new AWS account, see [the section called "Set up your AWS account" \(p. 17\)](#).

- **A Raspberry Pi or compatible IoT device**

The tutorials use a [Raspberry Pi](#) because it comes in different form factors, it's ubiquitous, and it's a relatively inexpensive demonstration device. The tutorials have been tested on the [Raspberry Pi 3 Model B+](#), the [Raspberry Pi 4 Model B](#), and on an Amazon EC2 instance running Ubuntu Server 20.04 LTS (HVM).

Note

The tutorials explain the goals of each step to help you adapt them to IoT hardware that we haven't tried them on; however, they do not specifically describe how to adapt them to other devices.

- **Familiarity with the IoT device's operating system**

The steps in these tutorials assume that you are familiar with using basic Linux commands and operations from the command line interface supported by a Raspberry Pi. If you're not familiar with these operations, you might want to give yourself more time to complete the tutorials.

To complete these tutorials, you should already understand how to:

- Safely perform basic device operations such as assembling and connecting components, connecting the device to required power sources, and installing and removing memory cards.
- Upload and download system software and files to the device. If your device doesn't use a removable storage device, such as a microSD card, you'll need to know how to connect to your device and upload and download system software and files to the device.
- Connect your device to the networks that you plan to use it on.
- Connect to your device from another computer using an SSH terminal or similar program.
- Use a command line interface to create, copy, move, rename, and set the permissions of files and directories on the device.
- Install new programs on the device.
- Transfer files to and from your device using tools such as FTP or SCP.

- **A development and testing environment for your IoT solution**

The tutorials describe the software and hardware required; however, the tutorials assume that you'll be able to perform operations that might not be described explicitly. Examples of such hardware and operations include:

- **A local host computer to download and store files on**

For the Raspberry Pi, this is usually a personal computer or laptop that can read and write to microSD memory cards. The local host computer must:

- Be connected to the Internet.
- Have the [AWS CLI](#) installed and configured.
- Have a web browser that supports the AWS console.
- **A way to connect your local host computer to your device to communicate with it, to enter commands, and to transfer files**

On the Raspberry Pi, this is often done using SSH and SCP from the local host computer.

- **A monitor and keyboard to connect to your IoT device**

These can be helpful, but are not required to complete the tutorials.

- **A way for your local host computer and your IoT devices to connect to the internet**

This could be a cabled or a wireless network connection to a router or gateway that's connected to the internet. The local host must also be able to connect to the Raspberry Pi. This might require them to be on the same local area network. The tutorials can't show you how to set this up for your particular device or device configuration, but they show how you can test this connectivity.

- **Access to your local area network's router to view the connected devices**

To complete the tutorials in this learning path, you'll need to be able to find the IP address of your IoT device.

On a local area network, this can be done by accessing the admin interface of the network router your devices connect to. If you can assign a fixed IP address for your device in the router, you can simplify reconnection after each time the device restarts.

If you have a keyboard and a monitor attached to the device, **ifconfig** can display the device's IP address.

If none of these are an option, you'll need to find a way to identify the device's IP address after each time it restarts.

After you have all your materials, continue to [the section called "Preparing your devices for the AWS IoT Device Client" \(p. 118\)](#).

Tutorials in this learning path

- [Tutorial: Preparing your devices for the AWS IoT Device Client \(p. 118\)](#)
- [Tutorial: Installing and configuring the AWS IoT Device Client \(p. 128\)](#)
- [Tutorial: Demonstrate MQTT message communication with the AWS IoT Device Client \(p. 136\)](#)
- [Tutorial: Demonstrate remote actions \(jobs\) with the AWS IoT Device Client \(p. 150\)](#)
- [Tutorial: Cleaning up after running the AWS IoT Device Client tutorials \(p. 159\)](#)

Tutorial: Preparing your devices for the AWS IoT Device Client

This tutorial walks you through the initialization of your Raspberry Pi to prepare it for the subsequent tutorials in this learning path.

The goal of this tutorial is to install the current version of the device's operating system and make sure that you can communicate with your device in the context of your development environment.

To start this tutorial:

- Have the items listed in [the section called "Prerequisites to building demos with the AWS IoT Device Client" \(p. 116\)](#) available and ready to use.

This tutorial takes about 90 minutes to complete.

When you finish this tutorial:

- Your IoT device will have an up-to-date operating system.
- Your IoT device will have the additional software that it needs for the subsequent tutorials.

- You'll know that your device has connectivity to the internet.
- You will have installed a required certificate on your device.

After you complete this tutorial, the next tutorial prepares your device for the demos that use the AWS IoT Device Client.

Procedures in this tutorial

- [Step 1: Install and update the device's operating system \(p. 119\)](#)
- [Step 2: Install and verify required software on your device \(p. 121\)](#)
- [Step 3: Test your device and save the Amazon CA cert \(p. 124\)](#)

Step 1: Install and update the device's operating system

The procedures in this section describe how to initialize the microSD card that the Raspberry Pi uses for its system drive. The Raspberry Pi's microSD card contains its operating system (OS) software as well as space for its application file storage. If you're not using a Raspberry Pi, follow the device's instructions to install and update the device's operating system software.

After you complete this section, you should be able to start your IoT device and connect to it from the terminal program on your local host computer.

Required equipment:

- Your local development and testing environment
- A Raspberry Pi that or your IoT device, that can connect to the internet
- A microSD memory card with at least 8 GB capacity or sufficient storage for the OS and required software.

Note

When selecting a microSD card for these exercises, choose one that is as large as necessary but, as small as possible.

A small SD card will be faster to back up and update. On the Raspberry Pi, you won't need more than an 8-GB microSD card for these tutorials. If you need more space for your specific application, the smaller image files you save in these tutorials can resize the file system on a larger card to use all the supported space of the card you choose.

Optional equipment:

- A USB keyboard connected to the Raspberry Pi
- An HDMI monitor and cable to connect the monitor to the Raspberry Pi

Procedures in this section:

- [Load the device's operating system onto microSD card \(p. 119\)](#)
- [Start your IoT device with the new operating system \(p. 120\)](#)
- [Connect your local host computer to your device \(p. 121\)](#)

Load the device's operating system onto microSD card

This procedure uses the local host computer to load the device's operating system onto a microSD card.

Note

If your device doesn't use a removable storage medium for its operating system, install the operating system using the procedure for that device and continue to [the section called "Start your IoT device with the new operating system" \(p. 120\)](#).

To install the operating system on your Raspberry Pi

1. On your local host computer, download and unzip the Raspberry Pi operating system image that you want to use. The latest versions are available from <https://www.raspberrypi.com/software/operating-systems/>

Choosing a version of Raspberry Pi OS

This tutorial uses the **Raspberry Pi OS Lite** version because it's the smallest version that supports these the tutorials in this learning path. This version of the Raspberry Pi OS has only a command line interface and doesn't have a graphical user interface. A version of the latest Raspberry Pi OS with a graphical user interface will also work with these tutorials; however, the procedures described in this learning path use only the command line interface to perform operations on the Raspberry Pi.

2. Insert your microSD card into the local host computer.
3. Using an SD card imaging tool, write the unzipped OS image file to the microSD card.
4. After writing the Raspberry Pi OS image to the microSD card:
 - a. Open the BOOT partition on the microSD card in a command line window or file explorer window.
 - b. In the BOOT partition of the microSD card, in the root directory, create an empty file named `ssh` with no file extension and no content. This tells the Raspberry Pi to enable SSH communications the first time it starts.
5. Eject the microSD card and safely remove it from the local host computer.

Your microSD card is ready to [the section called "Start your IoT device with the new operating system" \(p. 120\)](#).

Start your IoT device with the new operating system

This procedure installs the microSD card and starts your Raspberry Pi for the first time using the downloaded operating system.

To start your IoT device with the new operation system

1. With the power disconnected from the device, insert the microSD card from the previous step, [the section called "Load the device's operating system onto microSD card" \(p. 119\)](#), into the Raspberry Pi.
2. Connect the device to a wired network.
3. These tutorials will interact with your Raspberry Pi from your local host computer using an SSH terminal.

If you also want to interact with the device directly, you can:

- a. Connect an HDMI monitor to it to watch the Raspberry Pi's console messages before you can connect the terminal window on your local host computer to your Raspberry Pi.
 - b. Connect a USB keyboard to it if you want to interact directly with the Raspberry Pi.
4. Connect the power to the Raspberry Pi and wait about a minute for it to initialize.

If you have a monitor connected to your Raspberry Pi, you can watch the start-up process on it.

5. Find out your device's IP address:

- If you connected an HDMI monitor to the Raspberry Pi, the IP address appears in the messages displayed on the monitor
- If you have access to the router your Raspberry Pi connects to, you can see its address in the router's admin interface.

After you have your Raspberry Pi's IP address, you're ready to [the section called "Connect your local host computer to your device" \(p. 121\)](#).

Connect your local host computer to your device

This procedure uses the terminal program on your local host computer to connect to your Raspberry Pi and change its default password.

To connect your local host computer to your device

1. On your local host computer, open the SSH terminal program:

- Windows: PuTTY
- Linux/macOS: Terminal

Note

PuTTY isn't installed automatically on Windows. If it's not on your computer, you might need to download and install it.

2. Connect the terminal program to your Raspberry Pi's IP address and log in using its default credentials.

```
username: pi
password: raspberry
```

3. After you log in to your Raspberry Pi, change the password for the pi user.

```
passwd
```

Follow the prompts to change the password.

```
Changing password for pi.
Current password: raspberry
New password: YourNewPassword
Retype new password: YourNewPassword
passwd: password updated successfully
```

After you have the Raspberry Pi's command line prompt in the terminal window and changed the password, you're ready to continue to [the section called "Step 2: Install and verify required software on your device" \(p. 121\)](#).

Step 2: Install and verify required software on your device

The procedures in this section continue from [the previous section \(p. 119\)](#) to bring your Raspberry Pi's operating system up to date and install the software on the Raspberry Pi that will be used in the next section to build and install the AWS IoT Device Client.

After you complete this section, your Raspberry Pi will have an up-to-date operating system, the software required by the tutorials in this learning path, and it will be configured for your location.

Required equipment:

- Your local development and testing environment from [the previous section \(p. 119\)](#)
- The Raspberry Pi that you used in [the previous section \(p. 119\)](#)
- The microSD memory card from [the previous section \(p. 119\)](#)

Note

The Raspberry Pi Model 3+ and Raspberry Pi Model 4 can perform all the commands described in this learning path. If your IoT device can't compile software or run the AWS Command Line Interface, you might need to install the required compilers on your local host computer to build the software and then transfer it to your IoT device. For more information about how to install and build software for your device, see the documentation for your device's software.

Procedures in this section:

- [Update the operating system software \(p. 122\)](#)
- [Install the required applications and libraries \(p. 123\)](#)
- [\(Optional\) Save the microSD card image \(p. 124\)](#)

Update the operating system software

This procedure updates the operating system software.

To update the operating system software on the Raspberry Pi

Perform these steps in the terminal window of your local host computer.

1. Enter these commands to update the system software on your Raspberry Pi.

```
sudo apt-get -y update
sudo apt-get -y upgrade
sudo apt-get -y autoremove
```

2. Update the Raspberry Pi's locale and time zone settings (optional).

Enter this command to update the device's locale and time zone settings.

```
sudo raspi-config
```

- a. To set the device's locale:

- i. In the **Raspberry Pi Software Configuration Tool (raspi-config)** screen, choose option **5**.

5 Localisation Options Configure language and regional settings

Use the **Tab** key to move to **<Select>**, and then press the **space bar**.

- ii. In the localization options menu, choose option **L1**.

L1 Locale Configure language and regional settings

Use the **Tab** key to move to **<Select>**, and then press the **space bar**.

- iii. In the list of locale options, choose the locales that you want to install on your Raspberry Pi by using the arrow keys to scroll and the **space bar** to mark those that you want.

In the United States, **en_US.UTF-8** is a good one to choose.

- iv. After selecting the locales for your device, use the **Tab** key to choose **<OK>**, and then press the **space bar** to display the **Configuring locales** confirmation page.
- b. To set the device's time zone:
 - i. In the **raspi-config** screen, choose option **5**.

5 Localisation Options Configure language and regional settings

Use the **Tab** key to move to **<Select>**, and then press the **space bar**.
 - ii. In the localization options menu, use the arrow key to choose option **L2**:

L2 time zone Configure time zone

Use the **Tab** key to move to **<Select>**, and then press the **space bar**.
 - iii. In the **Configuring tzdata** menu, choose your geographical area from the list.

Use the **Tab** key to move to **<OK>**, and then press the **space bar**.
 - iv. In the list of cities, use the arrow keys to choose a city in your time zone.

To set the time zone, use the **Tab** key to move to **<OK>**, and then press the **space bar**.
- c. When you've finished updating the settings, use the **Tab** key to move to **<Finish>**, and then press the **space bar** to close the **raspi-config** app.

3. Enter this command to restart your Raspberry Pi.

```
sudo shutdown -r 0
```

4. Wait for your Raspberry Pi to restart.
5. After your Raspberry Pi has restarted, reconnect the terminal window on your local host computer to your Raspberry Pi.

Your Raspberry Pi system software is now configured and you're ready to continue to [the section called "Install the required applications and libraries" \(p. 123\)](#).

Install the required applications and libraries

This procedure installs the application software and libraries that the subsequent tutorials use.

If you are using a Raspberry Pi, or if you can compile the required software on your IoT device, perform these steps in the terminal window on your local host computer. If you must compile software for your IoT device on your local host computer, review the software documentation for your IoT device for information about how to do these steps on your device.

To install the application software and libraries on your Raspberry Pi

1. Enter this command to install the application software and libraries.

```
sudo apt-get -y install build-essential libssl-dev cmake unzip git python3-pip
```

2. Enter these commands to confirm that the correct version of the software was installed.

```
gcc --version
cmake --version
openssl version
git --version
```

3. Confirm that these versions of the application software are installed:

- gcc: 9.3.0 or later
- cmake: 3.10.x or later
- OpenSSL: 1.1.1 or later
- git: 2.20.1 or later

If your Raspberry Pi has acceptable versions of the required application software, you're ready to continue to the section called “[\(Optional\) Save the microSD card image](#)” (p. 124).

(Optional) Save the microSD card image

Throughout the tutorials in this learning path, you'll encounter these procedures to save a copy of the Raspberry Pi's microSD card image to a file on your local host computer. While encouraged, they are not required tasks. By saving the microSD card image where suggested, you can skip the procedures that precede the save point in this learning path, which can save time if you find the need to retry something. The consequence of not saving the microSD card image periodically is that you might have to restart the tutorials in the learning path from the beginning if your microSD card is damaged or if you accidentally configure an app or its settings incorrectly.

At this point, your Raspberry Pi's microSD card has an updated OS and the basic application software loaded. You can save the time it took you to complete the preceding steps by saving the contents of the microSD card to a file now. Having the current image of your device's microSD card image lets you start from this point to continue or retry a tutorial or procedure without the need to install and update the software from scratch.

To save the microSD card image to a file

1. Enter this command to shut down the Raspberry Pi.

```
sudo shutdown -h 0
```

2. After the Raspberry Pi shuts down completely, remove its power.
3. Remove the microSD card from the Raspberry Pi.
4. On your local host computer:
 - a. Insert the microSD card.
 - b. Using your SD card imaging tool, save the microSD card's image to a file.
 - c. After the microSD card's image has been saved, eject the card from the local host computer.
5. With the power disconnected from the Raspberry Pi, insert the microSD card into the Raspberry Pi.
6. Apply power to the Raspberry Pi.
7. After waiting about a minute, on the local host computer, reconnect the terminal window on your local host computer that was connected to your Raspberry Pi., and then log in to the Raspberry Pi.

Step 3: Test your device and save the Amazon CA cert

The procedures in this section continue from [the previous section \(p. 121\)](#) to install the AWS Command Line Interface and the Certificate Authority certificate used to authenticate your connections with AWS IoT Core.

After you complete this section, you'll know that your Raspberry Pi has the necessary system software to install the AWS IoT Device Client and that it has a working connection to the internet.

Required equipment:

- Your local development and testing environment from [the previous section \(p. 121\)](#)

- The Raspberry Pi that you used in [the previous section \(p. 121\)](#)
- The microSD memory card from [the previous section \(p. 121\)](#)

Procedures in this section:

- [Install the AWS Command Line Interface \(p. 125\)](#)
- [Configure your AWS account credentials \(p. 125\)](#)
- [Download the Amazon Root CA certificate \(p. 126\)](#)
- [\(Optional\) Save the microSD card image \(p. 127\)](#)

Install the AWS Command Line Interface

This procedure installs the AWS CLI onto your Raspberry Pi.

If you are using a Raspberry Pi or if you can compile software on your IoT device, perform these steps in the terminal window on your local host computer. If you must compile software for your IoT device on your local host computer, review the software documentation for your IoT device for information about the libraries it requires.

To install the AWS CLI on your Raspberry Pi

1. Run these commands to download and install the AWS CLI.

```
export PATH=$PATH:-/.local/bin # configures the path to include the directory with the
AWS CLI
git clone https://github.com/aws/aws-cli.git # download the AWS CLI code from GitHub
cd aws-cli && git checkout v2 # go to the directory with the repo and checkout version
2
pip3 install -r requirements.txt # install the prerequisite software
```

2. Run this command to install the AWS CLI. This command can take up to 15 minutes to complete.

```
pip3 install . # install the AWS CLI
```

3. Run this command to confirm that the correct version of the AWS CLI was installed.

```
aws --version
```

The version of the AWS CLI should be 2.2 or later.

If the AWS CLI displayed its current version, you're ready to continue to [the section called "Configure your AWS account credentials" \(p. 125\)](#).

Configure your AWS account credentials

In this procedure, you'll obtain AWS account credentials and add them for use on your Raspberry Pi.

To add your AWS account credentials to your device

1. Obtain an **Access Key ID** and **Secret Access Key** from your AWS account to authenticate the AWS CLI on your device.

If you're new to AWS IAM, <https://aws.amazon.com/premiumsupport/knowledge-center/create-access-key/> describes the process to run in the AWS console to create AWS IAM credentials to use on your device.

2. In the terminal window on your local host computer that's connected to your Raspberry Pi, and with the **Access Key ID** and **Secret Access Key** credentials for your device:

- a. Run the AWS configure app with this command:

```
aws configure
```

- b. Enter your credentials and configuration information when prompted:

```
AWS Access Key ID: your Access Key ID
AWS Secret Access Key: your Secret Access Key
Default region name: your AWS Region code
Default output format: json
```

3. Run this command to test your device's access to your AWS account and AWS IoT Core endpoint.

```
aws iot describe-endpoint --endpoint-type iot:Data-ATS
```

It should return your AWS account-specific AWS IoT data endpoint, such as this example:

```
{  
    "endpointAddress": "a3EXAMPLEffp-ats.iot.us-west-2.amazonaws.com"  
}
```

If you see your AWS account-specific AWS IoT data endpoint, your Raspberry Pi has the connectivity and permissions to continue to [the section called “Download the Amazon Root CA certificate” \(p. 126\)](#).

Important

Your AWS account credentials are now stored on the microSD card in your Raspberry Pi. While this makes future interactions with AWS easy for you and the software you'll create in these tutorials, they will also be saved and duplicated in any microSD card images you make after this step by default.

To protect the security of your AWS account credentials, before you save any more microSD card images, consider erasing the credentials by running `aws configure` again and entering random characters for the **Access Key ID** and **Secret Access Key** to prevent your AWS account credentials from compromised.

If you find that you have saved your AWS account credentials inadvertently, you can deactivate them in the AWS IAM console.

Download the Amazon Root CA certificate

This procedure downloads and saves a copy of a certificate of the Amazon Root Certificate Authority (CA). Downloading this certificate saves it for use in the subsequent tutorials and it also tests your device's connectivity with AWS services.

To download and save the Amazon Root CA certificate

1. Run this command to create a directory for the certificate.

```
mkdir ~/certs
```

2. Run this command to download the Amazon Root CA certificate.

```
curl -o ~/certs/AmazonRootCA1.pem https://www.amazontrust.com/repository/  
AmazonRootCA1.pem
```

3. Run these commands to set the access to the certificate directory and its file.

```
chmod 745 ~  
chmod 700 ~/certs  
chmod 644 ~/certs/AmazonRootCA1.pem
```

- Run this command to see the CA certificate file in the new directory.

```
ls -l ~/certs
```

You should see an entry like this. The date and time will be different; however, the file size and all other info should be the same as shown here.

```
-rw-r--r-- 1 pi pi 1188 Oct 28 13:02 AmazonRootCA1.pem
```

If the file size is not 1188, check the **curl** command parameters. You might have downloaded an incorrect file.

(Optional) Save the microSD card image

At this point, your Raspberry Pi's microSD card has an updated OS and the basic application software loaded.

To save the microSD card image to a file

- In the terminal window on your local host computer, clear your AWS credentials.

- Run the AWS configure app with this command:

```
aws configure
```

- Replace your credentials when prompted. You can leave **Default region name** and **Default output format** as they are by pressing **Enter**.

```
AWS Access Key ID [*****YT2H]: XXXXXXXXXX  
AWS Secret Access Key [*****9plH]: XXXXXXXXXX  
Default region name [us-west-2]:  
Default output format [json]:
```

- Enter this command to shut down the Raspberry Pi.

```
sudo shutdown -h 0
```

- After the Raspberry Pi shuts down completely, remove its power connector.

- Remove the microSD card from your device.

- On your local host computer:

- Insert the microSD card.
- Using your SD card imaging tool, save the microSD card's image to a file.
- After the microSD card's image has been saved, eject the card from the local host computer.
- With the power disconnected from the Raspberry Pi, insert the microSD card into the Raspberry Pi.
- Apply power to the device.
- After about a minute, on the local host computer, restart the terminal window session and log in to the device.

Don't reenter your AWS account credentials yet.

After you have restarted and logged in to your Raspberry Pi, you're ready to continue to [the section called "Installing and configuring the AWS IoT Device Client" \(p. 128\)](#).

Tutorial: Installing and configuring the AWS IoT Device Client

This tutorial walks you through the installation and configuration of the AWS IoT Device Client and the creation of AWS IoT resources that you'll use in this and other demos.

To start this tutorial:

- Have your local host computer and Raspberry Pi from [the previous tutorial \(p. 118\)](#) ready.

This tutorial can take up to 90 minutes to complete.

When you're finished with this topic:

- Your IoT device will be ready to use in other AWS IoT Device Client demos.
- You'll have provisioned your IoT device in AWS IoT Core.
- You'll have downloaded and installed the AWS IoT Device Client on your device.
- You'll have saved an image of your device's microSD card that can be used in subsequent tutorials.

Required equipment:

- Your local development and testing environment from [the previous section \(p. 124\)](#)
- The Raspberry Pi that you used in [the previous section \(p. 124\)](#)
- The microSD memory card from the Raspberry Pi that you used in [the previous section \(p. 124\)](#)

Procedures in this tutorial

- [Step 1: Download and save the AWS IoT Device Client \(p. 128\)](#)
- [\(Optional\) Save the microSD card image \(p. 129\)](#)
- [Step 2: Provision your Raspberry Pi in AWS IoT \(p. 130\)](#)
- [Step 3: Configure the AWS IoT Device Client to test connectivity \(p. 134\)](#)

Step 1: Download and save the AWS IoT Device Client

The procedures in this section download the AWS IoT Device Client, compile it, and install it on your Raspberry Pi. After you test the installation, you can save the image of the Raspberry Pi's microSD card to use later when you want to try the tutorials again.

Procedures in this section:

- [Download and build the AWS IoT Device Client \(p. 128\)](#)
- [Create the directories used by the tutorials \(p. 129\)](#)

Download and build the AWS IoT Device Client

This procedure installs the AWS IoT Device Client on your Raspberry Pi.

Perform these commands in the terminal window on your local host computer that is connected to your Raspberry Pi.

To install the AWS IoT Device Client on your Raspberry Pi

1. Enter these commands to download and build the AWS IoT Device Client on your Raspberry Pi.

```
cd ~  
git clone https://github.com/awslabs/aws-iot-device-client aws-iot-device-client  
mkdir ~/aws-iot-device-client/build && cd ~/aws-iot-device-client/build  
cmake .. /
```

2. Run this command to build the AWS IoT Device Client. This command can take up to 15 minutes to complete.

```
cmake --build . --target aws-iot-device-client
```

The warning messages displayed as the AWS IoT Device Client compiles can be ignored.

These tutorials have been tested with the AWS IoT Device Client built on **gcc**, version (Raspbian 10.2.1-6+rpi1) 10.2.1 20210110 on the Oct 30th 2021 version of Raspberry Pi OS (bullseye) on **gcc**, version (Raspbian 8.3.0-6+rpi1) 8.3.0 on the May 7th 2021 version of the Raspberry Pi OS (buster).

3. After the AWS IoT Device Client finishes building, test it by running this command.

```
./aws-iot-device-client --help
```

If you see the command line help for the AWS IoT Device Client, the AWS IoT Device Client has been built successfully and is ready for you to use.

Create the directories used by the tutorials

This procedure creates the directories on the Raspberry Pi that will be used to store the files used by the tutorials in this learning path.

To create the directories used by the tutorials in this learning path:

1. Run these commands to create the required directories.

```
mkdir ~/dc-configs  
mkdir ~/policies  
mkdir ~/messages  
mkdir ~/certs/testconn  
mkdir ~/certs/pubsub  
mkdir ~/certs/jobs
```

2. Run these commands to set the permissions on the new directories.

```
chmod 745 ~  
chmod 700 ~/certs/testconn  
chmod 700 ~/certs/pubsub  
chmod 700 ~/certs/jobs
```

After you create these directories and set their permission, continue to [the section called “\(Optional\) Save the microSD card image” \(p. 129\)](#).

(Optional) Save the microSD card image

At this point, your Raspberry Pi's microSD card has an updated OS, the basic application software, and the AWS IoT Device Client.

If you want to come back to try these exercises and tutorials again, you can skip the preceding procedures by writing the microSD card image that you save with this procedure to a new microSD card and continue the tutorials from [the section called “Step 2: Provision your Raspberry Pi in AWS IoT” \(p. 130\)](#).

To save the microSD card image to a file:

In the terminal window on your local host computer that's connected to your Raspberry Pi:

1. Confirm that your AWS account credentials have not been stored.

- a. Run the AWS configure app with this command:

```
aws configure
```

- b. If your credentials have been stored (if they are displayed in the prompt), then enter the **XXXXXXYYXX** string when prompted as shown here. Leave **Default region name** and **Default output format** blank.

```
AWS Access Key ID [*****YXXX]: XXXXXXXXYYXX
AWS Secret Access Key [*****YXXX]: XXXXXXXXYYXX
Default region name:
Default output format:
```

2. Enter this command to shutdown the Raspberry Pi.

```
sudo shutdown -h 0
```

3. After the Raspberry Pi shuts down completely, remove its power connector.
4. Remove the microSD card from your device.
5. On your local host computer:
 - a. Insert the microSD card.
 - b. Using your SD card imaging tool, save the microSD card's image to a file.
 - c. After the microSD card's image has been saved, eject the card from the local host computer.

You can continue with this microSD card in [the section called “Step 2: Provision your Raspberry Pi in AWS IoT” \(p. 130\)](#).

Step 2: Provision your Raspberry Pi in AWS IoT

The procedures in this section start with the saved microSD image that has the AWS CLI and AWS IoT Device Client installed and create the AWS IoT resources and device certificates that provision your Raspberry Pi in AWS IoT.

Install the microSD card in your Raspberry Pi

This procedure installs the microSD card with the necessary software loaded and configured into the Raspberry Pi and configures your AWS account so that you can continue with the tutorials in this learning path.

Use a microSD card from [the section called “\(Optional\) Save the microSD card image” \(p. 129\)](#) that has the necessary software for the exercises and tutorials in this learning path.

To install the microSD card in your Raspberry Pi

1. With the power disconnected from the Raspberry Pi, insert the microSD card into the Raspberry Pi.

2. Apply power to the Raspberry Pi.
3. After about a minute, on the local host computer, restart the terminal window session and log in to the Raspberry Pi.
4. On your local host computer, in the terminal window, and with the **Access Key ID** and **Secret Access Key** credentials for your Raspberry Pi:
 - a. Run the AWS configure app with this command:

```
aws configure
```

- b. Enter your AWS account credentials and configuration information when prompted:

```
AWS Access Key ID [*****YXXY]: your Access Key ID
AWS Secret Access Key [*****YXXY]: your Secret Access Key
Default region name [us-west-2]: your AWS Region code
Default output format [json]: json
```

After you have restored your AWS account credentials, you're ready to continue to [the section called "Provision your device in AWS IoT Core" \(p. 131\)](#).

Provision your device in AWS IoT Core

The procedures in this section create the AWS IoT resources that provision your Raspberry Pi in AWS IoT. As you create these resources, you'll be asked to record various pieces of information. This information is used by the AWS IoT Device Client configuration in the next procedure.

For your Raspberry Pi to work with AWS IoT, it must be provisioned. Provisioning is the process of creating and configuring the AWS IoT resources that are necessary to support your Raspberry Pi as an IoT device.

With your Raspberry Pi powered up and restarted, connect the terminal window on your local host computer to the Raspberry Pi and complete these procedures.

Procedures in this section:

- [Create and download device certificate files \(p. 131\)](#)
- [Create AWS IoT resources \(p. 132\)](#)

Create and download device certificate files

This procedure creates the device certificate files for this demo.

To create and download the device certificate files for your Raspberry Pi

1. In the terminal window on your local host computer, enter these commands to create the device certificate files for your device.

```
mkdir ~/certs/testconn
aws iot create-keys-and-certificate \
--set-as-active \
--certificate-pem-outfile "~/certs/testconn/device.pem.crt" \
--public-key-outfile "~/certs/testconn/public.pem.key" \
--private-key-outfile "~/certs/testconn/private.pem.key"
```

The command returns a response like the following. Record the *certificateArn* value for later use.

```
{  
    "certificateArn": "arn:aws:iot:us-  
west-2:57EXAMPLE833:cert/76e7e4edb3e52f52334be2f387a06145b2aa4c7fc810f3aea2d92abc227d269",  
    "certificateId":  
    "76e7e4edb3e52f5233EXAMPLE7a06145b2aa4c7fc810f3aea2d92abc227d269",  
    "certificatePem": "----BEGIN CERTIFICATE----  
\nMIIDWTCCAkGgAwIBAgI_SHORTENED_FOR_EXAMPLE_Lgn4jfgtS\n----END CERTIFICATE----\n",  
    "keyPair": {  
        "PublicKey": "----BEGIN PUBLIC KEY----  
\nMIIBIjANBgkqhkiG9w0BA_SHORTENED_FOR_EXAMPLE_ImwIDAQAB\n----END PUBLIC KEY----\n",  
        "PrivateKey": "----BEGIN RSA PRIVATE KEY----  
\nMIIEowIBAAKCAQE_SHORTENED_FOR_EXAMPLE_T9RoDiukY\n----END RSA PRIVATE KEY----\n"  
    }  
}
```

2. Enter the following commands to set the permissions on the certificate directory and its files.

```
chmod 745 ~  
chmod 700 ~/certs/testconn  
chmod 644 ~/certs/testconn/*  
chmod 600 ~/certs/testconn/private.pem.key
```

3. Run this command to review the permissions on your certificate directories and files.

```
ls -l ~/certs/testconn
```

The output of the command should be the same as what you see here, except the file dates and times will be different.

```
-rw-r--r-- 1 pi pi 1220 Oct 28 13:02 device.pem.crt  
-rw----- 1 pi pi 1675 Oct 28 13:02 private.pem.key  
-rw-r--r-- 1 pi pi 451 Oct 28 13:02 public.pem.key
```

At this point, you have the device certificate files installed on your Raspberry Pi and you can continue to the section called “Create AWS IoT resources” ([p. 132](#)).

Create AWS IoT resources

This procedure provisions your device in AWS IoT by creating the resources that your device needs to access AWS IoT features and services.

To provision your device in AWS IoT

1. In the terminal window on your local host computer, enter the following command to get the address of the device data endpoint for your AWS account.

```
aws iot describe-endpoint --endpoint-type IoT:Data-ATS
```

The command from the previous steps returns a response like the following. Record the `endpointAddress` value for later use.

```
{  
    "endpointAddress": "a3qjEXAMPLEffp-ats.iot.us-west-2.amazonaws.com"  
}
```

2. Enter this command to create an AWS IoT thing resource for your Raspberry Pi.

```
aws iot create-thing --thing-name "DevCliTestThing"
```

If your AWS IoT thing resource was created, the command returns a response like this.

```
{  
    "thingName": "DevCliTestThing",  
    "thingArn": "arn:aws:iot:us-west-2:57EXAMPLE833:thing/DevCliTestThing",  
    "thingId": "8ea78707-32c3-4f8a-9232-14bEXAMPLEfd"  
}
```

3. In the terminal window:

- Open a text editor, such as nano.
- Copy this JSON policy document and paste it into your open text editor.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:Publish",  
                "iot:Subscribe",  
                "iot:Receive",  
                "iot:Connect"  
            ],  
            "Resource": [  
                "*"  
            ]  
        }  
    ]  
}
```

Note

This policy document generously grants every resource permission to connect, receive, publish, and subscribe. Normally policies grant only permission to specific resources to perform specific actions. However, for the initial device connectivity test, this overly general and permissive policy is used to minimize the chance of an access problem during this test. In the subsequent tutorials, more narrowly scoped policy documents will be used to demonstrate better practices in policy design.

- Save the file in your text editor as `~/policies/dev_cli_test_thing_policy.json`.
- Run this command to use the policy document from the previous steps to create an AWS IoT policy.

```
aws iot create-policy \  
--policy-name "DevCliTestThingPolicy" \  
--policy-document "file://~/policies/dev_cli_test_thing_policy.json"
```

If the policy is created, the command returns a response like this.

```
{  
    "policyName": "DevCliTestThingPolicy",  
    "policyArn": "arn:aws:iot:us-west-2:57EXAMPLE833:policy/DevCliTestThingPolicy",  
    "policyDocument": "{\n        \"Version\": \"2012-10-17\", \"Statement\": [\n            {\n                \"Effect\": \"Allow\", \"Action\": [\n                    \"iot:Publish\", \"iot:Subscribe\", \"iot:Receive\",  
                    \"iot:Connect\"\n                ], \"Resource\": [\n                    \"*\"\n                ]\n            }\n        ]\n    }",  
    "version": 1,  
    "lastModifiedDate": "2017-01-12T17:15:00Z",  
    "createDate": "2017-01-12T17:15:00Z",  
    "lastUpdate": "2017-01-12T17:15:00Z",  
    "lastUpdateBy": "AWS IoT Device Client",  
    "version": 1  
}
```

```
    "policyVersionId": "1"  
}
```

5. Run this command to attach the policy to the device certificate. Replace `certificateArn` with the `certificateArn` value you saved earlier.

```
aws iot attach-policy \  
--policy-name "DevCliTestThingPolicy" \  
--target "certificateArn"
```

If successful, this command returns nothing.

6. Run this command to attach the device certificate to the AWS IoT thing resource. Replace `certificateArn` with the `certificateArn` value you saved earlier.

```
aws iot attach-thing-principal \  
--thing-name "DevCliTestThing" \  
--principal "certificateArn"
```

If successful, this command returns nothing.

After you successfully provisioned your device in AWS IoT, you're ready to continue to [the section called "Step 3: Configure the AWS IoT Device Client to test connectivity" \(p. 134\)](#).

Step 3: Configure the AWS IoT Device Client to test connectivity

The procedures in this section configure the AWS IoT Device Client to publish an MQTT message from your Raspberry Pi.

Procedures in this section:

- [Create the config file \(p. 134\)](#)
- [Open MQTT test client \(p. 135\)](#)
- [Run AWS IoT Device Client \(p. 136\)](#)

Create the config file

This procedure creates the config file to test the AWS IoT Device Client.

To create the config file to test the AWS IoT Device Client

- In the terminal window on your local host computer that's connected to your Raspberry Pi:
 - a. Enter these commands to create a directory for the config files and set the permission on the directory:

```
mkdir ~/dc-configs  
chmod 745 ~/dc-configs
```

- b. Open a text editor, such as nano.
- c. Copy this JSON document and paste it into your open text editor.

```
{  
  "endpoint": "a3qEXAMPLEaffp-ats.iot.us-west-2.amazonaws.com",  
  "cert": "~/certs/testconn/device.pem.crt",  
  "key": "~/certs/testconn/private.pem.key",  
  "root-ca": "~/certs/AmazonRootCA1.pem",
```

```
"thing-name": "DevCliTestThing",
"logging": {
    "enable-sdk-logging": true,
    "level": "DEBUG",
    "type": "STDOUT",
    "file": ""
},
"jobs": {
    "enabled": false,
    "handler-directory": ""
},
"tunneling": {
    "enabled": false
},
"device-defender": {
    "enabled": false,
    "interval": 300
},
"fleet-provisioning": {
    "enabled": false,
    "template-name": "",
    "template-parameters": "",
    "csr-file": "",
    "device-key": ""
},
"samples": {
    "pub-sub": {
        "enabled": true,
        "publish-topic": "test/dc/pubtopic",
        "publish-file": "",
        "subscribe-topic": "test/dc/subtopic",
        "subscribe-file": ""
    }
},
"config-shadow": {
    "enabled": false
},
"sample-shadow": {
    "enabled": false,
    "shadow-name": "",
    "shadow-input-file": "",
    "shadow-output-file": ""
}
}
```

- d. Replace the **endpoint** value with device data endpoint for your AWS account that you found in [the section called “Provision your device in AWS IoT Core” \(p. 131\)](#).
- e. Save the file in your text editor as `~/dc-configs/dc-testconn-config.json`.
- f. Run this command to set the permissions on the new config file.

```
chmod 644 ~/dc-configs/dc-testconn-config.json
```

After you save the file, you’re ready to continue to [the section called “Open MQTT test client” \(p. 135\)](#).

Open MQTT test client

This procedure prepares the **MQTT test client** in the AWS IoT console to subscribe to the MQTT message that the AWS IoT Device Client publishes when it runs.

To prepare the MQTT test client to subscribe to all MQTT messages

1. On your local host computer, in the [AWS IoT console](#), choose **MQTT test client**.

2. In the **Subscribe to a topic** tab, in **Topic filter**, enter # (a single pound sign), and choose **Subscribe** to subscribe to every MQTT topic.
3. Below the **Subscriptions** label, confirm that you see # (a single pound sign).

Leave the window with the **MQTT test client** open as you continue to [the section called “Run AWS IoT Device Client” \(p. 136\)](#).

Run AWS IoT Device Client

This procedure runs the AWS IoT Device Client so that it publishes a single MQTT message that the **MQTT test client** receives and displays.

To send an MQTT message from the AWS IoT Device Client

1. Make sure that both the terminal window that's connected to your Raspberry Pi and the window with the **MQTT test client** are visible while you perform this procedure.
2. In the terminal window, enter these commands to run the AWS IoT Device Client using the config file created in [the section called “Create the config file” \(p. 134\)](#).

```
cd ~/aws-iot-device-client/build
./aws-iot-device-client --config-file ~/dc-configs/dc-testconn-config.json
```

In the terminal window, the AWS IoT Device Client displays information messages and any errors that occur when it runs.

If no errors are displayed in the terminal window, review the **MQTT test client**.

3. In the **MQTT test client**, in the Subscriptions window, see the *Hello World!* message sent to the `test/dc/pubtopic` message topic.
4. If the AWS IoT Device Client displays no errors and you see *Hello World!* sent to the `test/dc/pubtopic` message in the **MQTT test client**, you've demonstrated a successful connection.
5. In the terminal window, enter `^C` (Ctrl-C) to stop the AWS IoT Device Client.

After you've demonstrated that the AWS IoT Device Client is running correctly on your Raspberry Pi and can communicate with AWS IoT, you can continue to [the section called “Demonstrate MQTT message communication with the AWS IoT Device Client” \(p. 136\)](#).

Tutorial: Demonstrate MQTT message communication with the AWS IoT Device Client

This tutorial demonstrates how the AWS IoT Device Client can subscribe to and publish MQTT messages, which are commonly used in IoT solutions.

To start this tutorial:

- Have your local host computer and Raspberry Pi configured as used in [the previous section \(p. 128\)](#).
If you saved the microSD card image after installing the AWS IoT Device Client, you can use a microSD card with that image with your Raspberry Pi.
- If you have run this demo before, review [??? \(p. 161\)](#) to delete all AWS IoT resources that you created in earlier runs to avoid duplicate resource errors.

This tutorial takes about 45 minutes to complete.

When you're finished with this topic:

- You'll have demonstrated different ways that your IoT device can subscribe to MQTT messages from AWS IoT and publish MQTT messages to AWS IoT.

Required equipment:

- Your local development and testing environment from [the previous section \(p. 128\)](#)
- The Raspberry Pi that you used in [the previous section \(p. 128\)](#)
- The microSD memory card from the Raspberry Pi that you used in [the previous section \(p. 128\)](#)

Procedures in this tutorial

- [Step 1: Prepare the Raspberry Pi to demonstrate MQTT message communication \(p. 137\)](#)
- [Step 2: Demonstrate publishing messages with the AWS IoT Device Client \(p. 142\)](#)
- [Step 3: Demonstrate subscribing to messages with the AWS IoT Device Client \(p. 145\)](#)

Step 1: Prepare the Raspberry Pi to demonstrate MQTT message communication

This procedure creates the resources in AWS IoT and in the Raspberry Pi to demonstrate MQTT message communication using the AWS IoT Device Client.

Procedures in this section:

- [Create the certificate files to demonstrate MQTT communication \(p. 137\)](#)
- [Provision your device to demonstrate MQTT communication \(p. 138\)](#)
- [Configure the AWS IoT Device Client config file and MQTT test client to demonstrate MQTT communication \(p. 140\)](#)

Create the certificate files to demonstrate MQTT communication

This procedure creates the device certificate files for this demo.

To create and download the device certificate files for your Raspberry Pi

1. In the terminal window on your local host computer, enter the following command to create the device certificate files for your device.

```
mkdir ~/certs/pubsub
aws iot create-keys-and-certificate \
--set-as-active \
--certificate-pem-outfile "~/certs/pubsub/device.pem.crt" \
--public-key-outfile "~/certs/pubsub/public.pem.key" \
--private-key-outfile "~/certs/pubsub/private.pem.key"
```

The command returns a response like the following. Save the `certificateArn` value for later use.

```
{
"certificateArn": "arn:aws:iot:us-
west-2:57EXAMPLE833:cert/76e7e4edb3e52f52334be2f387a06145b2aa4c7fc810f3aea2d92abc227d269",
"certificateId": "76e7e4edb3e52f5233EXAMPLE7a06145b2aa4c7fc810f3aea2d92abc227d269",
"certificatePem": "-----BEGIN CERTIFICATE-----
\nMIIDWTCCAkGgAwIBAgI_SHORTENED_FOR_EXAMPLE_Lgn4jfgtS\n-----END CERTIFICATE-----\n",
```

```
"keyPair": {  
    "PublicKey": "-----BEGIN PUBLIC KEY-----  
    \nMIIBIjANBgkqhkiG9w0BA_SHORTENED_FOR_EXAMPLE_ImwIDAQAB\n-----END PUBLIC KEY-----\n",  
    "PrivateKey": "-----BEGIN RSA PRIVATE KEY-----  
    \nMIIEowIBAAKCAQE_SHORTENED_FOR_EXAMPLE_T9RoDiukY\n-----END RSA PRIVATE KEY-----\n"  
}
```

2. Enter the following commands to set the permissions on the certificate directory and its files.

```
chmod 700 ~/certs/pubsub  
chmod 644 ~/certs/pubsub/*  
chmod 600 ~/certs/pubsub/private.pem.key
```

3. Run this command to review the permissions on your certificate directories and files.

```
ls -l ~/certs/pubsub
```

The output of the command should be the same as what you see here, except the file dates and times will be different.

```
-rw-r--r-- 1 pi pi 1220 Oct 28 13:02 device.pem.crt  
-rw----- 1 pi pi 1675 Oct 28 13:02 private.pem.key  
-rw-r--r-- 1 pi pi 451 Oct 28 13:02 public.pem.key
```

4. Enter these commands to create the directories for the log files.

```
mkdir ~/.aws-iot-device-client  
mkdir ~/.aws-iot-device-client/log  
chmod 745 ~/.aws-iot-device-client/log  
echo " " > ~/.aws-iot-device-client/log/aws-iot-device-client.log  
echo " " > ~/.aws-iot-device-client/log/pubsub_rx_msgs.log  
chmod 600 ~/.aws-iot-device-client/log/*
```

Provision your device to demonstrate MQTT communication

This section creates the AWS IoT resources that provision your Raspberry Pi in AWS IoT.

To provision your device in AWS IoT:

1. In the terminal window on your local host computer, enter the following command to get the address of the device data endpoint for your AWS account.

```
aws iot describe-endpoint --endpoint-type IoT:Data-ATS
```

The endpoint value hasn't changed since the time you ran this command for the previous tutorial. Running the command again here is done to make it easy to find and paste the data endpoint value into the config file used in this tutorial.

The command from the previous steps returns a response like the following. Record the *endpointAddress* value for later use.

```
{  
    "endpointAddress": "a3qjEXAMPLEffp-ats.iot.us-west-2.amazonaws.com"  
}
```

2. Enter this command to create a new AWS IoT thing resource for your Raspberry Pi.

```
aws iot create-thing --thing-name "PubSubTestThing"
```

Because an AWS IoT thing resource is a *virtual* representation of your device in the cloud, we can create multiple thing resources in AWS IoT to use for different purposes. They can all be used by the same physical IoT device to represent different aspects of the device.

These tutorials will only use one thing resource at a time to represent the Raspberry Pi. This way, in these tutorials, they represent the different demos so that after you create the AWS IoT resources for a demo, you can go back and repeat the demo using the resources you created specifically for each.

If your AWS IoT thing resource was created, the command returns a response like this.

```
{
  "thingName": "PubSubTestThing",
  "thingArn": "arn:aws:iot:us-west-2:57EXAMPLE833:thing/PubSubTestThing",
  "thingId": "8ea78707-32c3-4f8a-9232-14bEXAMPLEfd"
}
```

3. In the terminal window:

- a. Open a text editor, such as nano.
- b. Copy this JSON document and paste it into your open text editor.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:Connect"
      ],
      "Resource": [
        "arn:aws:iot:us-west-2:57EXAMPLE833:client/PubSubTestThing"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iot:Publish"
      ],
      "Resource": [
        "arn:aws:iot:us-west-2:57EXAMPLE833:topic/test/dc/pubtopic"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iot:Subscribe"
      ],
      "Resource": [
        "arn:aws:iot:us-west-2:57EXAMPLE833:topicfilter/test/dc/subtopic"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iot:Receive"
      ],
      "Resource": [
        "arn:aws:iot:us-west-2:57EXAMPLE833:topic/test/dc/subtopic"
      ]
    }
  ]
}
```

```
        ]  
    }  
}
```

- c. In the editor, in each Resource section of the policy document, replace `us-west-2:57EXAMPLE833` with your AWS Region, a colon character (:), and your 12-digit AWS account number.
- d. Save the file in your text editor as `~/policies/pubsub_test_thing_policy.json`.
4. Run this command to use the policy document from the previous steps to create an AWS IoT policy.

```
aws iot create-policy \  
--policy-name "PubSubTestThingPolicy" \  
--policy-document "file://~/policies/pubsub_test_thing_policy.json"
```

If the policy is created, the command returns a response like this.

```
{  
    "policyName": "PubSubTestThingPolicy",  
    "policyArn": "arn:aws:iot:us-west-2:57EXAMPLE833:policy/PubSubTestThingPolicy",  
    "policyDocument": "{\n        \"Version\": \"2012-10-17\",  
        \"Statement\": [\n            {\n                \"Effect\": \"Allow\",  
                \"Action\": \"iot:Connect\",  
                \"Resource\": \"arn:aws:iot:us-west-2:57EXAMPLE833:client/PubSubTestThing\"  
            },  
            {\n                \"Effect\": \"Allow\",  
                \"Action\": \"iot:Publish\",  
                \"Resource\": \"arn:aws:iot:us-west-2:57EXAMPLE833:topic/test/dc/pubtopic\"  
            },  
            {\n                \"Effect\": \"Allow\",  
                \"Action\": \"iot:Subscribe\",  
                \"Resource\": \"arn:aws:iot:us-west-2:57EXAMPLE833:topicfilter/test/dc/subtopic\"  
            },  
            {\n                \"Effect\": \"Allow\",  
                \"Action\": \"iot:Receive\",  
                \"Resource\": \"arn:aws:iot:us-west-2:57EXAMPLE833:topic/test/dc/*\"  
            }  
        ]  
    }  
    "policyVersionId": "1"
```

5. Run this command to attach the policy to the device certificate. Replace `certificateArn` with the `certificateArn` value you saved earlier in this section.

```
aws iot attach-policy \  
--policy-name "PubSubTestThingPolicy" \  
--target "certificateArn"
```

If successful, this command returns nothing.

6. Run this command to attach the device certificate to the AWS IoT thing resource. Replace `certificateArn` with the `certificateArn` value you saved earlier in this section.

```
aws iot attach-thing-principal \  
--thing-name "PubSubTestThing" \  
--principal "certificateArn"
```

If successful, this command returns nothing.

After you successfully provision your device in AWS IoT, you're ready to continue to the section called ["Configure the AWS IoT Device Client config file and MQTT test client to demonstrate MQTT communication" \(p. 140\)](#).

Configure the AWS IoT Device Client config file and MQTT test client to demonstrate MQTT communication

This procedure creates a config file to test the AWS IoT Device Client.

To create the config file to test the AWS IoT Device Client

1. In the terminal window on your local host computer that's connected to your Raspberry Pi:
 - a. Open a text editor, such as nano.
 - b. Copy this JSON document and paste it into your open text editor.

```
{  
    "endpoint": "a3qEXAMPLEaffp-ats.iot.us-west-2.amazonaws.com",  
    "cert": "~/.certs/pubsub/device.pem.crt",  
    "key": "~/.certs/pubsub/private.pem.key",  
    "root-ca": "~/.certs/AmazonRootCA1.pem",  
    "thing-name": "PubSubTestThing",  
    "logging": {  
        "enable-sdk-logging": true,  
        "level": "DEBUG",  
        "type": "STDOUT",  
        "file": ""  
    },  
    "jobs": {  
        "enabled": false,  
        "handler-directory": ""  
    },  
    "tunneling": {  
        "enabled": false  
    },  
    "device-defender": {  
        "enabled": false,  
        "interval": 300  
    },  
    "fleet-provisioning": {  
        "enabled": false,  
        "template-name": "",  
        "template-parameters": "",  
        "csr-file": "",  
        "device-key": ""  
    },  
    "samples": {  
        "pub-sub": {  
            "enabled": true,  
            "publish-topic": "test/dc/pubtopic",  
            "publish-file": "",  
            "subscribe-topic": "test/dc/subtopic",  
            "subscribe-file": "~/.aws-iot-device-client/log/pubsub_rx_msgs.log"  
        }  
    },  
    "config-shadow": {  
        "enabled": false  
    },  
    "sample-shadow": {  
        "enabled": false,  
        "shadow-name": "",  
        "shadow-input-file": "",  
        "shadow-output-file": ""  
    }  
}
```

- c. Replace the `endpoint` value with device data endpoint for your AWS account that you found in the section called "Provision your device in AWS IoT Core" (p. 131).
- d. Save the file in your text editor as `~/dc-configs/dc-pubsub-config.json`.
- e. Run this command to set the permissions on the new config file.

```
chmod 644 ~/dc-configs/dc-pubsub-config.json
```

2. To prepare the **MQTT test client** to subscribe to all MQTT messages:
 - a. On your local host computer, in the [AWS IoT console](#), choose **MQTT test client**.
 - b. In the **Subscribe to a topic** tab, in **Topic filter**, enter **#** (a single pound sign), and choose **Subscribe**.
 - c. Below the **Subscriptions** label, confirm that you see **#** (a single pound sign).

Leave the window with the **MQTT test client** open while you continue this tutorial.

After you save the file and configure the **MQTT test client**, you're ready to continue to [the section called "Step 2: Demonstrate publishing messages with the AWS IoT Device Client" \(p. 142\)](#).

Step 2: Demonstrate publishing messages with the AWS IoT Device Client

The procedures in this section demonstrate how the AWS IoT Device Client can send default and custom MQTT messages.

These policy statements in the policy that you created in the previous step for these exercises give the Raspberry Pi permission to perform these actions:

- **iot:Connect**

Gives the client named `PubSubTestThing`, your Raspberry Pi running the AWS IoT Device Client, to connect.

```
{  
    "Effect": "Allow",  
    "Action": [  
        "iot:Connect"  
    ],  
    "Resource": [  
        "arn:aws:iot:us-west-2:57EXAMPLE833:client/PubSubTestThing"  
    ]  
}
```

- **iot:Publish**

Gives the Raspberry Pi permission to publish messages with an MQTT topic of `test/dc/pubtopic`.

```
{  
    "Effect": "Allow",  
    "Action": [  
        "iot:Publish"  
    ],  
    "Resource": [  
        "arn:aws:iot:us-west-2:57EXAMPLE833:topic/test/dc/pubtopic"  
    ]  
}
```

The `iot:Publish` action gives permission to publish to the MQTT topics listed in the Resource array. The *content* of those messages is not controlled by the policy statement.

Publish the default message using the AWS IoT Device Client

This procedure runs the AWS IoT Device Client so that it publishes a single default MQTT message that the **MQTT test client** receives and displays.

To send the default MQTT message from the AWS IoT Device Client

1. Make sure that both the terminal window on your local host computer that's connected to your Raspberry Pi and the window with the **MQTT test client** are visible while you perform this procedure.
2. In the terminal window, enter these commands to run the AWS IoT Device Client using the config file created in [the section called "Create the config file" \(p. 134\)](#).

```
cd ~/aws-iot-device-client/build
./aws-iot-device-client --config-file ~/dc-configs/dc-pubsub-config.json
```

In the terminal window, the AWS IoT Device Client displays information messages and any errors that occur when it runs.

If no errors are displayed in the terminal window, review the **MQTT test client**.

3. In the **MQTT test client**, in the **Subscriptions** window, see the *Hello World!* message sent to the `test/dc/pubtopic` message topic.
4. If the AWS IoT Device Client displays no errors and you see *Hello World!* sent to the `test/dc/pubtopic` message in the **MQTT test client**, you've demonstrated a successful connection.
5. In the terminal window, enter `^C` (Ctrl-C) to stop the AWS IoT Device Client.

After you've demonstrated that the AWS IoT Device Client published the default MQTT message, you can continue to the [the section called "Publish a custom message using the AWS IoT Device Client." \(p. 143\)](#).

Publish a custom message using the AWS IoT Device Client.

The procedures in this section create a custom MQTT message and then runs the AWS IoT Device Client so that it publishes the custom MQTT message one time for the **MQTT test client** to receive and display.

Create a custom MQTT message for the AWS IoT Device Client

Perform these steps in the terminal window on the local host computer that's connected to your Raspberry Pi.

To create a custom message for the AWS IoT Device Client to publish

1. In the terminal window, open a text editor, such as nano.
2. Into the text editor, copy and paste the following JSON document. This will be the MQTT message payload that the AWS IoT Device Client publishes.

```
{
  "temperature": 28,
  "humidity": 80,
  "barometer": 1013,
  "wind": {
    "velocity": 22,
    "bearing": 255
  }
}
```

3. Save the contents of the text editor as `~/messages/sample-ws-message.json`.
4. Enter the following command to set the permissions of the message file that you just created.

```
chmod 600 ~/messages/*
```

To create a config file for the AWS IoT Device Client to use to send the custom message

1. In the terminal window, in a text editor such as nano, open the existing AWS IoT Device Client config file: `~/dc-configs/dc-pubsub-config.json`.
2. Edit the `samples` object to look like this. No other part of this file needs to be changed.

```
"samples": {  
    "pub-sub": {  
        "enabled": true,  
        "publish-topic": "test/dc/pubtopic",  
        "publish-file": "~/messages/sample-ws-message.json",  
        "subscribe-topic": "test/dc/subtopic",  
        "subscribe-file": "~/.aws-iot-device-client/log/pubsub_rx_msgs.log"
```

3. Save the contents of the text editor as `~/dc-configs/dc-pubsub-custom-config.json`.
4. Run this command to set the permissions on the new config file.

```
chmod 644 ~/dc-configs/dc-pubsub-custom-config.json
```

Publish the custom MQTT message by using the AWS IoT Device Client

This change affects only the *contents* of the MQTT message payload, so the current policy will continue to work. However, if the *MQTT topic* (as defined by the `publish-topic` value in `~/dc-configs/dc-pubsub-custom-config.json`) was changed, the `iot::Publish` policy statement would also need to be modified to allow the Raspberry Pi to publish to the new MQTT topic.

To send the MQTT message from the AWS IoT Device Client

1. Make sure that both the terminal window and the window with the **MQTT test client** are visible while you perform this procedure. Also, make sure that your **MQTT test client** is still subscribed to the `#` topic filter. If it isn't, subscribe to the `#` topic filter again.
2. In the terminal window, enter these commands to run the AWS IoT Device Client using the config file created in [the section called "Create the config file" \(p. 134\)](#).

```
cd ~/aws-iot-device-client/build  
./aws-iot-device-client --config-file ~/dc-configs/dc-pubsub-custom-config.json
```

In the terminal window, the AWS IoT Device Client displays information messages and any errors that occur when it runs.

- If no errors are displayed in the terminal window, review the MQTT test client.
3. In the **MQTT test client**, in the **Subscriptions** window, see the custom message payload sent to the `test/dc/pubtopic` message topic.
 4. If the AWS IoT Device Client displays no errors and you see the custom message payload that you published to the `test/dc/pubtopic` message in the **MQTT test client**, you've published a custom message successfully.
 5. In the terminal window, enter `^C` (Ctrl-C) to stop the AWS IoT Device Client.

After you've demonstrated that the AWS IoT Device Client published a custom message payload, you can continue to the section called "[Step 3: Demonstrate subscribing to messages with the AWS IoT Device Client](#)" (p. 145).

Step 3: Demonstrate subscribing to messages with the AWS IoT Device Client

In this section, you'll demonstrate two types of message subscriptions:

- Single topic subscription
- Wild-card topic subscription

These policy statements in the policy created for these exercises give the Raspberry Pi permission to perform these actions:

- **iot:Receive**

Gives the AWS IoT Device Client permission to receive MQTT topics that match those named in the Resource object.

```
{  
    "Effect": "Allow",  
    "Action": [  
        "iot:Receive"  
    ],  
    "Resource": [  
        "arn:aws:iot:us-west-2:57EXAMPLE833:topic/test/dc/subtopic"  
    ]  
}
```

- **iot:Subscribe**

Gives the AWS IoT Device Client permission to subscribe to MQTT topic filters that match those named in the Resource object.

```
{  
    "Effect": "Allow",  
    "Action": [  
        "iot:Subscribe"  
    ],  
    "Resource": [  
        "arn:aws:iot:us-west-2:57EXAMPLE833:topicfilter/test/dc/subtopic"  
    ]  
}
```

Subscribe to a single MQTT message topic

This procedure demonstrates how the AWS IoT Device Client can subscribe to and log MQTT messages.

In the terminal window on your local host computer that's connected to your Raspberry Pi, list the contents of `~/dc-configs/dc-pubsub-custom-config.json` or open the file in a text editor to review its contents. Locate the `samples` object, which should look like this.

```
"samples": {  
    "pub-sub": {  
        "enabled": true,  
    }  
}
```

```
"publish-topic": "test/dc/pubtopic",
"publish-file": "~/messages/sample-ws-message.json",
"subscribe-topic": "test/dc/subtopic",
"subscribe-file": "~/.aws-iot-device-client/log/pubsub_rx_msgs.log"
```

Notice the `subscribe-topic` value is the MQTT topic to which the AWS IoT Device Client will subscribe when it runs. The AWS IoT Device Client writes the message payloads that it receives from this subscription to the file named in the `subscribe-file` value.

To subscribe to a MQTT message topic from the AWS IoT Device Client

1. Make sure that both the terminal window and the window with the MQTT test client are visible while you perform this procedure. Also, make sure that your **MQTT test client** is still subscribed to the `#` topic filter. If it isn't, subscribe to the `#` topic filter again.
2. In the terminal window, enter these commands to run the AWS IoT Device Client using the config file created in [the section called "Create the config file" \(p. 134\)](#).

```
cd ~/aws-iot-device-client/build
./aws-iot-device-client --config-file ~/dc-configs/dc-pubsub-custom-config.json
```

In the terminal window, the AWS IoT Device Client displays information messages and any errors that occur when it runs.

If no errors are displayed in the terminal window, continue in the AWS IoT console.

3. In the AWS IoT console, in the **MQTT test client**, choose the **Publish to a topic** tab.
4. In **Topic name**, enter `test/dc/subtopic`
5. In **Message payload**, review the message contents.
6. Choose **Publish** to publish the MQTT message.
7. In the terminal window, watch for the *message received* entry from the AWS IoT Device Client that looks like this.

```
2021-11-10T16:02:20.890Z [DEBUG] {samples/PubSubFeature.cpp}: Message received on
subscribe topic, size: 45 bytes
```

8. After you see the *message received* entry that shows the message was received, enter **^C** (Ctrl-C) to stop the AWS IoT Device Client.
9. Enter this command to view the end of the message log file and see the message you published from the **MQTT test client**.

```
tail ~/.aws-iot-device-client/log/pubsub_rx_msgs.log
```

By viewing the message in the log file, you've demonstrated that the AWS IoT Device Client received the message that you published from the MQTT test client.

Subscribe to multiple MQTT message topic using wildcard characters

These procedures demonstrate how the AWS IoT Device Client can subscribe to and log MQTT messages using wildcard characters. To do this, you'll:

1. Update the topic filter that the AWS IoT Device Client uses to subscribe to MQTT topics.
2. Update the policy used by the device to allow the new subscriptions.
3. Run the AWS IoT Device Client and publish messages from the MQTT test console.

To create a config file to subscribe to multiple MQTT message topics by using a wildcard MQTT topic filter

1. In the terminal window on your local host computer that's connected to your Raspberry Pi, open `~/dc-configs/dc-pubsub-custom-config.json` for editing and locate the `samples` object.
2. In the text editor, locate the `samples` object and update the `subscribe-topic` value to look like this.

```
"samples": {  
    "pub-sub": {  
        "enabled": true,  
        "publish-topic": "test/dc/pubtopic",  
        "publish-file": "~/messages/sample-ws-message.json",  
        "subscribe-topic": "test/dc/#",  
        "subscribe-file": "~/.aws-iot-device-client/log/pubsub_rx_msgs.log"
```

The new `subscribe-topic` value is an [MQTT topic filter \(p. 94\)](#) with an MQTT wild card character at the end. This describes a subscription to all MQTT topics that start with `test/dc/`. The AWS IoT Device Client writes the message payloads that it receives from this subscription to the file named in `subscribe-file`.

3. Save the modified config file as `~/dc-configs/dc-pubsub-wild-config.json`, and exit the editor.

To modify the policy used by your Raspberry Pi to allow subscribing to and receiving multiple MQTT message topics

1. In the terminal window on your local host computer that's connected to your Raspberry Pi, in your favorite text editor, open `~/policies/pubsub_test_thing_policy.json` for editing, and then locate the `iot::Subscribe` and `iot::Receive` policy statements in the file.
2. In the `iot::Subscribe` policy statement, update the string in the `Resource` object to replace `subtopic` with `*`, so that it looks like this.

```
{  
    "Effect": "Allow",  
    "Action": [  
        "iot:Subscribe"  
    ],  
    "Resource": [  
        "arn:aws:iot:us-west-2:57EXAMPLE833:topicfilter/test/dc/*"  
    ]  
}
```

Note

The [MQTT topic filter wild card characters \(p. 94\)](#) are the + (plus sign) and the # (pound sign). A subscription request with a # at the end subscribes to all topics that start with the string that precedes the # character (for example, `test/dc/` in this case).

The resource value in the policy statement that authorizes this subscription, however, must use a * (an asterisk) in place of the # (a pound sign) in the topic filter ARN. This is because the policy processor uses a different wild card character than MQTT uses.

For more information about using wild card characters for topics and topic filters in policies, see [Policies for MQTT clients \(p. 332\)](#).

3. In the `iot::Receive` policy statement, update the string in the `Resource` object to replace `subtopic` with `*`, so that it looks like this.

```
{  
    "Effect": "Allow",
```

```
"Action": [
    "iot:Receive"
],
"Resource": [
    "arn:aws:iot:us-west-2:57EXAMPLE833:topic/test/dc/*"
]
}
```

4. Save the updated policy document as `~/policies/pubsub_wild_test_thing_policy.json`, and exit the editor.
5. Enter this command to update the policy for this tutorial to use the new resource definitions.

```
aws iot create-policy-version \
--set-as-default \
--policy-name "PubSubTestThingPolicy" \
--policy-document "file://~/policies/pubsub_wild_test_thing_policy.json"
```

If the command succeeds, it returns a response like this. Notice that `policyVersionId` is now 2, indicating this is the second version of this policy.

If you successfully updated the policy, you can continue to the next procedure.

```
{
    "policyArn": "arn:aws:iot:us-west-2:57EXAMPLE833:policy/PubSubTestThingPolicy",
    "policyDocument": "{\n        \"Version\": \"2012-10-17\", \n        \"Statement\": [\n            {\n                \"Effect\": \"Allow\", \n                \"Action\": \"iot:Connect\", \n                \"Resource\": \"arn:aws:iot:us-west-2:57EXAMPLE833:client/PubSubTestThing\" \n            },\n            {\n                \"Effect\": \"Allow\", \n                \"Action\": \"iot:Publish\", \n                \"Resource\": \"arn:aws:iot:us-west-2:57EXAMPLE833:topic/test/dc/pubtopic\" \n            },\n            {\n                \"Effect\": \"Allow\", \n                \"Action\": \"iot:Subscribe\", \n                \"Resource\": \"arn:aws:iot:us-west-2:57EXAMPLE833:topicfilter/test/dc/*\" \n            },\n            {\n                \"Effect\": \"Allow\", \n                \"Action\": \"iot:Receive\", \n                \"Resource\": \"arn:aws:iot:us-west-2:57EXAMPLE833:topic/test/dc/*\" \n            } \n        ]\n    },\n    \"policyVersionId\": \"2\", \n    \"isDefaultVersion\": true\n}
```

If you get an error that there are too many policy versions to save a new one, enter this command to list the current versions of the policy. Review the list that this command returns to find a policy version that you can delete.

```
aws iot list-policy-versions --policy-name "PubSubTestThingPolicy"
```

Enter this command to delete a version that you no longer need. Note that you can't delete the default policy version. The default policy version is the one with a `isDefaultVersion` value of `true`.

```
aws iot delete-policy-version \
--policy-name "PubSubTestThingPolicy" \
--policy-version-id policyId
```

After deleting a policy version, retry this step.

With the updated config file and policy, you're ready to demonstrate wild card subscriptions with the AWS IoT Device Client.

To demonstrate how the AWS IoT Device Client subscribes to and receives multiple MQTT message topics

1. In the **MQTT test client**, check the subscriptions. If the **MQTT test client** is subscribed to the **#** topic filter, continue to the next step. If not, in the **MQTT test client**, in **Subscribe to a topic** tab, in **Topic filter**, enter **#** (a pound sign character), and then choose **Subscribe** to subscribe to it.
2. In the terminal window on your local host computer that's connected to your Raspberry Pi, enter these commands to start the AWS IoT Device Client.

```
cd ~/aws-iot-device-client/build  
./aws-iot-device-client --config-file ~/dc-configs/dc-pubsub-wild-config.json
```

3. While watching the AWS IoT Device Client output in the terminal window on the local host computer, return to the **MQTT test client**. In the **Publish to a topic** tab, in **Topic name**, enter **test/dc/subtopic**, and then choose **Publish**.
4. In the terminal window, confirm that the message was received by looking for a message such as:

```
2021-11-10T16:34:20.101Z [DEBUG] {samples/PubSubFeature.cpp}: Message received on  
subscribe topic, size: 76 bytes
```

5. While watching the AWS IoT Device Client output in the terminal window of the local host computer, return to the **MQTT test client**. In the **Publish to a topic** tab, in **Topic name**, enter **test/dc/subtopic2**, and then choose **Publish**.
6. In the terminal window, confirm that the message was received by looking for a message such as:

```
2021-11-10T16:34:32.078Z [DEBUG] {samples/PubSubFeature.cpp}: Message received on  
subscribe topic, size: 77 bytes
```

7. After you see the messages that confirm both messages were received, enter **^C** (Ctrl-C) to stop the AWS IoT Device Client.
8. Enter this command to view the end of the message log file and see the message you published from the **MQTT test client**.

```
tail -n 20 ~/.aws-iot-device-client/log/pubsub_rx_msgs.log
```

Note

The log file contains only message payloads. The message topics are not recorded in the received message log file.

You might also see the message published by the AWS IoT Device Client in the received log. This is because the wild card topic filter includes that message topic and, sometimes, the subscription request can be processed by message broker before the published message is sent to subscribers.

The entries in the log file demonstrate that the messages were received. You can repeat this procedure using other topic names. All messages that have a topic name that begins with `test/dc/` should be received and logged. Messages with topic names that begin with any other text are ignored.

After demonstrating how the AWS IoT Device Client can publish and subscribe to MQTT messages, continue to [Tutorial: Demonstrate remote actions \(jobs\) with the AWS IoT Device Client \(p. 150\)](#).

Tutorial: Demonstrate remote actions (jobs) with the AWS IoT Device Client

In these tutorials, you'll configure and deploy jobs to your Raspberry Pi to demonstrate how you can send remote operations to your IoT devices.

To start this tutorial:

- Have your local host computer an Raspberry Pi configured as used in [the previous section \(p. 136\)](#).
- If you haven't completed the tutorial in the previous section, you can try this tutorial by using the Raspberry Pi with a microSD card that has the image you saved after you installed the AWS IoT Device Client in [\(Optional\) Save the microSD card image \(p. 129\)](#).
- If you have run this demo before, review [??? \(p. 161\)](#) to delete all AWS IoT resources that you created in earlier runs to avoid duplicate resource errors.

This tutorial takes about 45 minutes to complete.

When you're finished with this topic:

- You'll have demonstrated different ways that your IoT device can use the AWS IoT Core to run remote operations that are managed by AWS IoT .

Required equipment:

- Your local development and testing environment that you tested in [a previous section \(p. 128\)](#)
- The Raspberry Pi that you tested in [a previous section \(p. 128\)](#)
- The microSD memory card from the Raspberry Pi that you tested in [a previous section \(p. 128\)](#)

Procedures in this tutorial

- [Step 1: Prepare the Raspberry Pi to run jobs \(p. 150\)](#)
- [Step 2: Create and run the job in AWS IoT \(p. 156\)](#)

Step 1: Prepare the Raspberry Pi to run jobs

The procedures in this section describe how to prepare your Raspberry Pi to run jobs by using the AWS IoT Device Client.

Note

These procedures are device specific. If you want to perform the procedures in this section with more than one device at the same time, each device will need its own policy and unique, device-specific certificate and thing name. To give each device its unique resources, perform this procedure one time for each device while changing the device-specific elements as described in the procedures.

Procedures in this tutorial

- [Provision your Raspberry Pi to demonstrate jobs \(p. 151\)](#)
- [Configure the AWS IoT Device Client to run the jobs agent \(p. 155\)](#)

Provision your Raspberry Pi to demonstrate jobs

The procedures in this section provision your Raspberry Pi in AWS IoT by creating AWS IoT resources and device certificates for it.

Create and download device certificate files to demonstrate AWS IoT jobs

This procedure creates the device certificate files for this demo.

If you are preparing more than one device, this procedure must be performed on each device.

To create and download the device certificate files for your Raspberry Pi:

In the terminal window on your local host computer that's connected to your Raspberry Pi, enter these commands.

1. Enter the following command to create the device certificate files for your device.

```
aws iot create-keys-and-certificate \
--set-as-active \
--certificate-pem-outfile "~/certs/jobs/device.pem.crt" \
--public-key-outfile "~/certs/jobs/public.pem.key" \
--private-key-outfile "~/certs/jobs/private.pem.key"
```

The command returns a response like the following. Save the *certificateArn* value for later use.

```
{
  "certificateArn": "arn:aws:iot:us-
west-2:57EXAMPLE833:cert/76e7e4edb3e52f52334be2f387a06145b2aa4c7fc810f3aea2d92abc227d269",
  "certificateId": "76e7e4edb3e52f5233EXAMPLE7a06145b2aa4c7fc810f3aea2d92abc227d269",
  "certificatePem": "-----BEGIN CERTIFICATE-----
/nMIIDWTCCAkGgAwIBAgI_SHORTENED_FOR_EXAMPLE_Lgn4jfgtS\n-----END CERTIFICATE-----\n",
  "keyPair": {
    "PublicKey": "-----BEGIN PUBLIC KEY-----
\nMIIBIjANBggqhkiG9w0BA_SHORTENED_FOR_EXAMPLE_ImwIDAQAB\n-----END PUBLIC KEY-----\n",
    "PrivateKey": "-----BEGIN RSA PRIVATE KEY-----
\nMIIEowIBAAKCAQE_SHORTENED_FOR_EXAMPLE_T9RoDiukY\n-----END RSA PRIVATE KEY-----\n"
  }
}
```

2. Enter the following commands to set the permissions on the certificate directory and its files.

```
chmod 700 ~/certs/jobs
chmod 644 ~/certs/jobs/*
chmod 600 ~/certs/jobs/private.pem.key
```

3. Run this command to review the permissions on your certificate directories and files.

```
ls -l ~/certs/jobs
```

The output of the command should be the same as what you see here, except the file dates and times will be different.

```
-rw-r--r-- 1 pi pi 1220 Oct 28 13:02 device.pem.crt
-rw----- 1 pi pi 1675 Oct 28 13:02 private.pem.key
-rw-r--r-- 1 pi pi 451 Oct 28 13:02 public.pem.key
```

After you have downloaded the device certificate files to your Raspberry Pi, you're ready to continue to the section called "Provision your Raspberry Pi to demonstrate jobs" (p. 151).

Create AWS IoT resources to demonstrate AWS IoT jobs

Create the AWS IoT resources for this device.

If you are preparing more than one device, this procedure must be performed for each device.

To provision your device in AWS IoT:

In the terminal window on your local host computer that's connected to your Raspberry Pi:

1. Enter the following command to get the address of the device data endpoint for your AWS account.

```
aws iot describe-endpoint --endpoint-type IoT:Data-ATS
```

The endpoint value hasn't changed since the last time you ran this command. Running the command again here makes it easy to find and paste the data endpoint value into the config file used in this tutorial.

The **describe-endpoint** command returns a response like the following. Record the **endpointAddress** value for later use.

```
{  
    "endpointAddress": "a3qjEXAMPLEffp-ats.iot.us-west-2.amazonaws.com"  
}
```

2. Replace **uniqueThingName** with a unique name for your device. If you want to perform this tutorial with multiple devices, give each device its own name. For example, **TestDevice01**, **TestDevice02**, and so on.

Enter this command to create a new AWS IoT thing resource for your Raspberry Pi.

```
aws iot create-thing --thing-name "uniqueThingName"
```

Because an AWS IoT thing resource is a *virtual* representation of your device in the cloud, we can create multiple thing resources in AWS IoT to use for different purposes. They can all be used by the same physical IoT device to represent different aspects of the device.

These tutorials will only use one thing resource at a time per device. This way, in these tutorials, they represent the different demos so that after you create the AWS IoT resources for a demo, you can go back and repeat the demos using the resources you created specifically for each.

If your AWS IoT thing resource was created, the command returns a response like this. Record the **thingArn** value for use later when you create the job to run on this device.

```
{  
    "thingName": "uniqueThingName",  
    "thingArn": "arn:aws:iot:us-west-2:57EXAMPLE833:thing/uniqueThingName",  
    "thingId": "8ea78707-32c3-4f8a-9232-14bEXAMPLEfd"  
}
```

3. In the terminal window:

- a. Open a text editor, such as nano.
- b. Copy this JSON document and paste it into your open text editor.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:Connect"  
            ],  
            "Resource": [  
                "arn:aws:iot:us-west-2:57EXAMPLE833:client/uniqueThingName"  
            ]  
        },  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:Publish"  
            ],  
            "Resource": [  
                "arn:aws:iot:us-west-2:57EXAMPLE833:topic/test/dc/pubtopic",  
                "arn:aws:iot:us-west-2:57EXAMPLE833:topic/$aws/events/job/*",  
                "arn:aws:iot:us-west-2:57EXAMPLE833:topic/$aws/events/jobExecution/*",  
                "arn:aws:iot:us-west-2:57EXAMPLE833:topic/$aws/things/uniqueThingName/jobs/*"  
            ]  
        },  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:Subscribe"  
            ],  
            "Resource": [  
                "arn:aws:iot:us-west-2:57EXAMPLE833:topicfilter/test/dc/subtopic",  
                "arn:aws:iot:us-west-2:57EXAMPLE833:topic/$aws/events/jobExecution/*",  
                "arn:aws:iot:us-west-2:57EXAMPLE833:topicfilter/$aws/things/uniqueThingName/jobs/*"  
            ]  
        },  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:Receive"  
            ],  
            "Resource": [  
                "arn:aws:iot:us-west-2:57EXAMPLE833:topic/test/dc/subtopic",  
                "arn:aws:iot:us-west-2:57EXAMPLE833:topic/$aws/things/uniqueThingName/jobs/*"  
            ]  
        },  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:DescribeJobExecution",  
                "iot:GetPendingJobExecutions",  
                "iot:StartNextPendingJobExecution",  
                "iot:UpdateJobExecution"  
            ],  
            "Resource": [  
                "arn:aws:iot:us-west-2:57EXAMPLE833:topic/$aws/things/uniqueThingName"  
            ]  
        }  
    ]  
}
```

- c. In the editor, in the Resource section of every policy statement, replace **us-west-2:57EXAMPLE833** with your AWS Region, a colon character (:), and your 12-digit AWS account number.
- d. In the editor, in every policy statement, replace **uniqueThingName** with the thing name you gave this thing resource.
- e. Save the file in your text editor as **~/policies/jobs_test_thing_policy.json**.

If you are running this procedure for multiple devices, save the file to this file name on each device.

4. Replace **uniqueThingName** with the thing name for the device, and then run this command to create an AWS IoT policy that is tailored for that device.

```
aws iot create-policy \
--policy-name "JobTestPolicyForuniqueThingName" \
--policy-document "file://~/policies/jobs_test_thing_policy.json"
```

If the policy is created, the command returns a response like this.

```
{  
    "policyName": "JobTestPolicyForuniqueThingName",  
    "policyArn": "arn:aws:iot:us-west-2:57EXAMPLE833:policy/  
JobTestPolicyForuniqueThingName",  
    "policyDocument": "{\n\"Version\": \"2012-10-17\", \n\"Statement\": [\n{\n\"Effect\": \"Allow\", \n\"Action\": [\n\"iot:Connect\"\n], \n\"Resource\": [\n\"arn:aws:iot:us-west-2:57EXAMPLE833:client/PubSubTestThing\"\n]\n}, \n{\n\"Effect\": \"Allow\", \n\"Action\": [\n\"iot:Publish\"\n], \n\"Resource\": [\n\"arn:aws:iot:us-west-2:57EXAMPLE833:topic/test/dc/pubtopic\"\n], \n\"Effect\": \"Allow\", \n\"Action\": [\n\"iot:Subscribe\"\n], \n\"Resource\": [\n\"arn:aws:iot:us-west-2:57EXAMPLE833:topicfilter/test/dc/subtopic\"\n], \n\"Effect\": \"Allow\", \n\"Action\": [\n\"iot:Receive\"\n], \n\"Resource\": [\n\"arn:aws:iot:us-west-2:57EXAMPLE833:topic/test/dc/*\"\n]\n}\n]  
    "policyVersionId": "1"
```

5. Replace **uniqueThingName** with the thing name for the device and **certificateArn** with the certificateArn value you saved earlier in this section for this device, and then run this command to attach the policy to the device certificate.

```
aws iot attach-policy \
--policy-name "JobTestPolicyForuniqueThingName" \
--target "certificateArn"
```

If successful, this command returns nothing.

6. Replace **uniqueThingName** with the thing name for the device, replace **certificateArn** with the certificateArn value that you saved earlier in this section, and then run this command to attach the device certificate to the AWS IoT thing resource.

```
aws iot attach-thing-principal \
--thing-name "uniqueThingName" \
--principal "certificateArn"
```

If successful, this command returns nothing.

After you successfully provisioned your Raspberry Pi, you're ready to repeat this section for another Raspberry Pi in your test or, if all devices have been provisioned, continue to [the section called "Configure the AWS IoT Device Client to run the jobs agent" \(p. 155\)](#).

Configure the AWS IoT Device Client to run the jobs agent

This procedure creates a config file for the AWS IoT Device Client to run the jobs agent.

Note: if you are preparing more than one device, this procedure must be performed on each device.

To create the config file to test the AWS IoT Device Client:

1. In the terminal window on your local host computer that's connected to your Raspberry Pi:
 - a. Open a text editor, such as nano.
 - b. Copy this JSON document and paste it into your open text editor.

```
{  
    "endpoint": "a3qEXAMPLEaffp-ats.iot.us-west-2.amazonaws.com",  
    "cert": "~/certs/jobs/device.pem.crt",  
    "key": "~/certs/jobs/private.pem.key",  
    "root-ca": "~/certs/AmazonRootCA1.pem",  
    "thing-name": "uniqueThingName",  
    "logging": {  
        "enable-sdk-logging": true,  
        "level": "DEBUG",  
        "type": "STDOUT",  
        "file": ""  
    },  
    "jobs": {  
        "enabled": true,  
        "handler-directory": ""  
    },  
    "tunneling": {  
        "enabled": false  
    },  
    "device-defender": {  
        "enabled": false,  
        "interval": 300  
    },  
    "fleet-provisioning": {  
        "enabled": false,  
        "template-name": "",  
        "template-parameters": "",  
        "csr-file": "",  
        "device-key": ""  
    },  
    "samples": {  
        "pub-sub": {  
            "enabled": false,  
            "publish-topic": "",  
            "publish-file": "",  
            "subscribe-topic": "",  
            "subscribe-file": ""  
        }  
    },  
    "config-shadow": {  
        "enabled": false  
    },  
    "sample-shadow": {  
        "enabled": false,  
        "shadow-name": "",  
        "shadow-input-file": "",  
        "shadow-output-file": ""  
    }  
}
```

- c. Replace the *endpoint* value with device data endpoint value for your AWS account that you found in [the section called "Provision your device in AWS IoT Core" \(p. 131\)](#).
 - d. Replace *uniqueThingName* with the thing name that you used for this device.
 - e. Save the file in your text editor as `~/dc-configs/dc-jobs-config.json`.
2. Run this command to set the file permissions of the new config file.

```
chmod 644 ~/dc-configs/dc-jobs-config.json
```

You won't use the **MQTT test client** for this test. While the device will exchange jobs-related MQTT messages with AWS IoT, job progress messages are only exchanged with the device running the job. Because job progress messages are only exchanged with the device running the job, you can't subscribe to them from another device, such as the AWS IoT console.

After you save the config file, you're ready to continue to [the section called "Step 2: Create and run the job in AWS IoT" \(p. 156\)](#).

Step 2: Create and run the job in AWS IoT

The procedures in this section create a job document and an AWS IoT job resource. After you create the job resource, AWS IoT sends the job document to the specified job targets on which a jobs agent applies the job document to the device or client.

Procedures in this section

- [Create and store the job's job document \(p. 156\)](#)
- [Run a job in AWS IoT for one IoT device \(p. 157\)](#)

Create and store the job's job document

This procedure creates a simple job document to include in an AWS IoT job resource. This job document displays "Hello world!" on the job target.

To create and store a job document:

1. Select the Amazon S3 bucket into which you'll save your job document. If you don't have an existing Amazon S3 bucket to use for this, you'll need to create one. For information about how to create Amazon S3 buckets, see the topics in [Getting started with Amazon S3](#).
2. Create and save the job document for this job
 - a. On your local host computer, open a text editor.
 - b. Copy and paste this text into the editor.

```
{  
    "operation": "echo",  
    "args": ["Hello world!"]  
}
```
- c. On the local host computer, save the contents of the editor to a file named **hello-world-job.json**.
- d. Confirm the file was saved correctly. Some text editors automatically append `.txt` to the file name when they save a text file. If your editor appended `.txt` to the file name, correct the file name before proceeding.
3. Replace the *path_to_file* with the path to **hello-world-job.json**, if it's not in your current directory, replace *s3_bucket_name* with the Amazon S3 bucket path to the bucket you selected, and then run this command to put your job document into the Amazon S3 bucket.

```
aws s3api put-object \  
--key hello-world-job.json \  
--body path_to_file/hello-world-job.json --bucket s3_bucket_name
```

The job document URL that identifies the job document that you stored in Amazon S3 is determined by replacing the *s3_bucket_name* and *AWS_region* in the following URL. Record the resulting URL to use later as the *job_document_path*

```
https://s3\_bucket\_name.s3.AWS\_Region.amazonaws.com/hello-world-job.json
```

Note

AWS security prevents you from being able to open this URL outside of your AWS account, for example by using a browser. The URL is used by the AWS IoT jobs engine, which has access to the file, by default. In a production environment, you'll need to make sure that your AWS IoT services have permission to access to the job documents stored in Amazon S3.

After you have saved the job document's URL, continue to [the section called "Run a job in AWS IoT for one IoT device" \(p. 157\)](#).

Run a job in AWS IoT for one IoT device

The procedures in this section start the AWS IoT Device Client on your Raspberry Pi to run the jobs agent on the device to wait for jobs to run. It also creates a job resource in AWS IoT, which will send the job to and run on your IoT device.

Note

This procedure runs a job on only a single device.

To start the jobs agent on your Raspberry Pi:

1. In the terminal window on your local host computer that's connected to your Raspberry Pi, run this command to start the AWS IoT Device Client.

```
cd ~/aws-iot-device-client/build  
./aws-iot-device-client --config-file ~/dc-configs/dc-jobs-config.json
```

2. In the terminal window, confirm that the AWS IoT Device Client displays these messages

```
2021-11-15T18:45:56.708Z [INFO]  {Main.cpp}: Jobs is enabled  
.  
. .  
2021-11-15T18:45:56.708Z [INFO]  {Main.cpp}: Client base has been notified that Jobs  
has started  
2021-11-15T18:45:56.708Z [INFO]  {JobsFeature.cpp}: Running Jobs!  
2021-11-15T18:45:56.708Z [DEBUG] {JobsFeature.cpp}: Attempting to subscribe to  
startNextPendingJobExecution accepted and rejected  
2021-11-15T18:45:56.708Z [DEBUG] {JobsFeature.cpp}: Attempting to subscribe to  
nextJobChanged events  
2021-11-15T18:45:56.708Z [DEBUG] {JobsFeature.cpp}: Attempting to subscribe to  
updateJobExecutionStatusAccepted for jobId +  
2021-11-15T18:45:56.738Z [DEBUG] {JobsFeature.cpp}: Ack received for  
SubscribeToUpdateJobExecutionAccepted with code {0}  
2021-11-15T18:45:56.739Z [DEBUG] {JobsFeature.cpp}: Attempting to subscribe to  
updateJobExecutionStatusRejected for jobId +  
2021-11-15T18:45:56.753Z [DEBUG] {JobsFeature.cpp}: Ack received for  
SubscribeToNextJobChanged with code {0}
```

```
2021-11-15T18:45:56.760Z [DEBUG] {JobsFeature.cpp}: Ack received for
SubscribeToStartNextJobRejected with code {0}
2021-11-15T18:45:56.776Z [DEBUG] {JobsFeature.cpp}: Ack received for
SubscribeToStartNextJobAccepted with code {0}
2021-11-15T18:45:56.776Z [DEBUG] {JobsFeature.cpp}: Ack received for
SubscribeToUpdateJobExecutionRejected with code {0}
2021-11-15T18:45:56.777Z [DEBUG] {JobsFeature.cpp}: Publishing
startNextPendingJobExecutionRequest
2021-11-15T18:45:56.785Z [DEBUG] {JobsFeature.cpp}: Ack received for
StartNextPendingJobPub with code {0}
2021-11-15T18:45:56.785Z [INFO] {JobsFeature.cpp}: No pending jobs are scheduled,
waiting for the next incoming job
```

3. In the terminal window, after you see this message, continue to the next procedure and create the job resource. Note that it might not be the last entry in the list.

```
2021-11-15T18:45:56.785Z [INFO] {JobsFeature.cpp}: No pending jobs are scheduled,
waiting for the next incoming job
```

To create an AWS IoT job resource

1. On your local host computer:
 - a. Replace *job_document_url* with the job document URL from the section called "Create and store the job's job document" (p. 156).
 - b. Replace *thing_arn* with the ARN of the thing resource you created for your device and then run this command.

```
aws iot create-job \
--job-id hello-world-job-1 \
--document-source "job_document_url" \
--targets "thing_arn" \
--target-selection SNAPSHOT
```

If successful, the command returns a result like this one.

```
{  
    "jobArn": "arn:aws:iot:us-west-2:57EXAMPLE833:job/hello-world-job-1",  
    "jobId": "hello-world-job-1"  
}
```

2. In the terminal window, you should see output from the AWS IoT Device Client like this.

```
2021-11-15T18:02:26.688Z [INFO] {JobsFeature.cpp}: No pending jobs are scheduled,
waiting for the next incoming job
2021-11-15T18:10:24.890Z [DEBUG] {JobsFeature.cpp}: Job ids differ
2021-11-15T18:10:24.890Z [INFO] {JobsFeature.cpp}: Executing job: hello-world-job-1
2021-11-15T18:10:24.890Z [DEBUG] {JobsFeature.cpp}: Attempting to update job execution
status!
2021-11-15T18:10:24.890Z [DEBUG] {JobsFeature.cpp}: Not including stdout with the
status details
2021-11-15T18:10:24.890Z [DEBUG] {JobsFeature.cpp}: Not including stderr with the
status details
2021-11-15T18:10:24.890Z [DEBUG] {JobsFeature.cpp}: Assuming executable is in PATH
2021-11-15T18:10:24.890Z [INFO] {JobsFeature.cpp}: About to execute: echo Hello world!
2021-11-15T18:10:24.890Z [DEBUG] {Retry.cpp}: Retryable function starting, it will
retry until success
2021-11-15T18:10:24.890Z [DEBUG] {JobsFeature.cpp}: Created EphemeralPromise for
ClientToken 3TEWba9Xj6 in the updateJobExecution promises map
```

```
2021-11-15T18:10:24.890Z [DEBUG] {JobEngine.cpp}: Child process now running
2021-11-15T18:10:24.890Z [DEBUG] {JobEngine.cpp}: Child process about to call execvp
2021-11-15T18:10:24.890Z [DEBUG] {JobEngine.cpp}: Parent process now running, child PID
is 16737
2021-11-15T18:10:24.891Z [DEBUG] {16737}: Hello world!
2021-11-15T18:10:24.891Z [DEBUG] {JobEngine.cpp}: JobEngine finished waiting for child
process, returning 0
2021-11-15T18:10:24.891Z [INFO]  {JobsFeature.cpp}: Job exited with status: 0
2021-11-15T18:10:24.891Z [INFO]  {JobsFeature.cpp}: Job executed successfully!
2021-11-15T18:10:24.891Z [DEBUG] {JobsFeature.cpp}: Attempting to update job execution
status!
2021-11-15T18:10:24.891Z [DEBUG] {JobsFeature.cpp}: Not including stdout with the
status details
2021-11-15T18:10:24.891Z [DEBUG] {JobsFeature.cpp}: Not including stderr with the
status details
2021-11-15T18:10:24.892Z [DEBUG] {Retry.cpp}: Retryable function starting, it will
retry until success
2021-11-15T18:10:24.892Z [DEBUG] {JobsFeature.cpp}: Created EphemeralPromise for
ClientToken GmQOHTzWGg in the updateJobExecution promises map
2021-11-15T18:10:24.905Z [DEBUG] {JobsFeature.cpp}: Ack received for
PublishUpdateJobExecutionStatus with code {0}
2021-11-15T18:10:24.905Z [DEBUG] {JobsFeature.cpp}: Removing ClientToken 3TEWba9Xj6
from the updateJobExecution promises map
2021-11-15T18:10:24.905Z [DEBUG] {JobsFeature.cpp}: Success response after
UpdateJobExecution for job hello-world-job-1
2021-11-15T18:10:24.917Z [DEBUG] {JobsFeature.cpp}: Ack received for
PublishUpdateJobExecutionStatus with code {0}
2021-11-15T18:10:24.918Z [DEBUG] {JobsFeature.cpp}: Removing ClientToken GmQOHTzWGg
from the updateJobExecution promises map
2021-11-15T18:10:24.918Z [DEBUG] {JobsFeature.cpp}: Success response after
UpdateJobExecution for job hello-world-job-1
2021-11-15T18:10:25.861Z [INFO]  {JobsFeature.cpp}: No pending jobs are scheduled,
waiting for the next incoming job
```

3. While the AWS IoT Device Client is running and waiting for a job, you can submit another job by changing the job-id value and re-running the `create-job` from Step 1.

When you're done running jobs, in the terminal window, enter **^C** (control-C) to stop the AWS IoT Device Client.

Tutorial: Cleaning up after running the AWS IoT Device Client tutorials

The procedures in this tutorial walk you through removing the files and resources you created while completing the tutorials in this learning path.

Procedures in this tutorial

- [Step 1: Cleaning up your devices after building demos with the AWS IoT Device Client \(p. 159\)](#)
- [Step 2: Cleaning up your AWS account after building demos with the AWS IoT Device Client \(p. 161\)](#)

Step 1: Cleaning up your devices after building demos with the AWS IoT Device Client

This tutorial describes two options for how to clean up the microSD card after you built the demos in this learning path. Choose the option that provides the level of security that you need.

Note that cleaning the device's microSD card does not remove any AWS IoT resources that you created. To clean up the AWS IoT resources after you clean the device's microSD card, you should review the

tutorial on the section called “Cleaning up your AWS account after building demos with the AWS IoT Device Client” (p. 161).

Option 1: Cleaning up by rewriting the microSD card

The easiest and most thorough way to clean the microSD card after completing the tutorials in this learning path is to overwrite the microSD card with a saved image file that you created while preparing your device the first time.

This procedure uses the local host computer to write a saved microSD card image to a microSD card.

Note

If your device doesn't use a removable storage medium for its operating system, refer to the procedure for that device.

To write a new image to the microSD card

1. On your local host computer, locate the saved microSD card image that you want to write to your microSD card.
2. Insert your microSD card into the local host computer.
3. Using an SD card imaging tool, write selected image file to the microSD card.
4. After writing the Raspberry Pi OS image to the microSD card, eject the microSD card and safely remove it from the local host computer.

Your microSD card is ready to use.

Option 2: Cleaning up by deleting user directories

To clean the microSD card after completing the tutorials without rewriting the microSD card image, you can delete the user directories individually. This is not as thorough as rewriting the microSD card from a saved image because it does not remove any system files that might have been installed.

If removing the user directories is sufficiently thorough for your needs, you can follow this procedure.

To delete this learning path's user directories from your device

1. Run these commands to delete the user directories, subdirectories, and all their files that were created in this learning path, in the terminal window connected to your device.

Note

After you delete these directories and files, you won't be able to run the demos without completing the tutorials again.

```
rm -Rf ~/dc-configs
rm -Rf ~/policies
rm -Rf ~/messages
rm -Rf ~/certs
rm -Rf ~/.aws-iot-device-client
```

2. Run these commands to delete the application source directories and files, in the terminal window connected to your device.

Note

These commands don't uninstall any programs. They only remove the source files used to build and install them. After you delete these files, the AWS CLI and the AWS IoT Device Client might not work.

```
rm -Rf ~/aws-cli
rm -Rf ~/aws
```

```
rm -rf ~/aws-iot-device-client
```

Step 2: Cleaning up your AWS account after building demos with the AWS IoT Device Client

These procedures help you identify and remove the AWS resources that you created while completing the tutorials in this learning path.

Clean up AWS IoT resources

This procedure helps you identify and remove the AWS IoT resources that you created while completing the tutorials in this learning path.

AWS IoT resources created in this learning path

Tutorial	Thing resource	Policy resource
the section called "Installing and configuring the AWS IoT Device Client" (p. 128)	DevCliTestThing	DevCliTestThingPolicy
the section called "Demonstrate MQTT message communication with the AWS IoT Device Client" (p. 136)	PubSubTestThing	PubSubTestThingPolicy
the section called "Demonstrate remote actions (jobs) with the AWS IoT Device Client" (p. 150)	<i>user defined</i> (there could be more than one)	<i>user defined</i> (there could be more than one)

To delete the AWS IoT resources, follow this procedure for each thing resource that you created

1. Replace *thing_name* with the name of the thing resource you want to delete, and then run this command to list the certificates attached to the thing resource, from the local host computer.

```
aws iot list-thing-principals --thing-name thing_name
```

This command returns a response like this one that lists the certificates that are attached to *thing_name*. In most cases, there will only be one certificate in the list.

```
{  
    " principals": [  
        "arn:aws:iot:us-  
west-2:57EXAMPLE833:cert/23853eea3cf0edc7f8a69c74abeafa27b2b52823cab5b3e156295e94b26ae8ac"  
    ]  
}
```

2. For each certificate listed by the previous command:

- a. Replace *certificate_ID* with the certificate ID from the previous command. The certificate ID is the alphanumeric characters that follow cert/ in the ARN returned by the previous command. Then run this command to deactivate the certificate.

```
aws iot update-certificate --new-status INACTIVE --certificate-id certificate_ID
```

If successful, this command doesn't return anything.

- b. Replace `certificate_ARN` with the certificate ARN from the list of certificates returned earlier, and then run this command to list the policies attached to this certificate.

```
aws iot list-attached-policies --target certificate_ARN
```

This command returns a response like this one that lists the policies attached to the certificate. In most cases, there will only be one policy in the list.

```
{  
    "policies": [  
        {  
            "policyName": "DevCliTestThingPolicy",  
            "policyArn": "arn:aws:iot:us-west-2:57EXAMPLE833:policy/  
DevCliTestThingPolicy"  
        }  
    ]  
}
```

- c. For each policy attached to the certificate:

- i. Replace `policy_name` with the `policyName` value from the previous command, replace `certificate_ARN` with the certificate's ARN, and then run this command to detach the policy from the certificate.

```
aws iot detach-policy --policy-name policy_name --target certificate_ARN
```

If successful, this command doesn't return anything.

- ii. Replace `policy_name` with the `policyName` value, and then run this command to see if the policy is attached to any more certificates.

```
aws iot list-targets-for-policy --policy-name policy_name
```

If the command returns an empty list like this, the policy is not attached to any certificates and you continue to list the policy versions. If there are still certificates attached to the policy, continue with the **detach-thing-principal** step.

```
{  
    "targets": []  
}
```

- iii. Replace `policy_name` with the `policyName` value, and then run this command to check for policy versions. To delete the policy, it must have only one version.

```
aws iot list-policy-versions --policy-name policy_name
```

If the policy has only one version, like this example, you can skip to the **delete-policy** step and delete the policy now.

```
{  
    "policyVersions": [  
        {  
            "versionId": "1",  
            "isDefaultVersion": true,  
            "createDate": "2021-11-18T01:02:46.778000+00:00"  
        }  
    ]  
}
```

```
        }
    ]  
}
```

If the policy has more than one version, like this example, the policy versions with an `isDefaultVersion` value of `false` must be deleted before the policy can be deleted.

```
{
  "policyVersions": [
    {
      "versionId": "2",
      "isDefaultVersion": true,
      "createDate": "2021-11-18T01:52:04.423000+00:00"
    },
    {
      "versionId": "1",
      "isDefaultVersion": false,
      "createDate": "2021-11-18T01:30:18.083000+00:00"
    }
  ]
}
```

If you need to delete a policy version, replace `policy_name` with the `policyName` value, replace `version_ID` with the `versionId` value from the previous command, and then run this command to delete a policy version.

```
aws iot delete-policy-version --policy-name policy_name --policy-version-id version_ID
```

If successful, this command doesn't return anything.

After you delete a policy version, repeat this step until the policy has only one policy version.

- iv. Replace `policy_name` with the `policyName` value, and then run this command to delete the policy.

```
aws iot delete-policy --policy-name policy_name
```

- d. Replace `thing_name` with the thing's name, replace `certificate_ARN` with the certificate's ARN, and then run this command to detach the certificate from the thing resource.

```
aws iot detach-thing-principal --thing-name thing_name --principal certificate_ARN
```

If successful, this command doesn't return anything.

- e. Replace `certificate_ID` with the certificate ID from the previous command. The certificate ID is the alphanumeric characters that follow `cert/` in the ARN returned by the previous command. Then run this command to delete the certificate resource.

```
aws iot delete-certificate --certificate-id certificate_ID
```

If successful, this command doesn't return anything.

3. Replace `thing_name` with the thing's name, and then run this command to delete the thing.

```
aws iot delete-thing --thing-name thing_name
```

If successful, this command doesn't return anything.

Clean up AWS resources

This procedure helps you identify and remove other AWS resources that you created while completing the tutorials in this learning path.

Other AWS resources created in this learning path

Tutorial	Resource type	Resource name or ID
the section called "Demonstrate remote actions (jobs) with the AWS IoT Device Client" (p. 150)	Amazon S3 object	hello-world-job.json
the section called "Demonstrate remote actions (jobs) with the AWS IoT Device Client" (p. 150)	AWS IoT job resources	<i>user defined</i>

To delete the AWS resources created in this learning path

1. To delete the jobs created in this learning path
 - a. Run this command to list the jobs in your AWS account.

```
aws iot list-jobs
```

The command returns a list of the AWS IoT jobs in your AWS account and AWS Region that looks like this.

```
{
  "jobs": [
    {
      "jobArn": "arn:aws:iot:us-west-2:57EXAMPLE833:job/hello-world-job-2",
      "jobId": "hello-world-job-2",
      "targetSelection": "SNAPSHOT",
      "status": "COMPLETED",
      "createdAt": "2021-11-16T23:40:36.825000+00:00",
      "lastUpdatedAt": "2021-11-16T23:40:41.375000+00:00",
      "completedAt": "2021-11-16T23:40:41.375000+00:00"
    },
    {
      "jobArn": "arn:aws:iot:us-west-2:57EXAMPLE833:job/hello-world-job-1",
      "jobId": "hello-world-job-1",
      "targetSelection": "SNAPSHOT",
      "status": "COMPLETED",
      "createdAt": "2021-11-16T23:35:26.381000+00:00",
      "lastUpdatedAt": "2021-11-16T23:35:29.239000+00:00",
      "completedAt": "2021-11-16T23:35:29.239000+00:00"
    }
  ]
}
```

- b. For each job that you recognize from the list as a job you created in this learning path, replace **jobId** with the jobId value of the job to delete, and then run this command to delete an AWS IoT job.

```
aws iot delete-job --job-id jobId
```

If the command is successful, it returns nothing.

2. To delete the job documents you stored in an Amazon S3 bucket in this learning path.
 - a. Replace **bucket** with the name of the bucket you used, and then run this command to list the objects in the Amazon S3 bucket that you used.

```
aws s3api list-objects --bucket bucket
```

The command returns a list of the Amazon S3 objects in bucket that looks like this.

```
{
    "Contents": [
        {
            "Key": "hello-world-job.json",
            "LastModified": "2021-11-18T03:02:12+00:00",
            "ETag": "\"868c8bc3f56b5787964764d4b18ed5ef\"",
            "Size": 54,
            "StorageClass": "STANDARD",
            "Owner": {
                "DisplayName": "EXAMPLE",
                "ID": "e9e3d6ec1EXAMPLEf5bfb5e6bd0a2b6ed03884d1ed392a82ad011c144736a4ee"
            }
        },
        {
            "Key": "iot_job_firmware_update.json",
            "LastModified": "2021-04-13T21:57:07+00:00",
            "ETag": "\"7c68c591949391791ecf625253658c61\"",
            "Size": 66,
            "StorageClass": "STANDARD",
            "Owner": {
                "DisplayName": "EXAMPLE",
                "ID": "e9e3d6ec1EXAMPLEf5bfb5e6bd0a2b6ed03884d1ed392a82ad011c144736a4ee"
            }
        },
        {
            "Key": "order66.json",
            "LastModified": "2021-04-13T21:57:07+00:00",
            "ETag": "\"bca60d5380b88e1a70cc27d321cabab72\"",
            "Size": 29,
            "StorageClass": "STANDARD",
            "Owner": {
                "DisplayName": "EXAMPLE",
                "ID": "e9e3d6ec1EXAMPLEf5bfb5e6bd0a2b6ed03884d1ed392a82ad011c144736a4ee"
            }
        }
    ]
}
```

- b. For each object that you recognize from the list as an object you created in this learning path, replace **bucket** with the bucket name and **key** with key value of the object to delete, and then run this command to delete an Amazon S3 object.

```
aws s3api delete-object --bucket bucket --key key
```

If the command is successful, it returns nothing.

After you delete all the AWS resources and objects that you created while completing this learning path, you can start over and repeat the tutorials.

Building solutions with the AWS IoT Device SDKs

The tutorials in this section help walk you through the steps to develop an IoT solution that can be deployed to a production environment using AWS IoT.

These tutorials can take more time to complete than those in the section on [the section called “Building demos with the AWS IoT Device Client” \(p. 116\)](#) because they use the AWS IoT Device SDKs and explain the concepts being applied in more detail to help you create secure and reliable solutions.

Start building solutions with the AWS IoT Device SDKs

TODO: Add content for iot-sdk-tutorial-overview here.

Topics

- [Tutorial: Connecting a device to AWS IoT Core by using the AWS IoT Device SDK \(p. 166\)](#)
- [Creating AWS IoT rules to route device data to other services \(p. 182\)](#)
- [Retaining the device state while the device is offline \(p. 211\)](#)
- [Tutorial: Creating a custom authorizer for AWS IoT Core \(p. 231\)](#)
- [Tutorial: Monitoring soil moisture with AWS IoT and Raspberry Pi \(p. 242\)](#)

Tutorial: Connecting a device to AWS IoT Core by using the AWS IoT Device SDK

This tutorial demonstrates how to connect a device to AWS IoT Core so that it can send and receive data to and from AWS IoT. After you complete this tutorial, your device will be configured to connect to AWS IoT Core and you'll understand how devices communicate with AWS IoT.

In this tutorial, you will:

1. [the section called “Prepare your device for AWS IoT” \(p. 167\)](#)
2. [the section called “Review the MQTT protocol” \(p. 167\)](#)
3. [the section called “Review the pubsub.py Device SDK sample app” \(p. 168\)](#)
4. [the section called “Connect your device and communicate with AWS IoT Core” \(p. 173\)](#)
5. [the section called “Review the results” \(p. 178\)](#)

This tutorial takes about an hour to complete.

Before you start this tutorial, make sure that you have:

- [Completed Getting started with AWS IoT Core \(p. 16\)](#)

In the section of that tutorial where you must [the section called “Configure your device” \(p. 38\)](#), select the [the section called “Connect a Raspberry Pi or another device” \(p. 53\)](#) option for your device and use the Python language options to configure your device.

Keep open the terminal window you use in that tutorial because you'll also use it in this tutorial.

- **A device that can run the AWS IoT Device SDK v2 for Python.**

This tutorial shows how to connect a device to AWS IoT Core by using Python code examples, which require a relatively powerful device.

If you are working with resource-constrained devices, these code examples might not work on them. In that case, you might have more success by [the section called “Using the AWS IoT Device SDK for Embedded C” \(p. 179\)](#) tutorial.

Prepare your device for AWS IoT

In [Getting started with AWS IoT Core \(p. 16\)](#), you prepared your device and AWS account so they could communicate. This section reviews the aspects of that preparation that apply to any device connection with AWS IoT Core.

For a device to connect to AWS IoT Core:

1. You must have an **AWS account**.

The procedure in [Set up your AWS account \(p. 17\)](#) describes how to create an AWS account if you don't already have one.

2. In that account, you must have the following **AWS IoT resources** defined for the device in your AWS account and Region.

The procedure in [Create AWS IoT resources \(p. 34\)](#) describes how to create these resources for the device in your AWS account and Region.

- **A device certificate** registered with AWS IoT and activated to authenticate the device.

The certificate is often created with, and attached to, an **AWS IoT thing object**. While a thing object is not required for a device to connect to AWS IoT, it makes additional AWS IoT features available to the device.

- **A policy** attached to the device certificate that authorizes it to connect to AWS IoT Core and perform all the actions that you want it to.

3. An **internet connection** that can access your AWS account's device endpoints.

The device endpoints are described in [AWS IoT device data and service endpoints \(p. 74\)](#) and can be seen in the [settings page of the AWS IoT console](#).

4. **Communication software** such as the AWS IoT Device SDKs provide. This tutorial uses the [AWS IoT Device SDK v2 for Python](#).

Review the MQTT protocol

Before we talk about the sample app, it helps to understand the MQTT protocol. The MQTT protocol offers some advantages over other network communication protocols, such as HTTP, which makes it a popular choice for IoT devices. This section reviews the key aspects of MQTT that apply to this tutorial. For information about how MQTT compares to HTTP, see [Choosing a protocol for your device communication \(p. 79\)](#).

MQTT uses a publish/subscribe communication model

The MQTT protocol uses a publish/subscribe communication model with its host. This model differs from the request/response model that HTTP uses. With MQTT, devices establish a session with the host that is identified by a unique client ID. To send data, devices publish messages identified by topics to a message broker in the host. To receive messages from the message broker, devices subscribe to topics by sending topic filters in subscription requests to the message broker.

MQTT supports persistent sessions

The message broker receives messages from devices and publishes messages to devices that have subscribed to them. With [persistent sessions \(p. 82\)](#)—sessions that remain active even when the initiating device is disconnected—devices can retrieve messages that were published while they were disconnected. On the device side, MQTT supports Quality of Service levels ([QoS \(p. 82\)](#)) that ensure the host receives messages sent by the device.

Review the pubsub.py Device SDK sample app

This section reviews the `pubsub.py` sample app from the [AWS IoT Device SDK v2 for Python](#) used in this tutorial. Here, we'll review how it connects to AWS IoT Core to publish and subscribe to MQTT messages. The next section presents some exercises to help you explore how a device connects and communicates with AWS IoT Core.

The `pubsub.py` sample app demonstrates these aspects of an MQTT connection with AWS IoT Core:

- [Communication protocols \(p. 168\)](#)
- [Persistent sessions \(p. 171\)](#)
- [Quality of Service \(p. 171\)](#)
- [Message publish \(p. 172\)](#)
- [Message subscription \(p. 172\)](#)
- [Device disconnection and reconnection \(p. 173\)](#)

Communication protocols

The `pubsub.py` sample demonstrates an MQTT connection using the MQTT and MQTT over WSS protocols. The [AWS common runtime \(AWS CRT\)](#) library provides the low-level communication protocol support and is included with the AWS IoT Device SDK v2 for Python.

MQTT

The `pubsub.py` sample calls `mtls_from_path` (shown here) in the `mqtt_connection_builder` to establish a connection with AWS IoT Core by using the MQTT protocol. `mtls_from_path` uses X.509 certificates and TLS v1.2 to authenticate the device. The AWS CRT library handles the lower-level details of that connection.

```
mqtt_connection = mqtt_connection_builder.mtls_from_path(  
    endpoint=args.endpoint,  
    cert_filepath=args.cert,  
    pri_key_filepath=args.key,  
    ca_filepath=args.root_ca,  
    client_bootstrap=client_bootstrap,  
    on_connection_interrupted=on_connection_interrupted,  
    on_connection_resumed=on_connection_resumed,  
    client_id=args.client_id,  
    clean_session=False,  
    keep_alive_secs=6  
)
```

`endpoint`

Your AWS account's IoT device endpoint

In the sample app, this value is passed in from the command line.

`cert_filepath`

The path to the device's certificate file

In the sample app, this value is passed in from the command line.

`pri_key_filepath`

The path to the device's private key file that was created with its certificate file

In the sample app, this value is passed in from the command line.

`ca_filepath`

The path to the Root CA file. Required only if the MQTT server uses a certificate that's not already in your trust store.

In the sample app, this value is passed in from the command line.

`client_bootstrap`

The common runtime object that handles socket communication activities

In the sample app, this object is instantiated before the call to

`mqtt_connection_builder.mtls_from_path`.

`on_connection_interrupted`, `on_connection_resumed`

The callback functions to call when the device's connection is interrupted and resumed

`client_id`

The ID that uniquely identifies this device in the AWS Region

In the sample app, this value is passed in from the command line.

`clean_session`

Whether to start a new persistent session, or, if one is present, reconnect to an existing one

`keep_alive_secs`

The keep alive value, in seconds, to send in the CONNECT request. A ping will automatically be sent at this interval. If the server doesn't receive a ping after 1.5 times this value, it assumes that the connection is lost.

MQTT over WSS

The `pubsub.py` sample calls `websockets_with_default_aws_signing` (shown here) in the `mqtt_connection_builder` to establish a connection with AWS IoT Core using the MQTT protocol over WSS. `websockets_with_default_aws_signing` creates an MQTT connection over WSS using [Signature V4](#) to authenticate the device.

```
mqtt_connection = mqtt_connection_builder.websockets_with_default_aws_signing(
    endpoint=args.endpoint,
    client_bootstrap=client_bootstrap,
    region=args.signing_region,
    credentials_provider=credentials_provider,
    websocket_proxy_options=proxy_options,
    ca_filepath=args.root_ca,
    on_connection_interrupted=on_connection_interrupted,
    on_connection_resumed=on_connection_resumed,
    client_id=args.client_id,
    clean_session=False,
    keep_alive_secs=6
)
```

`endpoint`

Your AWS account's IoT device endpoint

In the sample app, this value is passed in from the command line.

`client_bootstrap`

The common runtime object that handles socket communication activities

In the sample app, this object is instantiated before the call to

`mqtt_connection_builder.websockets_with_default_aws_signing`.

`region`

The AWS signing Region used by Signature V4 authentication. In `pubsub.py`, it passes the parameter entered in the command line.

In the sample app, this value is passed in from the command line.

`credentials_provider`

The AWS credentials provided to use for authentication

In the sample app, this object is instantiated before the call to

`mqtt_connection_builder.websockets_with_default_aws_signing`.

`websocket_proxy_options`

HTTP proxy options, if using a proxy host

In the sample app, this value is initialized before the call to

`mqtt_connection_builder.websockets_with_default_aws_signing`.

`ca_filepath`

The path to the Root CA file. Required only if the MQTT server uses a certificate that's not already in your trust store.

In the sample app, this value is passed in from the command line.

`on_connection_interrupted`, `on_connection_resumed`

The callback functions to call when the device's connection is interrupted and resumed

`client_id`

The ID that uniquely identifies this device in the AWS Region.

In the sample app, this value is passed in from the command line.

`clean_session`

Whether to start a new persistent session, or, if one is present, reconnect to an existing one

`keep_alive_secs`

The keep alive value, in seconds, to send in the CONNECT request. A ping will automatically be sent at this interval. If the server doesn't receive a ping after 1.5 times this value, it assumes the connection is lost.

HTTPS

What about HTTPS? AWS IoT Core supports devices that publish HTTPS requests. From a programming perspective, devices send HTTPS requests to AWS IoT Core as would any other application. For an example of a Python program that sends an HTTP message from a device, see the [HTTPS code example \(p. 91\)](#) using Python's `requests` library. This example sends a message to AWS IoT Core using HTTPS such that AWS IoT Core interprets it as an MQTT message.

While AWS IoT Core supports HTTPS requests from devices, be sure to review the information about [Choosing a protocol for your device communication \(p. 79\)](#) so that you can make an informed decision on which protocol to use for your device communications.

Persistent sessions

In the sample app, setting the `clean_session` parameter to `False` indicates that the connection should be persistent. In practice, this means that the connection opened by this call reconnects to an existing persistent session, if one exists. Otherwise, it creates and connects to a new persistent session.

With a persistent session, messages that are sent to the device are stored by the message broker while the device is not connected. When a device reconnects to a persistent session, the message broker sends to the device any stored messages to which it has subscribed.

Without a persistent session, the device will not receive messages that are sent while the device isn't connected. Which option to use depends on your application and whether messages that occur while a device is not connected must be communicated. For more information, see [Using MQTT persistent sessions \(p. 82\)](#).

Quality of Service

When the device publishes and subscribes to messages, the preferred Quality of Service (QoS) can be set. AWS IoT supports QoS levels 0 and 1 for publish and subscribe operations. For more information about QoS levels in AWS IoT, see [MQTT Quality of Service \(QoS\) options \(p. 82\)](#).

The AWS CRT runtime for Python defines these constants for the QoS levels that it supports:

Python Quality of Service levels

MQTT QoS level	Python symbolic value used by SDK	Description
QoS level 0	<code>mqtt.QoS.AT_MOST_ONCE</code>	Only one attempt to send the message will be made, whether it is received or not. The message might not be sent at all, for example, if the device is not connected or there's a network error.
QoS level 1	<code>mqtt.QoS.AT_LEAST_ONCE</code>	The message is sent repeatedly until a <code>PUBACK</code> acknowledgement is received.

In the sample app, the publish and subscribe requests are made with a QoS level of 1 (`mqtt.QoS.AT_LEAST_ONCE`).

- **QoS on publish**

When a device publishes a message with QoS level 1, it sends the message repeatedly until it receives a `PUBACK` response from the message broker. If the device isn't connected, the message is queued to be sent after it reconnects.

- **QoS on subscribe**

When a device subscribes to a message with QoS level 1, the message broker saves the messages to which the device is subscribed until they can be sent to the device. The message broker resends the messages until it receives a `PUBACK` response from the device.

Message publish

After successfully establishing a connection to AWS IoT Core, devices can publish messages. The `pubsub.py` sample does this by calling the `publish` operation of the `mqtt_connection` object.

```
mqtt_connection.publish(  
    topic=args.topic,  
    payload=message,  
    qos= mqtt.QoS.AT_LEAST_ONCE  
)
```

topic

The message's topic name that identifies the message

In the sample app, this is passed in from the command line.

payload

The message payload formatted as a string (for example, a JSON document)

In the sample app, this is passed in from the command line.

A JSON document is a common payload format, and one that is recognized by other AWS IoT services; however, the data format of the message payload can be anything that the publishers and subscribers agree upon. Other AWS IoT services, however, only recognize JSON, and CBOR, in some cases, for most operations.

qos

The QoS level for this message

Message subscription

To receive messages from AWS IoT and other services and devices, devices subscribe to those messages by their topic name. Devices can subscribe to individual messages by specifying a [topic name \(p. 93\)](#), and to a group of messages by specifying a [topic filter \(p. 94\)](#), which can include wild card characters. The `pubsub.py` sample uses the code shown here to subscribe to messages and register the callback functions to process the message after it's received.

```
subscribe_future, packet_id = mqtt_connection.subscribe(  
    topic=args.topic,  
    qos= mqtt.QoS.AT_LEAST_ONCE,  
    callback=on_message_received  
)  
subscribe_result = subscribe_future.result()
```

topic

The topic to subscribe to. This can be a topic name or a topic filter.

In the sample app, this is passed in from the command line.

qos

Whether the message broker should store these messages while the device is disconnected.

A value of `mqtt.QoS.AT_LEAST_ONCE` (QoS level 1), requires a persistent session to be specified (`clean_session=False`) when the connection is created.

callback

The function to call to process the subscribed message.

The `mqtt_connection.subscribe` function returns a future and a packet ID. If the subscription request was initiated successfully, the packet ID returned is greater than 0. To make sure that the subscription was received and registered by the message broker, you must wait for the result of the asynchronous operation to return, as shown in the code example.

The callback function

The callback in the `pubsub.py` sample processes the subscribed messages as the device receives them.

```
def on_message_received(topic, payload, **kwargs):
    print("Received message from topic '{}': {}".format(topic, payload))
    global received_count
    received_count += 1
    if received_count == args.count:
        received_all_event.set()
```

topic

The message's topic

This is the specific topic name of the message received, even if you subscribed to a topic filter.

payload

The message payload

The format for this is application specific.

kwargs

Possible additional arguments as described in [mqtt.Connection.subscribe](#).

In the `pubsub.py` sample, `on_message_received` only displays the topic and its payload. It also counts the messages received to end the program after the limit is reached.

Your app would evaluate the topic and the payload to determine what actions to perform.

Device disconnection and reconnection

The `pubsub.py` sample includes callback functions that are called when the device is disconnected and when the connection is re-established. What actions your device takes on these events is application specific.

When a device connects for the first time, it must subscribe to topics to receive. If a device's session is present when it reconnects, its subscriptions are restored, and any stored messages from those subscriptions are sent to the device after it reconnects.

If a device's session no longer exists when it reconnects, it must resubscribe to its subscriptions. Persistent sessions have a limited lifetime and can expire when the device is disconnected for too long.

Connect your device and communicate with AWS IoT Core

This section presents some exercises to help you explore different aspects of connecting your device to AWS IoT Core. For these exercises, you'll use the [MQTT test client](#) in the AWS IoT console to see what your device publishes and to publish messages to your device. These exercises use the `pubsub.py` sample from the [AWS IoT Device SDK v2 for Python](#) and build on your experience with [Getting started with AWS IoT Core \(p. 16\)](#) tutorials.

In this section, you'll:

- [Subscribe to wild card topic filters \(p. 174\)](#)
- [Process topic filter subscriptions \(p. 175\)](#)
- [Publish messages from your device \(p. 177\)](#)

For these exercises, you'll start from the `pubsub.py` sample program.

Note

These exercises assume that you completed the [Getting started with AWS IoT Core \(p. 16\)](#) tutorials and use the terminal window for your device from that tutorial.

Subscribe to wild card topic filters

In this exercise, you'll modify the command line used to call `pubsub.py` to subscribe to a wild card topic filter and process the messages received based on the message's topic.

Exercise procedure

For this exercise, imagine that your device contains a temperature control and a light control. It uses these topic names to identify the messages about them.

1. Before starting the exercise, try running this command from the [Getting started with AWS IoT Core \(p. 16\)](#) tutorials on your device to make sure that everything is ready for the exercise.

```
cd ~/aws-iot-device-sdk-python-v2/samples
python3 pubsub.py --topic topic_1 --root-ca ~/certs/Amazon-root-CA-1.pem --cert ~/certs/device.pem.crt --key ~/certs/private.pem.key --endpoint your-iot-endpoint
```

You should see the same output as you saw in the [Getting started tutorial \(p. 57\)](#).

2. For this exercise, change these command line parameters.

Action	Command line parameter	Effect
add	--message ""	Configure <code>pubsub.py</code> to listen only
add	--count 2	End the program after receiving two messages
change	--topic device/+/details	Define the topic filter to subscribe to

Making these changes to the initial command line results in this command line. Enter this command in the terminal window for your device.

```
python3 pubsub.py --message "" --count 2 --topic device/+/details --root-ca ~/certs/
Amazon-root-CA-1.pem --cert ~/certs/device.pem.crt --key ~/certs/private.pem.key --
endpoint your-iot-endpoint
```

The program should display something like this:

```
Connecting to a3qexamplesffp-ats.iot.us-west-2.amazonaws.com with client ID
'test-24d7cdcc-cc01-458c-8488-2d05849691e1'...
Connected!
```

```
Subscribing to topic 'device/+/details'...
Subscribed with QoS.AT LEAST_ONCE
Waiting for all messages to be received...
```

If you see something like this on your terminal, your device is ready and listening for messages where the topic names start with `topic-1/` and end with `/detail`. So, let's test that.

3. Here are a couple of messages that your device might receive.

Topic name	Message payload
device/temp/details	{ "desiredTemp": 20, "currentTemp": 15 }
device/light/details	{ "desiredLight": 100, "currentLight": 50 }

4. Using the MQTT test client in the AWS IoT console, send the messages described in the previous step to your device.
 - a. Open the [MQTT test client](#) in the AWS IoT console.
 - b. In **Subscribe to a topic**, in the **Subscription topic field**, enter the topic filter: `device/+/details`, and then choose **Subscribe to topic**.
 - c. In the **Subscriptions** column of the MQTT test client, choose `device/+/details`.
 - d. For each of the topics in the preceding table, do the following in the MQTT test client:
 1. In **Publish**, enter the value from the **Topic name** column in the table.
 2. In the message payload field below the topic name, enter the value from the **Message payload** column in the table.
 3. Watch the terminal window where `pubsub.py` is running and, in the MQTT test client, choose **Publish to topic**.

You should see that the message was received by `pubsub.py` in the terminal window.

Exercise result

With this, `pubsub.py`, subscribed to the messages using a wild card topic filter, received them, and displayed them in the terminal window. Notice how you subscribed to a single topic filter, and the callback function was called to process messages having two distinct topics.

Process topic filter subscriptions

Building on the previous exercise, modify the `pubsub.py` sample app to evaluate the message topics and process the subscribed messages based on the topic.

Exercise procedure

To evaluate the message topic

1. Copy `pubsub.py` to `pubsub2.py`.
2. Open `pubsub2.py` in your favorite text editor or IDE.
3. In `pubsub2.py`, find the `on_message_received` function.
4. In `on_message_received`, insert the following code after the line that starts with `print("Received message` and before the line that starts with `global received_count`.

```
topic_parsed = False
if "/" in topic:
    parsed_topic = topic.split("/")
    if len(parsed_topic) == 3:
        # this topic has the correct format
        if (parsed_topic[0] == 'device') and (parsed_topic[2] == 'details'):
            # this is a topic we care about, so check the 2nd element
            if (parsed_topic[1] == 'temp'):
                print("Received temperature request: {}".format(payload))
                topic_parsed = True
            if (parsed_topic[1] == 'light'):
                print("Received light request: {}".format(payload))
                topic_parsed = True
    if not topic_parsed:
        print("Unrecognized message topic.")
```

5. Save your changes and run the modified program by using this command line.

```
python3 pubsub2.py --message "" --count 2 --topic device/+/details --root-ca ~/certs/
Amazon-root-CA-1.pem --cert ~/certs/device.pem.crt --key ~/certs/private.pem.key --
endpoint your-iot-endpoint
```

6. In the AWS IoT console, open the [MQTT test client](#).
7. In **Subscribe to a topic**, in the **Subscription topic field**, enter the topic filter: **device/+/details**, and then choose **Subscribe to topic**.
8. In the **Subscriptions** column of the MQTT test client, choose **device/+/details**.
9. For each of the topics in this table, do the following in the MQTT test client:

Topic name	Message payload
device/temp/details	{ "desiredTemp": 20, "currentTemp": 15 }
device/light/details	{ "desiredLight": 100, "currentLight": 50 }

1. In **Publish**, enter the value from the **Topic name** column in the table.
2. In the message payload field below the topic name, enter the value from the **Message payload** column in the table.
3. Watch the terminal window where pubsub.py is running and, in the MQTT test client, choose **Publish to topic**.

You should see that the message was received by pubsub.py in the terminal window.

You should see something similar to this in your terminal window.

```
Connecting to a3qexamplesffp-ats.iot.us-west-2.amazonaws.com with client ID 'test-
af794be0-7542-45a0-b0af-0b0ea7474517'...
Connected!
Subscribing to topic 'device/+/details'...
Subscribed with QoS.AT_LEAST_ONCE
Waiting for all messages to be received...
Received message from topic 'device/light/details': b'{ "desiredLight": 100,
"currentLight": 50 }'
Received light request: b'{ "desiredLight": 100, "currentLight": 50 }'
```

```
Received message from topic 'device/temp/details': b'{ "desiredTemp": 20, "currentTemp": 15 }'  
Received temperature request: b'{ "desiredTemp": 20, "currentTemp": 15 }'  
2 message(s) received.  
Disconnecting...  
Disconnected!
```

Exercise result

In this exercise, you added code so the sample app would recognize and process multiple messages in the callback function. With this, your device could receive messages and act on them.

Another way for your device to receive and process multiple messages is to subscribe to different messages separately and assign each subscription to its own callback function.

Publish messages from your device

You can use the pubsub.py sample app to publish messages from your device. While it will publish messages as it is, the messages can't be read as JSON documents. This exercise modifies the sample app to be able to publish JSON documents in the message payload that can be read by AWS IoT Core.

Exercise procedure

In this exercise, the following message will be sent with the device/data topic.

```
{  
    "timestamp": 1601048303,  
    "sensorId": 28,  
    "sensorData": [  
        {  
            "sensorName": "Wind speed",  
            "sensorValue": 34.2211224  
        }  
    ]  
}
```

To prepare your MQTT test client to monitor the messages from this exercise

1. In **Subscribe to a topic**, in the **Subscription topic field**, enter the topic filter: **device/data**, and then choose **Subscribe to topic**.
2. In the **Subscriptions** column of the MQTT test client, choose **device/data**.
3. Keep the MQTT test client window open to wait for messages from your device.

To send JSON documents with the pubsub.py sample app

1. On your device, copy pubsub.py to pubsub3.py.
2. Edit pubsub3.py to change how it formats the messages it publishes.
 - a. Open pubsub3.py in a text editor.
 - b. Locate this line of code:

```
message = "{} [{}]".format(args.message, publish_count)
```
 - c. Change it to:

```
message = "{}".format(args.message)
```
 - d. Save your changes.

3. On your device, run this command to send the message two times.

```
python3 pubsub3.py --root-ca ~/certs/Amazon-root-CA-1.pem --cert ~/certs/device.pem.crt --key ~/certs/private.pem.key --topic device/data --count 2 --message '{"timestamp":1601048303,"sensorId":28,"sensorData":[{"sensorName":"Wind speed","sensorValue":34.2211224}]}' --endpoint your-iot-endpoint
```

4. In the MQTT test client, check to see that it has interpreted and formatted the JSON document in the message payload, such as this:



By default, `pubsub3.py` also subscribes to the messages it sends. You should see that it received the messages in the app's output. The terminal window should look something like this.

```
Connecting to a3qEXAMPLEsffp-ats.iot.us-west-2.amazonaws.com with client ID  
'test-5cff18ae-1e92-4c38-a9d4-7b9771afc52f'...  
Connected!  
Subscribing to topic 'device/data'...  
Subscribed with QoS.AT_LEAST_ONCE  
Sending 2 message(s)  
Publishing message to topic 'device/data':  
{"timestamp":1601048303,"sensorId":28,"sensorData":[{"sensorName":"Wind speed","sensorValue":34.2211224}]}  
Received message from topic 'device/data':  
b'{"timestamp":1601048303,"sensorId":28,"sensorData":[{"sensorName":"Wind speed","sensorValue":34.2211224}]}'  
Publishing message to topic 'device/data':  
{"timestamp":1601048303,"sensorId":28,"sensorData":[{"sensorName":"Wind speed","sensorValue":34.2211224}]}  
Received message from topic 'device/data':  
b'{"timestamp":1601048303,"sensorId":28,"sensorData":[{"sensorName":"Wind speed","sensorValue":34.2211224}]}'  
2 message(s) received.  
Disconnecting...  
Disconnected!
```

Exercise result

With this, your device can generate messages to send to AWS IoT Core to test basic connectivity and provide device messages for AWS IoT Core to process. For example, you could use this app to send test data from your device to test AWS IoT rule actions.

Review the results

The examples in this tutorial gave you hands-on experience with the basics of how devices can communicate with AWS IoT Core—a fundamental part of your AWS IoT solution. When your devices are able to communicate with AWS IoT Core, they can pass messages to AWS services and other devices

on which they can act. Likewise, AWS services and other devices can process information that results in messages sent back to your devices.

When you are ready to explore AWS IoT Core further, try these tutorials:

- the section called “[Sending an Amazon SNS notification](#)” (p. 190)
- the section called “[Storing device data in a DynamoDB table](#)” (p. 197)
- the section called “[Formatting a notification by using an AWS Lambda function](#)” (p. 203)

Tutorial: Using the AWS IoT Device SDK for Embedded C

This section describes how to run the AWS IoT Device SDK for Embedded C.

Procedures in this section

- [Step1: Install the AWS IoT Device SDK for Embedded C](#) (p. 179)
- [Step 2: Configure the sample app](#) (p. 179)
- [Step 3: Build and run the sample application](#) (p. 181)

Step1: Install the AWS IoT Device SDK for Embedded C

The AWS IoT Device SDK for Embedded C is generally targeted at resource constrained devices that require an optimized C language runtime. You can use the SDK on any operating system and host it on any processor type (for example, MCUs and MPUs). If you have more memory and processing resources available, we recommend that you use one of the higher order AWS IoT Device and Mobile SDKs (for example, C++, Java, JavaScript, and Python).

In general, the AWS IoT Device SDK for Embedded C is intended for systems that use MCUs or low-end MPUs that run embedded operating systems. For the programming example in this section, we assume your device uses Linux.

Example

1. Download the AWS IoT Device SDK for Embedded C to your device from [GitHub](#).

```
git clone https://github.com/aws/aws-iot-device-sdk-embedded-c.git --recurse-submodules
```

This creates a directory named `aws-iot-device-sdk-embedded-c` in the current directory.

2. Navigate to that directory and checkout the latest release. Please see github.com/aws/aws-iot-device-sdk-embedded-C/tags for the latest release tag.

```
cd aws-iot-device-sdk-embedded-C
git checkout latest-release-tag
```

3. Install OpenSSL version 1.1.0 or later. The OpenSSL development libraries are usually called "libssl-dev" or "openssl-devel" when installed through a package manager.

```
sudo apt-get install libssl-dev
```

Step 2: Configure the sample app

The AWS IoT Device SDK for Embedded C includes sample applications for you to try. For simplicity, this tutorial uses the `mqtt_demo_mutual_auth` application, that illustrates how to connect to the AWS IoT Core message broker and subscribe and publish to MQTT topics.

1. Copy the certificate and private key you created in [Getting started with AWS IoT Core \(p. 16\)](#) into the build/bin/certificates directory.

Note

Device and root CA certificates are subject to expiration or revocation. If these certificates expire or are revoked, you must copy a new CA certificate or private key and device certificate onto your device.

2. You must configure the sample with your personal AWS IoT Core endpoint, private key, certificate, and root CA certificate. Navigate to the aws-iot-device-sdk-embedded-c/demos/mqtt/mqtt_demo_mutual_auth directory.

If you have the AWS CLI installed, you can use this command to find your account's endpoint URL.

```
aws iot describe-endpoint --endpoint-type iot:Data-ATS
```

If you don't have the AWS CLI installed, open your [AWS IoT console](#). From the navigation pane, choose **Manage**, and then choose **Things**. Choose the IoT thing for your device, and then choose **Interact**. Your endpoint is displayed in the **HTTPS** section of the thing details page.

3. Open the demo_config.h file and update the values for the following:

AWS_IOT_ENDPOINT

Your personal endpoint.

CLIENT_CERT_PATH

Your certificate file path, for example certificates/device.pem.crt".

CLIENT_PRIVATE_KEY_PATH

Your private key file name, for example certificates/private.pem.key.

For example:

```
// Get from demo_config.h
// =====
#define AWS_IOT_ENDPOINT      "my-endpoint-ats.iot.us-east-1.amazonaws.com"
#define AWS_MQTT_PORT          8883
#define CLIENT_IDENTIFIER       "testclient"
#define ROOT_CA_CERT_PATH      "certificates/AmazonRootCA1.crt"
#define CLIENT_CERT_PATH        "certificates/my-device-cert.pem.crt"
#define CLIENT_PRIVATE_KEY_PATH "certificates/my-device-private-key.pem.key"
// =====
```

4. Check to see if you have CMake installed on your device by using this command.

```
cmake --version
```

If you see the version information for the compiler, you can continue to the next section.

If you get an error or don't see any information, then you'll need to install the cmake package using this command.

```
sudo apt-get install cmake
```

Run the **cmake --version** command again and confirm that CMake has been installed and that you are ready to continue.

5. Check to see if you have the development tools installed on your device by using this command.

```
gcc --version
```

If you see the version information for the compiler, you can continue to the next section.

If you get an error or don't see any compiler information, you'll need to install the `build-essential` package using this command.

```
sudo apt-get install build-essential
```

Run the `gcc --version` command again and confirm that the build tools have been installed and that you are ready to continue.

Step 3: Build and run the sample application

To run the AWS IoT Device SDK for Embedded C sample applications

1. Navigate to `aws-iot-device-sdk-embedded-c` and create a build directory.

```
mkdir build && cd build
```

2. Enter the following CMake command to generate the Makefiles needed to build.

```
cmake ..
```

3. Enter the following command to build the executable app file.

```
make
```

4. Run the `mqtt_demo_mutual_auth` app with this command.

```
cd bin  
./mqtt_demo_mutual_auth
```

You should see output similar to the following:

```
[INFO] [DEMO] [mqtt_demo_mutual_auth.c:584] Establishing a TLS session to a2zk5tjv9x07ct-ats.iot.us-west-2.amazonaws.com:8883.  
[INFO] [DEMO] [mqtt_demo_mutual_auth.c:1264] Creating an MQTT connection to a2zk5tjv9x07ct-ats.iot.us-west-2.amazonaws.com.  
[INFO] [MQTT] [core_mqtt.c:855] Packet received. ReceivedBytes=2.  
[INFO] [MQTT] [core_mqtt_serializer.c:970] CONNACK session present bit not set.  
[INFO] [MQTT] [core_mqtt_serializer.c:912] Connection accepted.  
[INFO] [MQTT] [core_mqtt.c:1526] Received MQTT CONNACK successfully from broker.  
[INFO] [MQTT] [core_mqtt.c:1792] MQTT connection established with the broker.  
[INFO] [DEMO] [mqtt_demo_mutual_auth.c:1033] MQTT connection successfully established with broker.  
  
[INFO] [DEMO] [mqtt_demo_mutual_auth.c:1296] A clean MQTT connection is established. Cleaning up all the stored outgoing publishes.  
  
[INFO] [DEMO] [mqtt_demo_mutual_auth.c:1314] Subscribing to the MQTT topic testclient/example/topic.  
[INFO] [DEMO] [mqtt_demo_mutual_auth.c:1097] SUBSCRIBE sent for topic testclient/example/topic to broker.  
  
[INFO] [MQTT] [core_mqtt.c:855] Packet received. ReceivedBytes=3.  
[INFO] [DEMO] [mqtt_demo_mutual_auth.c:921] Subscribed to the topic testclient/example/topic. with maximum QoS 1.  
  
[INFO] [DEMO] [mqtt_demo_mutual_auth.c:1358] Sending Publish to the MQTT topic testclient/example/topic.  
[INFO] [DEMO] [mqtt_demo_mutual_auth.c:1195] PUBLISH sent for topic testclient/example/topic to broker with packet ID 2.  
  
[INFO] [MQTT] [core_mqtt.c:855] Packet received. ReceivedBytes=2.  
[INFO] [MQTT] [core_mqtt.c:1126] Ack packet deserialized with result: MQTTSuccess.  
[INFO] [MQTT] [core_mqtt.c:1139] State record updated. New state=MQTTPublishDone.  
[INFO] [DEMO] [mqtt_demo_mutual_auth.c:946] PUBACK received for packet id 2.  
  
[INFO] [DEMO] [mqtt_demo_mutual_auth.c:672] Cleaned up outgoing publish packet with packet id 2.  
  
[INFO] [MQTT] [core_mqtt.c:855] Packet received. ReceivedBytes=40.  
[INFO] [MQTT] [core_mqtt.c:1015] De-serialized incoming PUBLISH packet: DeserializerResult=MQTTSuccess.
```

Your device is now connected to AWS IoT using the AWS IoT Device SDK for Embedded C.

You can also use the AWS IoT console to view the MQTT messages that the sample app is publishing. For information about how to use the MQTT client in the [AWS IoT console](#), see the section called “[View MQTT messages with the AWS IoT MQTT client](#)” (p. 62).

Creating AWS IoT rules to route device data to other services

These tutorials show you how to create and test AWS IoT rules using some of the more common rule actions.

AWS IoT rules send data from your devices to other AWS services. They listen for specific MQTT messages, format the data in the message payloads, and send the result to other AWS services.

We recommend that you try these in the order they are shown here, even if your goal is to create a rule that uses a Lambda function or something more complex. The tutorials are presented in order from basic to complex. They present new concepts incrementally to help you learn the concepts you can use to create the rule actions that don't have a specific tutorial.

Note

AWS IoT rules help you send the data from your IoT devices to other AWS services. To do that successfully, however, you need a working knowledge of the other services where you want to send data. While these tutorials provide the necessary information to complete the tasks, you might find it helpful to learn more about the services you want to send data to before you use them in your solution. A detailed explanation of the other AWS services is outside of the scope of these tutorials.

Tutorial scenario overview

The scenario for these tutorials is that of a weather sensor device that periodically publishes its data. There are many such sensor devices in this imaginary system. The tutorials in this section, however, focus on a single device while showing how you might accommodate multiple sensors.

The tutorials in this section show you how to use AWS IoT rules to do the following tasks with this imaginary system of weather sensor devices.

- [Tutorial: Republishing an MQTT message \(p. 184\)](#)

This tutorial shows how to republish an MQTT message received from the weather sensors as a message that contains only the sensor ID and the temperature value. It uses only AWS IoT Core services and demonstrates a simple SQL query and how to use the MQTT client to test your rule.

- [Tutorial: Sending an Amazon SNS notification \(p. 190\)](#)

This tutorial shows how to send an SNS message when a value from a weather sensor device exceeds a specific value. It builds on the concepts presented in the previous tutorial and adds how to work with another AWS service, the [Amazon Simple Notification Service](#) (Amazon SNS).

If you're new to Amazon SNS, review its [Getting started](#) exercises before you start this tutorial.

- [Tutorial: Storing device data in a DynamoDB table \(p. 197\)](#)

This tutorial shows how to store the data from the weather sensor devices in a database table. It uses the rule query statement and substitution templates to format the message data for the destination service, [Amazon DynamoDB](#).

If you're new to DynamoDB, review its [Getting started](#) exercises before you start this tutorial.

- [Tutorial: Formatting a notification by using an AWS Lambda function \(p. 203\)](#)

This tutorial shows how to call a Lambda function to reformat the device data and then send it as a text message. It adds a Python script and AWS SDK functions in an [AWS Lambda](#) function to format with the message payload data from the weather sensor devices and send a text message.

If you're new to Lambda, review its [Getting started](#) exercises before you start this tutorial.

AWS IoT rule overview

All of these tutorials create AWS IoT rules.

For an AWS IoT rule to send the data from a device to another AWS service, it uses:

- A rule query statement that consists of:
 - A SQL SELECT clause that selects and formats the data from the message payload
 - A topic filter (the FROM object in the rule query statement) that identifies the messages to use
 - An optional conditional statement (a SQL WHERE clause) that specifies specific conditions on which to act
- At least one rule action

Devices publish messages to MQTT topics. The topic filter in the SQL SELECT statement identifies the MQTT topics to apply the rule to. The fields specified in the SQL SELECT statement format the data from the incoming MQTT message payload for use by the rule's actions. For a complete list of rule actions, see [AWS IoT Rule Actions \(p. 450\)](#).

Tutorials in this section

- [Tutorial: Republishing an MQTT message \(p. 184\)](#)
- [Tutorial: Sending an Amazon SNS notification \(p. 190\)](#)
- [Tutorial: Storing device data in a DynamoDB table \(p. 197\)](#)
- [Tutorial: Formatting a notification by using an AWS Lambda function \(p. 203\)](#)

Tutorial: Republishing an MQTT message

This tutorial demonstrates how to create an AWS IoT rule that publishes an MQTT message when a specified MQTT message is received. The incoming message payload can be modified by the rule before it's published. This makes it possible to create messages that are tailored to specific applications without the need to alter your device or its firmware. You can also use the filtering aspect of a rule to publish messages only when a specific condition is met.

The messages republished by a rule act like messages sent by any other AWS IoT device or client. Devices can subscribe to the republished messages the same way they can subscribe to any other MQTT message topic.

What you'll learn in this tutorial:

- How to use simple SQL queries and functions in a rule query statement
- How to use the MQTT client to test an AWS IoT rule

This tutorial takes about 30 minutes to complete.

In this tutorial, you'll:

- [Review MQTT topics and AWS IoT rules \(p. 184\)](#)
- [Step 1: Create an AWS IoT rule to republish an MQTT message \(p. 185\)](#)
- [Step 2: Test your new rule \(p. 186\)](#)
- [Step 3: Review the results and next steps \(p. 189\)](#)

Before you start this tutorial, make sure that you have:

- [Set up your AWS account \(p. 17\)](#)

You'll need your AWS account and AWS IoT console to complete this tutorial.

- [Reviewed **View MQTT messages with the AWS IoT MQTT client \(p. 62\)**](#)

Be sure you can use the MQTT client to subscribe and publish to a topic. You'll use the MQTT client to test your new rule in this procedure.

Review MQTT topics and AWS IoT rules

Before talking about AWS IoT rules, it helps to understand the MQTT protocol. In IoT solutions, the MQTT protocol offers some advantages over other network communication protocols, such as HTTP, which makes it a popular choice for use by IoT devices. This section reviews the key aspects of MQTT as they apply to this tutorial. For information about how MQTT compares to HTTP, see [Choosing a protocol for your device communication \(p. 79\)](#).

MQTT protocol

The MQTT protocol uses a publish/subscribe communication model with its host. To send data, devices publish messages that are identified by topics to the AWS IoT message broker. To receive messages from the message broker, devices subscribe to the topics they will receive by sending topic filters in subscription requests to the message broker. The AWS IoT rules engine receives MQTT messages from the message broker.

AWS IoT rules

AWS IoT rules consist of a rule query statement and one or more rule actions. When the AWS IoT rules engine receives an MQTT message, these elements act on the message as follows.

- **Rule query statement**

The rule's query statement describes the MQTT topics to use, interprets the data from the message payload, and formats the data as described by a SQL statement that is similar to statements used by popular SQL databases. The result of the query statement is the data that is sent to the rule's actions.

- **Rule action**

Each rule action in a rule acts on the data that results from the rule's query statement. AWS IoT supports [many rule actions \(p. 450\)](#). In this tutorial, however, you'll concentrate on the [Republish \(p. 509\)](#) rule action, which publishes the result of the query statement as an MQTT message with a specific topic.

Step 1: Create an AWS IoT rule to republish an MQTT message

The AWS IoT rule that you'll create in this tutorial subscribes to the `device/device_id/data` MQTT topics where `device_id` is the ID of the device that sent the message. These topics are described by a [topic filter \(p. 94\)](#) as `device/+/data`, where the `+` is a wildcard character that matches any string between the two forward slash characters.

When the rule receives a message from a matching topic, it republishes the `device_id` and `temperature` values as a new MQTT message with the `device/data/temp` topic.

For example, the payload of an MQTT message with the `device/22/data` topic looks like this:

```
{  
    "temperature": 28,  
    "humidity": 80,  
    "barometer": 1013,  
    "wind": {  
        "velocity": 22,  
        "bearing": 255  
    }  
}
```

The rule takes the `temperature` value from the message payload, and the `device_id` from the topic, and republishes them as an MQTT message with the `device/data/temp` topic and a message payload that looks like this:

```
{  
    "device_id": "22",  
    "temperature": 28  
}
```

With this rule, devices that only need the device's ID and the temperature data subscribe to the `device/data/temp` topic to receive only that information.

To create a rule that republishes an MQTT message

1. Open [the Rules hub of the AWS IoT console](#).
2. In **Rules**, choose **Create** and start creating your new rule.
3. In the top part of **Create a rule**:
 - a. In **Name**, enter the rule's name. For this tutorial, name it `republish_temp`.

Remember that a rule name must be unique within your Account and Region, and it can't have any spaces. We've used an underscore character in this name to separate the two words in the rule's name.

- b. In **Description**, describe the rule.

A meaningful description helps you remember what this rule does and why you created it. The description can be as long as needed, so be as detailed as possible.

4. In **Rule query statement** of **Create a rule**:

- a. In **Using SQL version**, select **2016-03-23**.
- b. In the **Rule query statement** edit box, enter the statement:

```
SELECT topic(2) as device_id, temperature FROM 'device/+/data'
```

This statement:

- Listens for MQTT messages with a topic that matches the `device/+/data` topic filter.
- Selects the second element from the topic string and assigns it to the `device_id` field.
- Selects the value `temperature` field from the message payload and assigns it to the `temperature` field.

5. In **Set one or more actions**:

- a. To open up the list of rule actions for this rule, choose **Add action**.
- b. In **Select an action**, choose **Republish a message to an AWS IoT topic**.
- c. At the bottom of the action list, choose **Configure action** to open the selected action's configuration page.

6. In **Configure action**:

- a. In **Topic**, enter `device/data/temp`. This is the MQTT topic of the message that this rule will publish.
- b. In **Quality of Service**, choose **0 - The message is delivered zero or more times**.
- c. In **Choose or create a role to grant AWS IoT access to perform this action**:
 - i. Choose **Create Role**. The **Create a new role** dialog box opens.
 - ii. Enter a name that describes the new role. In this tutorial, use `republish_role`.

When you create a new role, the correct policies to perform the rule action are created and attached to the new role. If you change the topic of this rule action or use this role in another rule action, you must update the policy for that role to authorize the new topic or action. To update an existing role, choose **Update role** in this section.

- iii. Choose **Create Role** to create the role and close the dialog box.
- d. Choose **Add action** to add the action to the rule and return to the **Create a rule** page.

7. The **Republish a message to an AWS IoT topic** action is now listed in **Set one or more actions**.

In the new action's tile, below **Republish a message to an AWS IoT topic**, you can see the topic to which your republish action will publish.

This is the only rule action you'll add to this rule.

8. In **Create a rule**, scroll down to the bottom and choose **Create rule** to create the rule and complete this step.

Step 2: Test your new rule

To test your new rule, you'll use the MQTT client to publish and subscribe to the MQTT messages used by this rule.

Open the [MQTT client in the AWS IoT console](#) in a new window. This will let you edit the rule without losing the configuration of your MQTT client. The MQTT client does not retain any subscriptions or message logs if you leave it to go to another page in the console.

To use the MQTT client to test your rule

1. In the [MQTT client in the AWS IoT console](#), subscribe to the input topics, in this case, `device/+/
data`.

- a. In the MQTT client, under **Subscriptions**, choose **Subscribe to a topic**.
- b. In **Subscription topic**, enter the topic of the input topic filter, `device/+/
data`.
- c. Keep the rest of the fields at their default settings.
- d. Choose **Subscribe to topic**.

In the **Subscriptions** column, under **Publish to a topic**, `device/+/
data` appears.

2. Subscribe to the topic that your rule will publish: `device/
data/temp`.

- a. Under **Subscriptions**, choose **Subscribe to a topic** again, and in **Subscription topic**, enter the topic of the republished message, `device/
data/temp`.
- b. Keep the rest of the fields at their default settings.
- c. Choose **Subscribe to topic**.

In the **Subscriptions** column, under `device/+/
data`, `device/
data/temp` appears.

3. Publish a message to the input topic with a specific device ID, `device/22/
data`. You can't publish to MQTT topics that contain wildcard characters.

- a. In the MQTT client, under **Subscriptions**, choose **Publish to topic**.
- b. In the **Publish** field, enter the input topic name, `device/22/
data`.
- c. Copy the sample data shown here and, in the edit box below the topic name, paste the sample data.

```
{  
    "temperature": 28,  
    "humidity": 80,  
    "barometer": 1013,  
    "wind": {  
        "velocity": 22,  
        "bearing": 255  
    }  
}
```

- d. To send your MQTT message, choose **Publish to topic**.
4. Review the messages that were sent.

- a. In the MQTT client, under **Subscriptions**, there is a green dot next to the two topics to which you subscribed earlier.

The green dots indicate that one or more new messages have been received since the last time you looked at them.

- b. Under **Subscriptions**, choose `device/+/
data` to check that the message payload matches what you just published and looks like this:

```
{  
    "temperature": 28,  
    "humidity": 80,  
    "barometer": 1013,  
    "wind": {
```

```
        "velocity": 22,
        "bearing": 255
    }
}
```

- c. Under **Subscriptions**, choose **device/data/temp** to check that your republished message payload looks like this:

```
{
    "device_id": "22",
    "temperature": 28
}
```

Notice that the `device_id` value is a quoted string and the `temperature` value is numeric. This is because the `topic()` function extracted the string from the input message's topic name while the `temperature` value uses the numeric value from the input message's payload.

If you want to make the `device_id` value a numeric value, replace `topic(2)` in the rule query statement with:

```
cast(topic(2) AS DECIMAL)
```

Note that casting the `topic(2)` value to a numeric value will only work if that part of the topic contains only numeric characters.

5. If you see that the correct message was published to the **device/data/temp** topic, then your rule worked. See what more you can learn about the Republish rule action in the next section.

If you don't see that the correct message was published to either the **device/+data** or **device/data/temp** topics, check the troubleshooting tips.

Troubleshooting your Republish message rule

Here are some things to check in case you're not seeing the results you expect.

- **You got an error banner**

If an error appeared when you published the input message, correct that error first. The following steps might help you correct that error.

- **You don't see the input message in the MQTT client**

Every time you publish your input message to the `device/22/data` topic, that message should appear in the MQTT client if you subscribed to the `device/+data` topic filter as described in the procedure.

Things to check

- **Check the topic filter you subscribed to**

If you subscribed to the input message topic as described in the procedure, you should see a copy of the input message every time you publish it.

If you don't see the message, check the topic name you subscribed to and compare it to the topic to which you published. Topic names are case sensitive and the topic to which you subscribed must be identical to the topic to which you published the message payload.

- **Check the message publish function**

In the MQTT client, under **Subscriptions**, choose **device/+data**, check the topic of the publish message, and then choose **Publish to topic**. You should see the message payload from the edit box below the topic appear in the message list.

- **You don't see your republished message in the MQTT client**

For your rule to work, it must have the correct policy that authorizes it to receive and republish a message and it must receive the message.

Things to check

- **Check the AWS Region of your MQTT client and the rule that you created**

The console in which you're running the MQTT client must be in the same AWS Region as the rule you created.

- **Check the input message topic in the rule query statement**

For the rule to work, it must receive a message with the topic name that matches the topic filter in the FROM clause of the rule query statement.

Check the spelling of the topic filter in the rule query statement with that of the topic in the MQTT client. Topic names are case sensitive and the message's topic must match the topic filter in the rule query statement.

- **Check the contents of the input message payload**

For the rule to work, it must find the data field in the message payload that is declared in the SELECT statement.

Check the spelling of the temperature field in the rule query statement with that of the message payload in the MQTT client. Field names are case sensitive and the temperature field in the rule query statement must be identical to the temperature field in the message payload.

Make sure that the JSON document in the message payload is correctly formatted. If the JSON has any errors, such as a missing comma, the rule will not be able to read it.

- **Check the republished message topic in the rule action**

The topic to which the Republish rule action publishes the new message must match the topic to which you subscribed in the MQTT client.

Open the rule you created in the console and check the topic to which the rule action will republish the message.

- **Check the role being used by the rule**

The rule action must have permission to receive the original topic and publish the new topic.

The policies that authorize the rule to receive message data and republish it are specific to the topics used. If you change the topic used to republish the message data, you must update the rule action's role to update its policy to match the current topic.

If you suspect this is the problem, edit the Republish rule action and create a new role. New roles created by the rule action receive the authorizations necessary to perform these actions.

Step 3: Review the results and next steps

In this tutorial

- You used a simple SQL query and a couple of functions in a rule query statement to produce a new MQTT message.

- You created a rule that republished that new message.
- You used the MQTT client to test your AWS IoT rule.

Next steps

After you republish a few messages with this rule, try experimenting with it to see how changing some aspects of the tutorial affect the republished message. Here are some ideas to get you started.

- Change the *device_id* in the input message's topic and observe the effect in the republished message payload.
- Change the fields selected in the rule query statement and observe the effect in the republished message payload.
- Try the next tutorial in this series and learn how to [Tutorial: Sending an Amazon SNS notification \(p. 190\)](#).

The Republish rule action used in this tutorial can also help you debug rule query statements. For example, you can add this action to a rule to see how its rule query statement is formatting the data used by its rule actions.

Tutorial: Sending an Amazon SNS notification

This tutorial demonstrates how to create an AWS IoT rule that sends MQTT message data to an Amazon SNS topic so that it can be sent as an SMS text message.

In this tutorial, you create a rule that sends message data from a weather sensor to all subscribers of an Amazon SNS topic, whenever the temperature exceeds the value set in the rule. The rule detects when the reported temperature exceeds the value set by the rule, and then creates a new message payload that includes only the device ID, the reported temperature, and the temperature limit that was exceeded. The rule sends the new message payload as a JSON document to an SNS topic, which notifies all subscribers to the SNS topic.

What you'll learn in this tutorial:

- How to create and test an Amazon SNS notification
- How to call an Amazon SNS notification from an AWS IoT rule
- How to use simple SQL queries and functions in a rule query statement
- How to use the MQTT client to test an AWS IoT rule

This tutorial takes about 30 minutes to complete.

In this tutorial, you'll:

- [Step 1: Create an Amazon SNS topic that sends an SMS text message \(p. 191\)](#)
- [Step 2: Create an AWS IoT rule to send the text message \(p. 192\)](#)
- [Step 3: Test the AWS IoT rule and Amazon SNS notification \(p. 193\)](#)
- [Step 4: Review the results and next steps \(p. 196\)](#)

Before you start this tutorial, make sure that you have:

- [Set up your AWS account \(p. 17\)](#)

You'll need your AWS account and AWS IoT console to complete this tutorial.

- [Reviewed View MQTT messages with the AWS IoT MQTT client \(p. 62\)](#)

Be sure you can use the MQTT client to subscribe and publish to a topic. You'll use the MQTT client to test your new rule in this procedure.

- Reviewed the [Amazon Simple Notification Service](#)

If you haven't used Amazon SNS before, review [Setting up access for Amazon SNS](#). If you've already completed other AWS IoT tutorials, your AWS account should already be configured correctly.

Step 1: Create an Amazon SNS topic that sends an SMS text message

To create an Amazon SNS topic that sends an SMS text message

1. Create an Amazon SNS topic.

- Sign in to the [Amazon SNS console](#).
- In the left navigation pane, choose **Topics**.
- On the **Topics** page, choose **Create topic**.
- In **Details**, choose the **Standard** type. By default, the console creates a FIFO topic.
- In **Name**, enter the SNS topic name. For this tutorial, enter **high_temp_notice**.
- Scroll to the end of the page and choose **Create topic**.

The console opens the new topic's **Details** page.

2. Create an Amazon SNS subscription.

Note

The phone number that you use in this subscription might incur text messaging charges from the messages you will send in this tutorial.

- In the **high_temp_notice** topic's details page, choose **Create subscription**.
- In **Create subscription**, in the **Details** section, in the **Protocol** list, choose **SMS**.
- In **Endpoint**, enter the number of a phone that can receive text messages. Be sure to enter it such that it starts with a +, includes the country and area code, and doesn't include any other punctuation characters.
- Choose **Create subscription**.

3. Test the Amazon SNS notification.

- In the [Amazon SNS console](#), in the left navigation pane, choose **Topics**.
- To open the topic's details page, in **Topics**, in the list of topics, choose **high_temp_notice**.
- To open the **Publish message to topic** page, in the **high_temp_notice** details page, choose **Publish message**.
- In **Publish message to topic**, in the **Message body** section, in **Message body to send to the endpoint**, enter a short message.
- Scroll down to the bottom of the page and choose **Publish message**.
- On the phone with the number you used earlier when creating the subscription, confirm that the message was received.

If you did not receive the test message, double check the phone number and your phone's settings.

Make sure you can publish test messages from the [Amazon SNS console](#) before you continue the tutorial.

Step 2: Create an AWS IoT rule to send the text message

The AWS IoT rule that you'll create in this tutorial subscribes to the device/*device_id*/data MQTT topics where *device_id* is the ID of the device that sent the message. These topics are described in a topic filter as device/+data, where the + is a wildcard character that matches any string between the two forward slash characters. This rule also tests the value of the temperature field in the message payload.

When the rule receives a message from a matching topic, it takes the *device_id* from the topic name, the temperature value from the message payload, and adds a constant value for the limit it's testing, and sends these values as a JSON document to an Amazon SNS notification topic.

For example, an MQTT message from weather sensor device number 32 uses the device/32/data topic and has a message payload that looks like this:

```
{  
    "temperature": 38,  
    "humidity": 80,  
    "barometer": 1013,  
    "wind": {  
        "velocity": 22,  
        "bearing": 255  
    }  
}
```

The rule's rule query statement takes the temperature value from the message payload, the *device_id* from the topic name, and adds the constant max_temperature value to send a message payload that looks like this to the Amazon SNS topic:

```
{  
    "device_id": "32",  
    "reported_temperature": 38,  
    "max_temperature": 30  
}
```

To create an AWS IoT rule to detect an over-limit temperature value and create the data to send to the Amazon SNS topic

1. Open [the Rules hub of the AWS IoT console](#).
2. If this is your first rule, choose **Create**, or **Create a rule**.
3. In **Create a rule**:

- a. In **Name**, enter **temp_limit_notify**.

Remember that a rule name must be unique within your AWS account and Region, and it can't have any spaces. We've used an underscore character in this name to separate the words in the rule's name.

- b. In **Description**, describe the rule.

A meaningful description makes it easier to remember what this rule does and why you created it. The description can be as long as needed, so be as detailed as possible.

4. In **Rule query statement of Create a rule**:
 - a. In **Using SQL version**, select **2016-03-23**.
 - b. In the **Rule query statement** edit box, enter the statement:

```
SELECT topic(2) as device_id,
```

```
temperature as reported_temperature,  
30 as max_temperature  
FROM 'device/+/data'  
WHERE temperature > 30
```

This statement:

- Listens for MQTT messages with a topic that matches the device/+/data topic filter and that have a temperature value greater than 30.
 - Selects the second element from the topic string and assigns it to the device_id field.
 - Selects the value temperature field from the message payload and assigns it to the reported_temperature field.
 - Creates a constant value 30 to represent the limit value and assigns it to the max_temperature field.
5. To open up the list of rule actions for this rule, in **Set one or more actions**, choose **Add action**.
6. In **Select an action**, choose **Send a message as an SNS push notification**.
7. To open the selected action's configuration page, at the bottom of the action list, choose **Configure action**.
8. In **Configure action**:
- a. In **SNS target**, choose **Select**, find your SNS topic named **high_temp_notice**, and choose **Select**.
 - b. In **Message format**, choose **RAW**.
 - c. In **Choose or create a role to grant AWS IoT access to perform this action**, choose **Create Role**.
 - d. In **Create a new role**, in **Name**, enter a unique name for the new role. For this tutorial, use **sns_rule_role**.
 - e. Choose **Create role**.

If you're repeating this tutorial or reusing an existing role, choose **Update role** before continuing. This updates the role's policy document to work with the SNS target.

9. Choose **Add action** and return to the **Create a rule** page.

In the new action's tile, below **Send a message as an SNS push notification**, you can see the SNS topic that your rule will call.

This is the only rule action you'll add to this rule.

10. To create the rule and complete this step, in **Create a rule**, scroll down to the bottom and choose **Create rule**.

Step 3: Test the AWS IoT rule and Amazon SNS notification

To test your new rule, you'll use the MQTT client to publish and subscribe to the MQTT messages used by this rule.

Open the [MQTT client in the AWS IoT console](#) in a new window. This will let you edit the rule without losing the configuration of your MQTT client. If you leave the MQTT client to go to another page in the console, it won't retain any subscriptions or message logs.

To use the MQTT client to test your rule

1. In the [MQTT client in the AWS IoT console](#), subscribe to the input topics, in this case, device/+/data.
 - a. In the MQTT client, under **Subscriptions**, choose **Subscribe to a topic**.
 - b. In **Subscription topic**, enter the topic of the input topic filter, **device/+/data**.

- c. Keep the rest of the fields at their default settings.
- d. Choose **Subscribe to topic**.

In the **Subscriptions** column, under **Publish to a topic**, `device/+data` appears.

2. Publish a message to the input topic with a specific device ID, `device/32/data`. You can't publish to MQTT topics that contain wildcard characters.
 - a. In the MQTT client, under **Subscriptions**, choose **Publish to topic**.
 - b. In the **Publish** field, enter the input topic name, `device/32/data`.
 - c. Copy the sample data shown here and, in the edit box below the topic name, paste the sample data.

```
{  
    "temperature": 38,  
    "humidity": 80,  
    "barometer": 1013,  
    "wind": {  
        "velocity": 22,  
        "bearing": 255  
    }  
}
```

- d. Choose **Publish to topic** to publish your MQTT message.
3. Confirm that the text message was sent.
 - a. In the MQTT client, under **Subscriptions**, there is a green dot next to the topic to which you subscribed earlier.

The green dot indicates that one or more new messages have been received since the last time you looked at them.

 - b. Under **Subscriptions**, choose `device/+data` to check that the message payload matches what you just published and looks like this:

```
{  
    "temperature": 38,  
    "humidity": 80,  
    "barometer": 1013,  
    "wind": {  
        "velocity": 22,  
        "bearing": 255  
    }  
}
```

- c. Check the phone that you used to subscribe to the SNS topic and confirm the contents of the message payload look like this:

```
{"device_id": "32", "reported_temperature": 38, "max_temperature": 30}
```

Notice that the `device_id` value is a quoted string and the `temperature` value is numeric. This is because the `topic()` function extracted the string from the input message's topic name while the `temperature` value uses the numeric value from the input message's payload.

If you want to make the `device_id` value a numeric value, replace `topic(2)` in the rule query statement with:

```
cast(topic(2) AS DECIMAL)
```

Note that casting the `topic(2)` value to a numeric, `DECIMAL` value will only work if that part of the topic contains only numeric characters.

4. Try sending an MQTT message in which the temperature does not exceed the limit.
 - a. In the MQTT client, under **Subscriptions**, choose **Publish to topic**.
 - b. In the **Publish** field, enter the input topic name, `device/33/data`.
 - c. Copy the sample data shown here and, in the edit box below the topic name, paste the sample data.

```
{  
    "temperature": 28,  
    "humidity": 80,  
    "barometer": 1013,  
    "wind": {  
        "velocity": 22,  
        "bearing": 255  
    }  
}
```

- d. To send your MQTT message, choose **Publish to topic**.

You should see the message that you sent in the `device/+data` subscription. However, because the temperature value is below the max temperature in the rule query statement, you shouldn't receive a text message.

If you don't see the correct behavior, check the troubleshooting tips.

Troubleshooting your SNS message rule

Here are some things to check, in case you're not seeing the results you expect.

- **You got an error banner**

If an error appeared when you published the input message, correct that error first. The following steps might help you correct that error.

- **You don't see the input message in the MQTT client**

Every time you publish your input message to the `device/22/data` topic, that message should appear in the MQTT client, if you subscribed to the `device/+data` topic filter as described in the procedure.

Things to check

- **Check the topic filter you subscribed to**

If you subscribed to the input message topic as described in the procedure, you should see a copy of the input message every time you publish it.

If you don't see the message, check the topic name you subscribed to and compare it to the topic to which you published. Topic names are case sensitive and the topic to which you subscribed must be identical to the topic to which you published the message payload.

- **Check the message publish function**

In the MQTT client, under **Subscriptions**, choose `device/+data`, check the topic of the publish message, and then choose **Publish to topic**. You should see the message payload from the edit box below the topic appear in the message list.

- **You don't receive an SMS message**

For your rule to work, it must have the correct policy that authorizes it to receive a message and send an SNS notification, and it must receive the message.

Things to check

- **Check the AWS Region of your MQTT client and the rule that you created**

The console in which you're running the MQTT client must be in the same AWS Region as the rule you created.

- **Check that the temperature value in the message payload exceeds the test threshold**

If the temperature value is less than or equal to 30, as defined in the rule query statement, the rule will not perform any of its actions.

- **Check the input message topic in the rule query statement**

For the rule to work, it must receive a message with the topic name that matches the topic filter in the FROM clause of the rule query statement.

Check the spelling of the topic filter in the rule query statement with that of the topic in the MQTT client. Topic names are case sensitive and the message's topic must match the topic filter in the rule query statement.

- **Check the contents of the input message payload**

For the rule to work, it must find the data field in the message payload that is declared in the SELECT statement.

Check the spelling of the temperature field in the rule query statement with that of the message payload in the MQTT client. Field names are case sensitive and the temperature field in the rule query statement must be identical to the temperature field in the message payload.

Make sure that the JSON document in the message payload is correctly formatted. If the JSON has any errors, such as a missing comma, the rule will not be able to read it.

- **Check the republished message topic in the rule action**

The topic to which the Republish rule action publishes the new message must match the topic to which you subscribed in the MQTT client.

Open the rule you created in the console and check the topic to which the rule action will republish the message.

- **Check the role being used by the rule**

The rule action must have permission to receive the original topic and publish the new topic.

The policies that authorize the rule to receive message data and republish it are specific to the topics used. If you change the topic used to republish the message data, you must update the rule action's role to update its policy to match the current topic.

If you suspect this is the problem, edit the Republish rule action and create a new role. New roles created by the rule action receive the authorizations necessary to perform these actions.

Step 4: Review the results and next steps

In this tutorial:

-
- You created and tested an Amazon SNS notification topic and subscription.

- You used a simple SQL query and functions in a rule query statement to create a new message for your notification.
- You created an AWS IoT rule to send an Amazon SNS notification that used your customized message payload.
- You used the MQTT client to test your AWS IoT rule.

Next steps

After you send a few text messages with this rule, try experimenting with it to see how changing some aspects of the tutorial affect the message and when it's sent. Here are some ideas to get you started.

- Change the `device_id` in the input message's topic and observe the effect in the text message contents.
- Change the fields selected in the rule query statement and observe the effect in the text message contents.
- Change the test in the rule query statement to test for a minimum temperature instead of a maximum temperature. Remember to change the name of `max_temperature`!
- Add a republish rule action to send an MQTT message when an SNS notification is sent.
- Try the next tutorial in this series and learn how to [Tutorial: Storing device data in a DynamoDB table \(p. 197\)](#).

Tutorial: Storing device data in a DynamoDB table

This tutorial demonstrates how to create an AWS IoT rule that sends message data to a DynamoDB table.

In this tutorial, you create a rule that sends message data from an imaginary weather sensor device to a DynamoDB table. The rule formats the data from many weather sensors such that they can be added to a single database table.

What you'll learn in this tutorial

- How to create a DynamoDB table
- How to send message data to a DynamoDB table from an AWS IoT rule
- How to use substitution templates in an AWS IoT rule
- How to use simple SQL queries and functions in a rule query statement
- How to use the MQTT client to test an AWS IoT rule

This tutorial takes about 30 minutes to complete.

In this tutorial, you'll:

- [Step 1: Create the DynamoDB table for this tutorial \(p. 198\)](#)
- [Step 2: Create an AWS IoT rule to send data to the DynamoDB table \(p. 198\)](#)
- [Step 3: Test the AWS IoT rule and DynamoDB table \(p. 200\)](#)
- [Step 4: Review the results and next steps \(p. 202\)](#)

Before you start this tutorial, make sure that you have:

- [Set up your AWS account \(p. 17\)](#)

You'll need your AWS account and AWS IoT console to complete this tutorial.

- Reviewed [View MQTT messages with the AWS IoT MQTT client \(p. 62\)](#)

Be sure you can use the MQTT client to subscribe and publish to a topic. You'll use the MQTT client to test your new rule in this procedure.

- Reviewed the [Amazon DynamoDB overview](#)

If you've not used DynamoDB before, review [Getting Started with DynamoDB](#) to become familiar with the basic concepts and operations of DynamoDB.

Step 1: Create the DynamoDB table for this tutorial

In this tutorial, you'll create a DynamoDB table with these attributes to record the data from the imaginary weather sensor devices:

- `sample_time` is a primary key and describes the time the sample was recorded.
- `device_id` is a sort key and describes the device that provided the sample
- `device_data` is the data received from the device and formatted by the rule query statement

To create the DynamoDB table for this tutorial

1. Open the [DynamoDB console](#), and then choose **Create table**.
2. In **Create DynamoDB table**:
 - a. In **Table name**, enter the table name: `wx_data`.
 - b. In **Primary key**, in **Partition key**, enter `sample_time`, and in the option list next to the field, choose **Number**.
 - c. Check **Add sort key**.
 - d. In the field that appears below **Add sort key**, enter `device_id`, and in the option list next to the field, choose **Number**.
 - e. At the bottom of the page, choose **Create**.

You'll define `device_data` later, when you configure the DynamoDB rule action.

Step 2: Create an AWS IoT rule to send data to the DynamoDB table

In this step, you'll use the rule query statement to format the data from the imaginary weather sensor devices to write to the database table.

A sample message payload received from a weather sensor device looks like this:

```
{  
  "temperature": 28,  
  "humidity": 80,  
  "barometer": 1013,  
  "wind": {  
    "velocity": 22,  
    "bearing": 255  
  }  
}
```

For the database entry, you'll use the rule query statement to flatten the structure of the message payload to look like this:

```
{  
  "temperature": 28,  
  "humidity": 80,
```

```
"barometer": 1013,  
"wind_velocity": 22,  
"wind_bearing": 255  
}
```

In this rule, you'll also use a couple of [Substitution templates \(p. 586\)](#). Substitution templates are expressions that let you insert dynamic values from functions and message data.

To create the AWS IoT rule to send data to the DynamoDB table

1. Open [the Rules hub of the AWS IoT console](#).
2. To start creating your new rule in **Rules**, choose **Create**.
3. In the top part of **Create a rule**:
 - a. In **Name**, enter the rule's name, **wx_data_ddb**.

Remember that a rule name must be unique within your AWS account and Region, and it can't have any spaces. We've used an underscore character in this name to separate the two words in the rule's name.
 - b. In **Description**, describe the rule.

A meaningful description makes it easier to remember what this rule does and why you created it. The description can be as long as needed, so be as detailed as possible.
4. In **Rule query statement** of **Create a rule**:
 - a. In **Using SQL version**, select **2016-03-23**.
 - b. In the **Rule query statement** edit box, enter the statement:

```
SELECT temperature, humidity, barometer,  
      wind.velocity as wind_velocity,  
      wind.bearing as wind_bearing,  
FROM 'device/+/data'
```

This statement:

- Listens for MQTT messages with a topic that matches the device/+/data topic filter.
- Formats the elements of the wind attribute as individual attributes.
- Passes the temperature, humidity, and barometer attributes unchanged.

5. In **Set one or more actions**:
 - a. To open up the list of rule actions for this rule, choose **Add action**.
 - b. In **Select an action**, choose **Insert a message into a DynamoDB table**.
 - c. To open the selected action's configuration page, at the bottom of the action list, choose **Configure action**.
6. In **Configure action**:
 - a. In **Table name**, choose the name of the DynamoDB table you created in a previous step: **wx_data**.

The **Partition key**, **Partition key type**, **Sort key**, and **Sort key type** fields are filled with the values from your DynamoDB table.

- b. In **Partition key value**, enter **\${timestamp()}`**.

This is the first of the [Substitution templates \(p. 586\)](#) you'll use in this rule. Instead of using a value from the message payload, it will use the value returned from the [timestamp \(p. 578\)](#) function.

- c. In Sort key value, enter `#{cast(topic(2) AS DECIMAL)}`.

This is the second one of the [Substitution templates \(p. 586\)](#) you'll use in this rule. It inserts the value of the second element in [topic \(p. 578\)](#) name, which is the device's ID, after it [casts \(p. 547\)](#) it to a DECIMAL value to match the numeric format of the key.

- d. In **Write message data to this column**, enter `device_data`.

This will create the `device_data` column in the DynamoDB table.

- e. Leave **Operation** blank.
- f. In **Choose or create a role to grant AWS IoT access to perform this action**, choose **Create Role**.
- g. In **Create a new role**, enter `wx_ddb_role`, and choose **Create role**.
- h. At the bottom of **Configure action**, choose **Add action**.
- i. To create the rule, at the bottom of **Create a rule**, choose **Create rule**.

Step 3: Test the AWS IoT rule and DynamoDB table

To test the new rule, you'll use the MQTT client to publish and subscribe to the MQTT messages used in this test.

Open the [MQTT client in the AWS IoT console](#) in a new window. This will let you edit the rule without losing the configuration of your MQTT client. The MQTT client does not retain any subscriptions or message logs if you leave it to go to another page in the console. You'll also want a separate console window open to the [DynamoDB Tables hub in the AWS IoT console](#) to view the new entries that your rule sends.

To use the MQTT client to test your rule

1. In the [MQTT client in the AWS IoT console](#), subscribe to the input topic, `device/+data`.
 - a. In the MQTT client, choose **Subscribe to a topic**.
 - b. For **Topic filter**, enter the topic of the input topic filter, `device/+data`.
 - c. Choose **Subscribe**.
2. Now, publish a message to the input topic with a specific device ID, `device/22/data`. You can't publish to MQTT topics that contain wildcard characters.
 - a. In the MQTT client, choose **Publish to a topic**.
 - b. For **Topic name**, enter the input topic name, `device/22/data`.
 - c. For **Message payload**, enter the following sample data.

```
{  
    "temperature": 28,  
    "humidity": 80,  
    "barometer": 1013,  
    "wind": {  
        "velocity": 22,  
        "bearing": 255  
    }  
}
```

- d. To publish the MQTT message, choose **Publish**.
- e. Now, in the MQTT client, choose **Subscribe to a topic**. In the **Subscribe** column, choose the `device/+data` subscription. Confirm that the sample data from the previous step appears there.
3. Check to see the row in the DynamoDB table that your rule created.

- a. In the [DynamoDB Tables hub in the AWS IoT console](#), choose **wx_data**, and then choose the **Items** tab.

If you're already on the **Items** tab, you might need to refresh the display by choosing the refresh icon in the upper-right corner of the table's header.

- b. Notice that the **sample_time** values in the table are links and open one. If you just sent your first message, it will be the only one in the list.

This link displays all the data in that row of the table.

- c. Expand the **device_data** entry to see the data that resulted from the rule query statement.
- d. Explore the different representations of the data that are available in this display. You can also edit the data in this display.
- e. After you have finished reviewing this row of data, to save any changes you made, choose **Save**, or to exit without saving any changes, choose **Cancel**.

If you don't see the correct behavior, check the troubleshooting tips.

Troubleshooting your DynamoDB rule

Here are some things to check in case you're not seeing the results you expect.

- **You got an error banner**

If an error appeared when you published the input message, correct that error first. The following steps might help you correct that error.

- **You don't see the input message in the MQTT client**

Every time you publish your input message to the `device/22/data` topic, that message should appear in the MQTT client if you subscribed to the `device/+data` topic filter as described in the procedure.

Things to check

- **Check the topic filter you subscribed to**

If you subscribed to the input message topic as described in the procedure, you should see a copy of the input message every time you publish it.

If you don't see the message, check the topic name you subscribed to and compare it to the topic to which you published. Topic names are case sensitive and the topic to which you subscribed must be identical to the topic to which you published the message payload.

- **Check the message publish function**

In the MQTT client, under **Subscriptions**, choose `device/+data`, check the topic of the publish message, and then choose **Publish to topic**. You should see the message payload from the edit box below the topic appear in the message list.

- **You don't see your data in the DynamoDB table**

The first thing to do is to refresh the display by choosing the refresh icon in the upper-right corner of the table's header. If that doesn't display the data you're looking for, check the following.

Things to check

- **Check the AWS Region of your MQTT client and the rule that you created**

The console in which you're running the MQTT client must be in the same AWS Region as the rule you created.

- **Check the input message topic in the rule query statement**

For the rule to work, it must receive a message with the topic name that matches the topic filter in the FROM clause of the rule query statement.

Check the spelling of the topic filter in the rule query statement with that of the topic in the MQTT client. Topic names are case sensitive and the message's topic must match the topic filter in the rule query statement.

- **Check the contents of the input message payload**

For the rule to work, it must find the data field in the message payload that is declared in the SELECT statement.

Check the spelling of the `temperature` field in the rule query statement with that of the message payload in the MQTT client. Field names are case sensitive and the `temperature` field in the rule query statement must be identical to the `temperature` field in the message payload.

Make sure that the JSON document in the message payload is correctly formatted. If the JSON has any errors, such as a missing comma, the rule will not be able to read it.

- **Check the key and field names used in the rule action**

The field names used in the topic rule must match those found in the JSON message payload of the published message.

Open the rule you created in the console and check the field names in the rule action configuration with those used in the MQTT client.

- **Check the role being used by the rule**

The rule action must have permission to receive the original topic and publish the new topic.

The policies that authorize the rule to receive message data and update the DynamoDB table are specific to the topics used. If you change the topic or DynamoDB table name used by the rule, you must update the rule action's role to update its policy to match.

If you suspect this is the problem, edit the rule action and create a new role. New roles created by the rule action receive the authorizations necessary to perform these actions.

Step 4: Review the results and next steps

After you send a few messages to the DynamoDB table with this rule, try experimenting with it to see how changing some aspects from the tutorial affect the data written to the table. Here are some ideas to get you started.

- Change the `device_id` in the input message's topic and observe the effect on the data. You could use this to simulate receiving data from multiple weather sensors.
- Change the fields selected in the rule query statement and observe the effect on the data. You could use this to filter the data stored in the table.
- Add a republish rule action to send an MQTT message for each row added to the table. You could use this for debugging.

After you have completed this tutorial, check out [the section called “Formatting a notification by using an AWS Lambda function” \(p. 203\)](#).

Tutorial: Formatting a notification by using an AWS Lambda function

This tutorial demonstrates how to send MQTT message data to an AWS Lambda action for formatting and sending to another AWS service. In this tutorial, the AWS Lambda action uses the AWS SDK to send the formatted message to the Amazon SNS topic you created in the tutorial about how to [the section called "Sending an Amazon SNS notification" \(p. 190\)](#).

In the tutorial about how to [the section called "Sending an Amazon SNS notification" \(p. 190\)](#), the JSON document that resulted from the rule's query statement was sent as the body of the text message. The result was a text message that looked something like this example:

```
{"device_id": "32", "reported_temperature": 38, "max_temperature": 30}
```

In this tutorial, you'll use an AWS Lambda rule action to call an AWS Lambda function that formats the data from the rule query statement into a friendlier format, such as this example:

```
Device 32 reports a temperature of 38, which exceeds the limit of 30.
```

The AWS Lambda function you'll create in this tutorial formats the message string by using the data from the rule query statement and calls the [SNS publish](#) function of the AWS SDK to create the notification.

What you'll learn in this tutorial

- How to create and test an AWS Lambda function
- How to use the AWS SDK in an AWS Lambda function to publish an Amazon SNS notification
- How to use simple SQL queries and functions in a rule query statement
- How to use the MQTT client to test an AWS IoT rule

This tutorial takes about 45 minutes to complete.

In this tutorial, you'll:

- [Step 1: Create an AWS Lambda function that sends a text message \(p. 204\)](#)
- [Step 2: Create an AWS IoT rule with an AWS Lambda rule action \(p. 206\)](#)
- [Step 3: Test the AWS IoT rule and AWS Lambda rule action \(p. 207\)](#)
- [Step 4: Review the results and next steps \(p. 210\)](#)

Before you start this tutorial, make sure that you have:

- [Set up your AWS account \(p. 17\)](#)

You'll need your AWS account and AWS IoT console to complete this tutorial.

- [Reviewed **View MQTT messages with the AWS IoT MQTT client \(p. 62\)**](#)

Be sure you can use the MQTT client to subscribe and publish to a topic. You'll use the MQTT client to test your new rule in this procedure.

- [Completed the other rules tutorials in this section](#)

This tutorial requires the SNS notification topic you created in the tutorial about how to [the section called "Sending an Amazon SNS notification" \(p. 190\)](#). It also assumes that you've completed the other rules-related tutorials in this section.

- [Reviewed the **AWS Lambda overview**](#)

If you haven't used AWS Lambda before, review [AWS Lambda](#) and [Getting started with Lambda](#) to learn its terms and concepts.

Step 1: Create an AWS Lambda function that sends a text message

The AWS Lambda function in this tutorial receives the result of the rule query statement, inserts the elements into a text string, and sends the resulting string to Amazon SNS as the message in a notification.

Unlike the tutorial about how to [the section called "Sending an Amazon SNS notification" \(p. 190\)](#), which used an AWS IoT rule action to send the notification, this tutorial sends the notification from the Lambda function by using a function of the AWS SDK. The actual Amazon SNS notification topic used in this tutorial, however, is the same one that you used in the tutorial about how to [the section called "Sending an Amazon SNS notification" \(p. 190\)](#).

To create an AWS Lambda function that sends a text message

1. Create a new AWS Lambda function.
 - a. In the [AWS Lambda console](#), choose **Create function**.
 - b. In **Create function**, select **Use a blueprint**.
Search for and select the **hello-world-python** blueprint, and then choose **Configure**.
 - c. In **Basic information**:
 - i. In **Function name**, enter the name of this function, **format-high-temp-notification**.
 - ii. In **Execution role**, choose **Create a new role from AWS policy templates**.
 - iii. In **Role name**, enter the name of the new role, **format-high-temp-notification-role**.
 - iv. In **Policy templates - optional**, search for and select **Amazon SNS publish policy**.
 - v. Choose **Create function**.
2. Modify the blueprint code to format and send an Amazon SNS notification.
 - a. After you created your function, you should see the **format-high-temp-notification** details page. If you don't, open it from the [Lambda Functions](#) page.
 - b. In the **format-high-temp-notification** details page, choose the **Configuration** tab and scroll to the **Function code** panel.
 - c. In the **Function code** window, in the **Environment** pane, choose the Python file, **lambda_function.py**.
 - d. In the **Function code** window, delete all of the original program code from the blueprint and replace it with this code.

```
import boto3
#
# expects event parameter to contain:
#
#     "device_id": "32",
#     "reported_temperature": 38,
#     "max_temperature": 30,
#     "notify_topic_arn": "arn:aws:sns:us-east-1:57EXAMPLE833:high_temp_notice"
#
# sends a plain text string to be used in a text message
#
#     "Device {0} reports a temperature of {1}, which exceeds the limit of {2}."
```

```
# where:  
#     {0} is the device_id value  
#     {1} is the reported_temperature value  
#     {2} is the max_temperature value  
#  
def lambda_handler(event, context):  
  
    # Create an SNS client to send notification  
    sns = boto3.client('sns')  
  
    # Format text message from data  
    message_text = "Device {0} reports a temperature of {1}, which exceeds the  
    limit of {2}.".format(  
        str(event['device_id']),  
        str(event['reported_temperature']),  
        str(event['max_temperature']))  
    )  
  
    # Publish the formatted message  
    response = sns.publish(  
        TopicArn = event['notify_topic_arn'],  
        Message = message_text  
    )  
  
    return response
```

e. Choose **Deploy**.

3. In a new window, look up the Amazon Resource Name (ARN) of your Amazon SNS topic from the tutorial about how to [the section called “Sending an Amazon SNS notification” \(p. 190\)](#).
 - a. In a new window, open the [Topics page of the Amazon SNS console](#).
 - b. In the **Topics** page, find the **high_temp_notice** notification topic in the list of Amazon SNS topics.
 - c. Find the **ARN** of the **high_temp_notice** notification topic to use in the next step.
4. Create a test case for your Lambda function.
 - a. In the [Lambda Functions](#) page of the console, on the **format-high-temp-notification** details page, choose **Select a test event** in the upper right corner of the page (even though it looks disabled), and then choose **Configure test events**.
 - b. In **Configure test event**, choose **Create new test event**.
 - c. In **Event name**, enter **SampleRuleOutput**.
 - d. In the JSON editor below **Event name**, paste this sample JSON document. This is an example of what your AWS IoT rule will send to the Lambda function.

```
{  
    "device_id": "32",  
    "reported_temperature": 38,  
    "max_temperature": 30,  
    "notify_topic_arn": "arn:aws:sns:us-east-1:57EXAMPLE833:high_temp_notice"  
}
```

- e. Refer to the window that has the **ARN** of the **high_temp_notice** notification topic and copy the ARN value.
- f. Replace the **notify_topic_arn** value in the JSON editor with the ARN from your notification topic.

Keep this window open so you can use this ARN value again when you create the AWS IoT rule.

g. Choose **Create**.

5. Test the function with sample data.

- a. In the **format-high-temp-notification** details page, in the upper-right corner of the page, confirm that **SampleRuleOutput** appears next to the **Test** button. If it doesn't, choose it from the list of available test events.
- b. To send the sample rule output message to your function, choose **Test**.

If the function and the notification both worked, you will get a text message on the phone that subscribed to the notification.

If you didn't get a text message on the phone, check the result of the operation. In the **Function code** panel, in the **Execution result** tab, review the response to find any errors that occurred. Don't continue to the next step until your function can send the notification to your phone.

Step 2: Create an AWS IoT rule with an AWS Lambda rule action

In this step, you'll use the rule query statement to format the data from the imaginary weather sensor device to send to a Lambda function, which will format and send a text message.

A sample message payload received from the weather devices looks like this:

```
{  
  "temperature": 28,  
  "humidity": 80,  
  "barometer": 1013,  
  "wind": {  
    "velocity": 22,  
    "bearing": 255  
  }  
}
```

In this rule, you'll use the rule query statement to create a message payload for the Lambda function that looks like this:

```
{  
  "device_id": "32",  
  "reported_temperature": 38,  
  "max_temperature": 30,  
  "notify_topic_arn": "arn:aws:sns:us-east-1:57EXAMPLE833:high_temp_notice"  
}
```

This contains all the information the Lambda function needs to format and send the correct text message.

To create the AWS IoT rule to call a Lambda function

1. Open the [Rules hub of the AWS IoT console](#).
2. To start creating your new rule in **Rules**, choose **Create**.
3. In the top part of **Create a rule**:
 - a. In **Name**, enter the rule's name, **wx_friendly_text**.
Remember that a rule name must be unique within your AWS account and Region, and it can't have any spaces. We've used an underscore character in this name to separate the two words in the rule's name.
 - b. In **Description**, describe the rule.
A meaningful description makes it easier to remember what this rule does and why you created it. The description can be as long as needed, so be as detailed as possible.

4. In **Rule query statement** of **Create a rule**:

- a. In **Using SQL version**, select **2016-03-23**.
- b. In the **Rule query statement** edit box, enter the statement:

```
SELECT
    cast(topic(2) AS DECIMAL) as device_id,
    temperature as reported_temperature,
    30 as max_temperature,
    'arn:aws:sns:us-east-1:57EXAMPLE833:high_temp_notice' as notify_topic_arn
FROM 'device/+/data' WHERE temperature > 30
```

This statement:

- Listens for MQTT messages with a topic that matches the `device/+/data` topic filter and that have a `temperature` value greater than 30.
 - Selects the second element from the topic string, converts it to a decimal number, and then assigns it to the `device_id` field.
 - Selects the value of the `temperature` field from the message payload and assigns it to the `reported_temperature` field.
 - Creates a constant value, 30, to represent the limit value and assigns it to the `max_temperature` field.
 - Creates a constant value for the `notify_topic_arn` field.
- c. Refer to the window that has the **ARN** of the **high_temp_notice** notification topic and copy the ARN value.
 - d. Replace the ARN value (`arn:aws:sns:us-east-1:57EXAMPLE833:high_temp_notice`) in the rule query statement editor with the ARN of your notification topic.
5. In **Set one or more actions**:
- a. To open up the list of rule actions for this rule, choose **Add action**.
 - b. In **Select an action**, choose **Send a message to a Lambda function**.
 - c. To open the selected action's configuration page, at the bottom of the action list, choose **Configure action**.
6. In **Configure action**:
- a. In **Function name**, choose **Select**.
 - b. Choose **format-high-temp-notification**.
 - c. At the bottom of **Configure action**, choose **Add action**.
 - d. To create the rule, at the bottom of **Create a rule**, choose **Create rule**.

Step 3: Test the AWS IoT rule and AWS Lambda rule action

To test your new rule, you'll use the MQTT client to publish and subscribe to the MQTT messages used by this rule.

Open the [MQTT client in the AWS IoT console](#) in a new window. Now you can edit the rule without losing the configuration of your MQTT client. If you leave the MQTT client to go to another page in the console, you'll lose your subscriptions or message logs.

To use the MQTT client to test your rule

1. In the [MQTT client in the AWS IoT console](#), subscribe to the input topics, in this case, `device/+/data`.

- a. In the MQTT client, under **Subscriptions**, choose **Subscribe to a topic**.
- b. In **Subscription topic**, enter the topic of the input topic filter, **device/+data**.
- c. Keep the rest of the fields at their default settings.
- d. Choose **Subscribe to topic**.

In the **Subscriptions** column, under **Publish to a topic**, **device/+data** appears.

2. Publish a message to the input topic with a specific device ID, **device/32/data**. You can't publish to MQTT topics that contain wildcard characters.
 - a. In the MQTT client, under **Subscriptions**, choose **Publish to topic**.
 - b. In the **Publish** field, enter the input topic name, **device/32/data**.
 - c. Copy the sample data shown here and, in the edit box below the topic name, paste the sample data.

```
{  
    "temperature": 38,  
    "humidity": 80,  
    "barometer": 1013,  
    "wind": {  
        "velocity": 22,  
        "bearing": 255  
    }  
}
```

2. Publish a message to the input topic with a specific device ID, **device/32/data**. You can't publish to MQTT topics that contain wildcard characters.
 - a. In the MQTT client, under **Subscriptions**, choose **Publish to topic**.
 - b. In the **Publish** field, enter the input topic name, **device/32/data**.
 - c. Copy the sample data shown here and, in the edit box below the topic name, paste the sample data.
 - d. To publish your MQTT message, choose **Publish to topic**.
3. Confirm that the text message was sent.

- a. In the MQTT client, under **Subscriptions**, there is a green dot next to the topic to which you subscribed earlier.

The green dot indicates that one or more new messages have been received since the last time you looked at them.

- b. Under **Subscriptions**, choose **device/+data** to check that the message payload matches what you just published and looks like this:

```
{  
    "temperature": 38,  
    "humidity": 80,  
    "barometer": 1013,  
    "wind": {  
        "velocity": 22,  
        "bearing": 255  
    }  
}
```

- c. Check the phone that you used to subscribe to the SNS topic and confirm the contents of the message payload look like this:

```
Device 32 reports a temperature of 38, which exceeds the limit of 30.
```

If you change the topic ID element in the message topic, remember that casting the `topic(2)` value to a numeric value will only work if that element in the message topic contains only numeric characters.

4. Try sending an MQTT message in which the temperature does not exceed the limit.
 - a. In the MQTT client, under **Subscriptions**, choose **Publish to topic**.

- b. In the **Publish** field, enter the input topic name, **device/33/data**.
- c. Copy the sample data shown here and, in the edit box below the topic name, paste the sample data.

```
{  
    "temperature": 28,  
    "humidity": 80,  
    "barometer": 1013,  
    "wind": {  
        "velocity": 22,  
        "bearing": 255  
    }  
}
```

- d. To send your MQTT message, choose **Publish to topic**.

You should see the message that you sent in the **device/+/**data**** subscription; however, because the temperature value is below the max temperature in the rule query statement, you shouldn't receive a text message.

If you don't see the correct behavior, check the troubleshooting tips.

Troubleshooting your AWS Lambda rule and notification

Here are some things to check, in case you're not seeing the results you expect.

- **You got an error banner**

If an error appeared when you published the input message, correct that error first. The following steps might help you correct that error.

- **You don't see the input message in the MQTT client**

Every time you publish your input message to the **device/32/**data**** topic, that message should appear in the MQTT client, if you subscribed to the **device/+/**data**** topic filter as described in the procedure.

Things to check

- **Check the topic filter you subscribed to**

If you subscribed to the input message topic as described in the procedure, you should see a copy of the input message every time you publish it.

If you don't see the message, check the topic name you subscribed to and compare it to the topic to which you published. Topic names are case sensitive and the topic to which you subscribed must be identical to the topic to which you published the message payload.

- **Check the message publish function**

In the MQTT client, under **Subscriptions**, choose **device/+/**data****, check the topic of the publish message, and then choose **Publish to topic**. You should see the message payload from the edit box below the topic appear in the message list.

- **You don't receive an SMS message**

For your rule to work, it must have the correct policy that authorizes it to receive a message and send an SNS notification, and it must receive the message.

Things to check

- **Check the AWS Region of your MQTT client and the rule that you created**

The console in which you're running the MQTT client must be in the same AWS Region as the rule you created.

- **Check that the temperature value in the message payload exceeds the test threshold**

If the temperature value is less than or equal to 30, as defined in the rule query statement, the rule will not perform any of its actions.

- **Check the input message topic in the rule query statement**

For the rule to work, it must receive a message with the topic name that matches the topic filter in the FROM clause of the rule query statement.

Check the spelling of the topic filter in the rule query statement with that of the topic in the MQTT client. Topic names are case sensitive and the message's topic must match the topic filter in the rule query statement.

- **Check the contents of the input message payload**

For the rule to work, it must find the data field in the message payload that is declared in the SELECT statement.

Check the spelling of the temperature field in the rule query statement with that of the message payload in the MQTT client. Field names are case sensitive and the temperature field in the rule query statement must be identical to the temperature field in the message payload.

Make sure that the JSON document in the message payload is correctly formatted. If the JSON has any errors, such as a missing comma, the rule will not be able to read it.

- **Check the Amazon SNS notification**

In [Step 1: Create an Amazon SNS topic that sends an SMS text message \(p. 191\)](#), refer to step 3 that describes how to test the Amazon SNS notification and test the notification to make sure the notification works.

- **Check the Lambda function**

In [Step 1: Create an AWS Lambda function that sends a text message \(p. 204\)](#), refer to step 5 that describes how to test the Lambda function using test data and test the Lambda function.

- **Check the role being used by the rule**

The rule action must have permission to receive the original topic and publish the new topic.

The policies that authorize the rule to receive message data and republish it are specific to the topics used. If you change the topic used to republish the message data, you must update the rule action's role to update its policy to match the current topic.

If you suspect this is the problem, edit the Republish rule action and create a new role. New roles created by the rule action receive the authorizations necessary to perform these actions.

Step 4: Review the results and next steps

In this tutorial:

- You created an AWS IoT rule to call a Lambda function that sent an Amazon SNS notification that used your customized message payload.

- You used a simple SQL query and functions in a rule query statement to create a new message payload for your Lambda function.
- You used the MQTT client to test your AWS IoT rule.

Next steps

After you send a few text messages with this rule, try experimenting with it to see how changing some aspects of the tutorial affect the message and when it's sent. Here are some ideas to get you started.

- Change the `device_id` in the input message's topic and observe the effect in the text message contents.
- Change the fields selected in the rule query statement, update the Lambda function to use them in a new message, and observe the effect in the text message contents.
- Change the test in the rule query statement to test for a minimum temperature instead of a maximum temperature. Update the Lambda function to format a new message and remember to change the name of `max_temperature`.
- To learn more about how to find errors that might occur while you're developing and using AWS IoT rules, see [Monitoring AWS IoT \(p. 400\)](#).

Retaining the device state while the device is offline

These tutorials show you how to use the AWS IoT Device Shadow service to store and update the state information of a device. The Shadow document, which is a JSON document, shows the change in the device's state based on the messages published by a device, local app, or service. In this tutorial, the Shadow document shows the change in the color of a light bulb. These tutorials also show how the shadow stores this information even when the device is disconnected from the internet, and passes the latest state information back to the device when it comes back online and requests this information.

We recommend that you try these tutorials in the order they're shown here, starting with the AWS IoT resources you need to create and the necessary hardware setup, which also helps you learn the concepts incrementally. These tutorials show how to configure and connect a Raspberry Pi device for use with AWS IoT. If you don't have the required hardware, you can follow these tutorials by adapting them to a device of your choice or by [creating a virtual device with Amazon EC2 \(p. 39\)](#).

Tutorial scenario overview

The scenario for these tutorials is a local app or service that changes the color of a light bulb and that publishes its data to reserved shadow topics. These tutorials are similar to the Device Shadow functionality described in the [interactive getting started tutorial \(p. 19\)](#) and are implemented on a Raspberry Pi device. The tutorials in this section focus on a single, classic shadow while showing how you might accommodate named shadows or multiple devices.

The following tutorials will help you learn how to use the AWS IoT Device Shadow service.

- [Tutorial: Preparing your Raspberry Pi to run the shadow application \(p. 213\)](#)

This tutorial shows how to set up a Raspberry Pi device for connecting with AWS IoT. You'll also create an AWS IoT policy document and a thing resource, download the certificates, and then attach the policy to that thing resource. This tutorial takes about 30 minutes to complete.

- [Tutorial: Installing the Device SDK and running the sample application for Device Shadows \(p. 217\)](#)

This tutorial shows how to install the required tools, software, and the AWS IoT Device SDK for Python, and then run the sample shadow application. This tutorial builds on concepts presented in [Connect a Raspberry Pi or another device \(p. 53\)](#) and takes 20 minutes to complete.

• **Tutorial: Interacting with Device Shadow using the sample app and the MQTT test client (p. 223)**

This tutorial shows how you use the `shadow.py` sample app and **AWS IoT console** to observe the interaction between AWS IoT Device Shadows and the state changes of the light bulb. The tutorial also shows how to send MQTT messages to the Device Shadow's reserved topics. This tutorial can take 45 minutes to complete.

AWS IoT Device Shadow overview

A Device Shadow is a persistent, virtual representation of a device that is managed by a [thing resource \(p. 251\)](#) you create in the AWS IoT registry. The Shadow document is a JSON or a JavaScript notation doc that is used to store and retrieve the current state information for a device. You can use the shadow to get and set the state of a device over MQTT topics or HTTP REST APIs, regardless of whether the device is connected to the internet.

A Shadow document contains a `state` property that describes these aspects of the device's state.

- `desired`: Apps specify the desired states of device properties by updating the `desired` object.
- `reported`: Devices report their current state in the `reported` object.
- `delta`: AWS IoT reports differences between the desired and the reported state in the `delta` object.

Here is an example of a Shadow state document.

```
{  
  "state": {  
    "desired": {  
      "color": "green"  
    },  
    "reported": {  
      "color": "blue"  
    },  
    "delta": {  
      "color": "green"  
    }  
  }  
}
```

To update a device's Shadow document, you can use the [reserved MQTT topics \(p. 106\)](#), the [Device Shadow REST APIs \(p. 615\)](#) that support the GET, UPDATE, and DELETE operations with HTTP, and the [AWS IoT CLI](#).

In the previous example, say you want to change the desired color to yellow. To do this, send a request to the [UpdateThingShadow \(p. 617\)](#) API or publish a message to the [Update \(p. 622\)](#) topic, `$aws/things/THING_NAME/shadow/update`.

```
{  
  "state": {  
    "desired": {  
      "color": yellow  
    }  
  }  
}
```

Updates affect only the fields specified in the request. After successfully updating the Device Shadow, AWS IoT publishes the new desired state to the delta topic, `$aws/things/THING_NAME/shadow/delta`. The Shadow document in this case looks like this:

```
{
```

```
"state": {  
    "desired": {  
        "color": yellow  
    },  
    "reported": {  
        "color": green  
    },  
    "delta": {  
        "color": yellow  
    }  
}
```

The new state is then reported to the AWS IoT Device Shadow using the Update topic `$aws/things/THING_NAME/shadow/update` with the following JSON message:

```
{  
    "state": {  
        "reported": {  
            "color": yellow  
        }  
    }  
}
```

If you want to get the current state information, send a request to the [GetThingShadow \(p. 616\)](#) API or publish an MQTT message to the [Get \(p. 620\)](#) topic, `$aws/things/THING_NAME/shadow/get`.

For more information about using the Device Shadow service, see [AWS IoT Device Shadow service \(p. 591\)](#).

For more information about using Device Shadows in devices, apps, and services, see [Using shadows in devices \(p. 594\)](#) and [Using shadows in apps and services \(p. 597\)](#).

For information about interacting with AWS IoT shadows, see [Interacting with shadows \(p. 609\)](#).

For information about the MQTT reserved topics and HTTP REST APIs, see [Device Shadow MQTT topics \(p. 619\)](#) and [Device Shadow REST API \(p. 615\)](#).

Tutorial: Preparing your Raspberry Pi to run the shadow application

This tutorial demonstrates how to set up and configure a Raspberry Pi device and create the AWS IoT resources that a device requires to connect and exchange MQTT messages.

Note

If you're planning to [the section called "Create a virtual device with Amazon EC2" \(p. 39\)](#), you can skip this page and continue to [the section called "Configure your device" \(p. 38\)](#). You'll create these resources when you create your virtual thing. If you would like to use a different device instead of the Raspberry Pi, you can try to follow these tutorials by adapting them to a device of your choice.

In this tutorial, you'll learn how to:

- Set up a Raspberry Pi device and configure it for use with AWS IoT.
- Create an AWS IoT policy document, which authorizes your device to interact with AWS IoT services.
- Create a thing resource in AWS IoT the X.509 device certificates, and then attach the policy document.

The thing is the virtual representation of your device in the AWS IoT registry. The certificate authenticates your device to AWS IoT Core, and the policy document authorizes your device to interact with AWS IoT.

How to run this tutorial

To run the `shadow.py` sample application for Device Shadows, you'll need a Raspberry Pi device that connects to AWS IoT. We recommend that you follow this tutorial in the order it's presented here, starting with setting up the Raspberry Pi and its accessories, and then creating a policy and attaching the policy to a thing resource that you create. You can then follow this tutorial by using the graphical user interface (GUI) supported by the Raspberry Pi to open the AWS IoT console on the device's web browser, which also makes it easier to download the certificates directly to your Raspberry Pi for connecting to AWS IoT.

Before you start this tutorial, make sure that you have:

- An AWS account. If you don't have one, complete the steps described in [Set up your AWS account \(p. 17\)](#) before you continue. You'll need your AWS account and AWS IoT console to complete this tutorial.
- The Raspberry Pi and its necessary accessories. You'll need:
 - A [Raspberry Pi 3 Model B](#) or more recent model. This tutorial might work on earlier versions of the Raspberry Pi, but we haven't tested it.
 - [Raspberry Pi OS \(32-bit\)](#) or later. We recommend using the latest version of the Raspberry Pi OS. Earlier versions of the OS might work, but we haven't tested it.
 - An Ethernet or Wi-Fi connection.
 - Keyboard, mouse, monitor, cables, and power supplies.

This tutorial takes about 30 minutes to complete.

Step 1: Set up and configure Raspberry Pi device

In this section, we'll configure a Raspberry Pi device for use with AWS IoT.

Important

Adapting these instructions to other devices and operating systems can be challenging. You'll need to understand your device well enough to be able to interpret these instructions and apply them to your device. If you encounter difficulties, you might try one of the other device options as an alternative, such as [Create a virtual device with Amazon EC2 \(p. 39\)](#) or [Use your Windows or Linux PC or Mac as an AWS IoT device \(p. 47\)](#).

You'll need to configure your Raspberry Pi such that it can start the operating system (OS), connect to the internet, and allow you to interact with it at a command line interface. You can also use the graphical user interface (GUI) supported with the Raspberry Pi to open the AWS IoT console and run the rest of this tutorial.

To set up the Raspberry Pi

1. Insert the SD card into the MicroSD card slot on the Raspberry Pi. Some SD cards come pre-loaded with an installation manager that prompts you with a menu to install the OS after booting up the board. You can also use the Raspberry Pi imager to install the OS on your card.
2. Connect an HDMI TV or monitor to the HDMI cable that connects to the HDMI port of the Raspberry Pi.
3. Connect the keyboard and mouse to the USB ports of the Raspberry Pi and then plug in the power adapter to boot up the board.

After the Raspberry Pi boots up, if the SD card came pre-loaded with the installation manager, a menu appears to install the operating system. If you have trouble installing the OS, you can try the following steps. For more information about setting up the Raspberry Pi, see [Setting up your Raspberry Pi](#).

If you're having trouble setting up the Raspberry Pi:

- Check whether you inserted the SD card before booting up the board. If you plug in the SD card after booting up the board, the installation menu might not appear.
- Make sure that the TV or monitor is turned on and the correct input is selected.
- Ensure that you are using Raspberry Pi compatible software.

After you have installed and configured the Raspberry Pi OS, open the Raspberry Pi's web browser and navigate to the AWS IoT Core console to continue the rest of the steps in this tutorial.

If you can open the AWS IoT Core console, your Raspberry Pi is ready and you can continue to [the section called "Provisioning your device in AWS IoT" \(p. 215\)](#).

If you're having trouble or need additional help, see [Getting help for your Raspberry Pi](#).

Tutorial: Provisioning your device in AWS IoT

This section creates the AWS IoT Core resources that your tutorial will use.

Steps to provision your device in AWS IoT

- [Step 1: Create an AWS IoT policy for the Device Shadow \(p. 215\)](#)
- [Step 2: Create a thing resource and attach the policy to the thing \(p. 216\)](#)
- [Step 3: Review the results and next steps \(p. 217\)](#)

Step 1: Create an AWS IoT policy for the Device Shadow

X.509 certificates authenticate your device with AWS IoT Core. AWS IoT policies are attached to the certificate that permits the device to perform AWS IoT operations, such as subscribing or publishing to MQTT reserved topics used by the Device Shadow service. Your device presents its certificate when it connects and sends messages to AWS IoT Core.

In this procedure, you'll create a policy that allows your device to perform the AWS IoT operations necessary to run the example program. We recommend that you create a policy that grants only the permissions required to perform the task. You create the AWS IoT policy first, and then attach it to the device certificate that you'll create later.

To create an AWS IoT policy

1. On the left menu, choose **Secure**, and then choose **Policies**. If your account has existing policies, choose **Create**, otherwise, on the **You don't have a policy yet** page, choose **Create a policy**.
2. On the **Create a policy** page:
 - a. Enter a name for the policy in the **Name** field (for example, `My_Device_Shadow_policy`). Do not use personally identifiable information in your policy names.
 - b. In the policy document, you describe connect, subscribe, receive, and publish actions that give the device permission to publish and subscribe to the MQTT reserved topics.

Copy the following sample policy and paste it in your policy document. Replace `thingname` with the name of the thing that you'll create (for example, `My_light_bulb`), `region` with the AWS IoT Region where you're using the services, and `account` with your AWS account number. For more information about AWS IoT policies, see [AWS IoT Core policies \(p. 314\)](#).

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {
```

```

    "Effect": "Allow",
    "Action": [
        "iot:Publish"
    ],
    "Resource": [
        "arn:aws:iot:region:account:topic/$aws/things/thingname/shadow/get",
        "arn:aws:iot:region:account:topic/$aws/things/thingname/shadow/update"
    ]
},
{
    "Effect": "Allow",
    "Action": [
        "iot:Receive"
    ],
    "Resource": [
        "arn:aws:iot:region:account:topic/$aws/things/thingname/shadow/get/accepted",
        "arn:aws:iot:region:account:topic/$aws/things/thingname/shadow/get/rejected",
        "arn:aws:iot:region:account:topic/$aws/things/thingname/shadow/update/accepted",
        "arn:aws:iot:region:account:topic/$aws/things/thingname/shadow/update/rejected",
        "arn:aws:iot:region:account:topic/$aws/things/thingname/shadow/update/delta"
    ],
    {
        "Effect": "Allow",
        "Action": [
            "iot:Subscribe"
        ],
        "Resource": [
            "arn:aws:iot:region:account:topicfilter/$aws/things/thingname/shadow/get/accepted",
            "arn:aws:iot:region:account:topicfilter/$aws/things/thingname/shadow/get/rejected",
            "arn:aws:iot:region:account:topicfilter/$aws/things/thingname/shadow/update/accepted",
            "arn:aws:iot:region:account:topicfilter/$aws/things/thingname/shadow/update/rejected",
            "arn:aws:iot:region:account:topicfilter/$aws/things/thingname/shadow/update/delta"
        ]
    },
    {
        "Effect": "Allow",
        "Action": "iot:Connect",
        "Resource": "arn:aws:iot:region:account:client/test-*"
    }
]
}

```

Step 2: Create a thing resource and attach the policy to the thing

Devices connected to AWS IoT can be represented by *thing resources* in the AWS IoT registry. A *thing resource* represents a specific device or logical entity, such as the light bulb in this tutorial.

To learn how to create a thing in AWS IoT, follow the steps described in [Create a thing object \(p. 37\)](#). Here are some key things to note as you follow the steps in that tutorial:

1. Choose **Create a single thing**, and in the **Name** field, enter a name for the thing that is the same as the thingname (for example, `My_light_bulb`) you specified when you created the policy earlier.

You can't change a thing name after it has been created. If you gave it a different name other than `thingname`, create a new thing with name as `thingname` and delete the old thing.

Note

Do not use personally identifiable information in your thing name. The thing name can appear in unencrypted communications and reports.

2. We recommend that you download each of the certificate files on the **Certificate created!** page into a location where you can easily find them. You'll need to install these files for running the sample application.

We recommend that you download the files into a `certs` subdirectory in your `home` directory on the Raspberry Pi and name each of them with a simpler name as suggested in the following table.

Certificate file names

File	File path
Root CA certificate	<code>~/certs/Amazon-root-CA-1.pem</code>
Device certificate	<code>~/certs/device.pem.crt</code>
Private key	<code>~/certs/private.pem.key</code>

3. After you activate the certificate to enable connections to AWS IoT, choose **Attach a policy** and make sure you attach the policy that you created earlier (for example, `My_Device_Shadow_policy`) to the thing.

After you've created a thing, you can see your thing resource displayed in the list of things in the AWS IoT console.

Step 3: Review the results and next steps

In this tutorial, you learned how to:

- Set up and configure the Raspberry Pi device.
- Create an AWS IoT policy document that authorizes your device to interact with AWS IoT services.
- Create a thing resource and associated X.509 device certificate, and attach the policy document to it.

Next steps

You can now install the AWS IoT device SDK for Python, run the `shadow.py` sample application, and use Device Shadows to control the state. For more information about how to run this tutorial, see [Tutorial: Installing the Device SDK and running the sample application for Device Shadows \(p. 217\)](#).

Tutorial: Installing the Device SDK and running the sample application for Device Shadows

This section shows how you can install the required software and the AWS IoT Device SDK for Python and run the `shadow.py` sample application to edit the Shadow document and control the shadow's state.

In this tutorial, you'll learn how to:

- Use the installed software and AWS IoT Device SDK for Python to run the sample app.
- Learn how entering a value using the sample app publishes the desired value in the AWS IoT console.
- Review the `shadow.py` sample app and how it uses the MQTT protocol to update the shadow's state.

Before you run this tutorial:

You must have set up your AWS account, configured your Raspberry Pi device, and created an AWS IoT thing and policy that gives the device permissions to publish and subscribe to the MQTT reserved topics of the Device Shadow service. For more information, see [Tutorial: Preparing your Raspberry Pi to run the shadow application \(p. 213\)](#).

You must have also installed Git, Python, and the AWS IoT Device SDK for Python. This tutorial builds on the concepts presented in the tutorial [Connect a Raspberry Pi or another device \(p. 53\)](#). If you haven't tried that tutorial, we recommend that you follow the steps described in that tutorial to install the certificate files and Device SDK and then come back to this tutorial to run the `shadow.py` sample app.

In this tutorial, you'll:

- [Step 1: Run the shadow.py sample app \(p. 218\)](#)
- [Step 2: Review the shadow.py Device SDK sample app \(p. 220\)](#)
- [Step 3: Troubleshoot problems with the shadow.py sample app \(p. 222\)](#)
- [Step 4: Review the results and next steps \(p. 223\)](#)

This tutorial takes about 20 minutes to complete.

Step 1: Run the shadow.py sample app

Before you run the `shadow.py` sample app, you'll need the following information in addition to the names and location of the certificate files that you installed.

Application parameter values

Parameter	Where to find the value
<code>your-iot-thing-name</code>	Name of the AWS IoT thing that you created earlier in the section called "Step 2: Create a thing resource and attach the policy to the thing" (p. 216) . To find this value, in the AWS IoT console , choose Manage , and then choose Things .
<code>your-iot-endpoint</code>	The <code>your-iot-endpoint</code> value has a format of: <code>endpoint_id-ats.iot.region.amazonaws.com</code> , for example, <code>a3qj468EXAMPLE-ats.iot.us-west-2.amazonaws.com</code> . To find this value: <ol style="list-style-type: none">1. In the AWS IoT console, choose Manage, and then choose Things.2. Choose the IoT thing you created for your device, My_light_bulb, that you used earlier, and then choose Interact. On the thing details page, your endpoint is displayed in the HTTPS section.

Install and run the sample app

1. Navigate to the sample app directory.

```
cd ~/aws-iot-device-sdk-python-v2/samples
```

2. In the command line window, replace `your-iot-endpoint` and `your-iot-thing-name` as indicated and run this command.

```
python3 shadow.py --root-ca ~/certs/Amazon-root-CA-1.pem --cert ~/certs/device.pem.crt  
--key ~/certs/private.pem.key --endpoint your-iot-endpoint --thing-name your-iot-  
thing-name
```

3. Observe that the sample app:

1. Connects to the AWS IoT service for your account.
2. Subscribes to Delta events and Update and Get responses.
3. Prompts you to enter a desired value in the terminal.
4. Displays output similar to the following:

```
Connecting to a3qEXAMPLEffp-ats.iot.us-west-2.amazonaws.com with client ID  
'test-0c8ae2ff-cc87-49d2-a82a-ae7ba1d0ca5a'...  
Connected!  
Subscribing to Delta events...  
Subscribing to Update responses...  
Subscribing to Get responses...  
Requesting current shadow state...  
Launching thread to read user input...  
Finished getting initial shadow state.  
Shadow contains reported value 'off'.  
Enter desired value:
```

Note

If you're having trouble running the `shadow.py` sample app, review the section called "Step 3: Troubleshoot problems with the `shadow.py` sample app" (p. 222). To get additional information that might help you correct the problem, add the `--verbosity` debug parameter to the command line so the sample app displays detailed messages about what it's doing.

Enter values and observe the updates in Shadow document

You can enter values in the terminal to specify the desired value, which also updates the reported value. Say you enter the color yellow in the terminal. The reported value is also updated to the color yellow. The following shows the messages displayed in the terminal:

```
Enter desired value:  
yellow  
Changed local shadow value to 'yellow'.  
Updating reported shadow value to 'yellow'...  
Update request published.  
Finished updating reported shadow value to 'yellow'.
```

When you publish this update request, AWS IoT creates a default, classic shadow for the thing resource. You can observe the update request that you published to the reported and desired values in the AWS IoT console by looking at the Shadow document for the thing resource that you created (for example, `My_light_bulb`). To see the update in the Shadow document:

1. In the AWS IoT console, choose **Manage** and then choose **Things**.
2. In the list of things displayed, select the thing that you created, choose **Shadows**, and then choose **Classic Shadow**.

The Shadow document should look similar to the following, showing the reported and desired values set to the color yellow. You see these values in the **Shadow state** section of the document.

```
{  
  "desired": {  
    "welcome": "aws-iot",  
    "color": "yellow"  
  },  
  "reported": {  
    "welcome": "aws-iot",  
    "color": "yellow"  
  }  
}
```

You also see a **Metadata** section that contains the timestamp information and version number of the request.

You can use the state document version to ensure you are updating the most recent version of a device's Shadow document. If you send another update request, the version number increments by 1. When you supply a version with an update request, the service rejects the request with an HTTP 409 conflict response code if the current version of the state document doesn't match the version supplied.

```
{  
  "metadata": {  
    "desired": {  
      "welcome": {  
        "timestamp": 1620156892  
      },  
      "color": {  
        "timestamp": 1620156893  
      }  
    },  
    "reported": {  
      "welcome": {  
        "timestamp": 1620156892  
      },  
      "color": {  
        "timestamp": 1620156893  
      }  
    }  
  },  
  "version": 10  
}
```

To learn more about the Shadow document and observe changes to the state information, proceed to the next tutorial [Tutorial: Interacting with Device Shadow using the sample app and the MQTT test client \(p. 223\)](#) as described in the [Step 4: Review the results and next steps \(p. 223\)](#) section of this tutorial. Optionally, you can also learn about the shadow.py sample code and how it uses the MQTT protocol in the following section.

Step 2: Review the shadow.py Device SDK sample app

This section reviews the shadow.py sample app from the [AWS IoT Device SDK v2 for Python](#) used in this tutorial. Here, we'll review how it connects to AWS IoT Core by using the MQTT and MQTT over WSS protocol. The [AWS common runtime \(AWS-CRT\)](#) library provides the low-level communication protocol support and is included with the AWS IoT Device SDK v2 for Python.

While this tutorial uses MQTT and MQTT over WSS, AWS IoT supports devices that publish HTTPS requests. For an example of a Python program that sends an HTTP message from a device, see the [HTTPS code example \(p. 91\)](#) using Python's `requests` library.

For information about how you can make an informed decision about which protocol to use for your device communications, review the [Choosing a protocol for your device communication \(p. 79\)](#).

MQTT

The `shadow.py` sample calls `mtls_from_path` (shown here) in the `mqtt_connection_builder` to establish a connection with AWS IoT Core by using the MQTT protocol. `mtls_from_path` uses X.509 certificates and TLS v1.2 to authenticate the device. The AWS-CRT library handles the lower-level details of that connection.

```
mqtt_connection = mqtt_connection_builder.mtls_from_path(  
    endpoint=args.endpoint,  
    cert_filepath=args.cert,  
    pri_key_filepath=args.key,  
    ca_filepath=args.root_ca,  
    client_bootstrap=client_bootstrap,  
    on_connection_interrupted=on_connection_interrupted,  
    on_connection_resumed=on_connection_resumed,  
    client_id=args.client_id,  
    clean_session=False,  
    keep_alive_secs=6  
)
```

- `endpoint` is your AWS IoT endpoint that you passed in from the command line and `client_id` is the ID that uniquely identifies this device in the AWS Region.
- `cert_filepath`, `pri_key_filepath`, and `ca_filepath` are the paths to the device's certificate and private key files, and the root CA file.
- `client_bootstrap` is the common runtime object that handles socket communication activities, and is instantiated prior to the call to `mqtt_connection_builder.mtls_from_path`.
- `on_connection_interrupted` and `on_connection_resumed` are callback functions to call when the device's connection is interrupted and resumed.
- `clean_session` is whether to start a new, persistent session, or if one is present, reconnect to an existing one. `keep_alive_secs` is the keep alive value, in seconds, to send in the CONNECT request. A ping will automatically be sent at this interval. The server assumes that the connection is lost if it doesn't receive a ping after 1.5 times this value.

The `shadow.py` sample also calls `websockets_with_default_aws_signing` in the `mqtt_connection_builder` to establish a connection with AWS IoT Core using MQTT protocol over WSS. MQTT over WSS also uses the same parameters as MQTT and takes these additional parameters:

- `region` is the AWS signing Region used by Signature V4 authentication, and `credentials_provider` is the AWS credentials provided to use for authentication. The Region is passed from the command line, and the `credentials_provider` object is instantiated just prior to the call to `mqtt_connection_builder.websockets_with_default_aws_signing`.
- `websocket_proxy_options` is the HTTP proxy options, if using a proxy host. In the `shadow.py` sample app, this value is instantiated just prior to the call to `mqtt_connection_builder.websockets_with_default_aws_signing`.

Subscribe to Shadow topics and events

The `shadow.py` sample attempts to establish a connection and waits to be fully connected. If it's not connected, commands are queued up. Once connected, the sample subscribes to delta events and update and get messages, and publishes messages with a Quality of Service (QoS) level of 1 (`mqtt.QoS.AT_LEAST_ONCE`).

When a device subscribes to a message with QoS level 1, the message broker saves the messages that the device is subscribed to until they can be sent to the device. The message broker resends the messages until it receives a PUBACK response from the device.

For more information about the MQTT protocol, see [Review the MQTT protocol \(p. 167\)](#) and [MQTT \(p. 80\)](#).

For more information about how MQTT, MQTT over WSS, persistent sessions, and QoS levels that are used in this tutorial, see [Review the pubsub.py Device SDK sample app \(p. 168\)](#).

Step 3: Troubleshoot problems with the shadow . py sample app

When you run the shadow . py sample app, you should see some messages displayed in the terminal and a prompt to enter a desired value. If the program throws an error, then to debug the error, you can start by checking whether you ran the correct command for your system.

In some cases, the error message might indicate connection issues and look similar to: Host name was invalid for dns resolution or Connection was closed unexpectedly. In such cases, here are some things you can check:

- **Check the endpoint address in the command**

Review the endpoint argument in the command you entered to run the sample app, (for example, a3qEXAMPLEffp-ats.iot.us-west-2.amazonaws.com) and check this value in the **AWS IoT console**.

To check whether you used the correct value:

1. In the **AWS IoT console**, choose **Manage** and then choose **Things**.
2. Choose the thing you created for your sample app (for example, **My_light_bulb**) and then choose **Interact**.

On the thing details page, your endpoint is displayed in the **HTTPS** section. You should also see a message that says: This thing already appears to be connected.

- **Check certificate activation**

Certificates authenticate your device with AWS IoT Core.

To check whether your certificate is active:

1. In the **AWS IoT console**, choose **Manage** and then choose **Things**.
2. Choose the thing you created for your sample app (for example, **My_light_bulb**) and then choose **Security**.
3. Select the certificate and then, from the certificate's details page, choose **Select the certificate** and then, from the certificate's details page, choose **Actions**.

If in the dropdown list **Activate** isn't available and you can only choose **Deactivate**, your certificate is active. If not, choose **Activate** and rerun the sample program.

If the program still doesn't run, check the certificate file names in the certs folder.

- **Check the policy attached to the thing resource**

While certificates authenticate your device, AWS IoT policies permit the device to perform AWS IoT operations, such as subscribing or publishing to MQTT reserved topics.

To check whether the correct policy is attached:

1. Find the certificate as described previously, and then choose **Policies**.
2. Choose the policy displayed and check whether it describes the connect, subscribe, receive, and publish actions that give the device permission to publish and subscribe to the MQTT reserved topics.

If you see error messages that indicate trouble connecting to AWS IoT, it could be because of the permissions you're using for the policy. If that's the case, we recommend that you start with a policy that provides full access to AWS IoT resources and then rerun the sample program. You can either edit the current policy, or choose the current policy, choose **Detach**, and then create another policy that provides full access and attach it to your thing resource. You can later restrict the policy to only the actions and policies you need to run the program.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:*"  
            ],  
            "Resource": "*"  
        }  
    ]  
}
```

- **Check your Device SDK installation**

If the program still doesn't run, you can reinstall the Device SDK to make sure that your SDK installation is complete and correct.

Step 4: Review the results and next steps

In this tutorial, you learned how to:

- Install the required software, tools, and the AWS IoT Device SDK for Python.
- Understand how the sample app, `shadow.py`, uses the MQTT protocol for retrieving and updating the shadow's current state.
- Run the sample app for Device Shadows and observe the update to the Shadow document in the AWS IoT console. You also learned to troubleshoot any issues and fix errors when running the program.

Next steps

You can now run the `shadow.py` sample application and use Device Shadows to control the state. You can observe the updates to the Shadow document in the AWS IoT Console and observe delta events that the sample app responds to. Using the MQTT test client, you can subscribe to the reserved shadow topics and observe messages received by the topics when running the sample program. For more information about how to run this tutorial, see [Tutorial: Interacting with Device Shadow using the sample app and the MQTT test client \(p. 223\)](#).

Tutorial: Interacting with Device Shadow using the sample app and the MQTT test client

To interact with the `shadow.py` sample app, enter a value in the terminal for the desired value. For example, you can specify colors that resemble the traffic lights and AWS IoT responds to the request and updates the reported values.

In this tutorial, you'll learn how to:

- Use the `shadow.py` sample app to specify desired states and update the shadow's current state.
- Edit the Shadow document to observe delta events and how the `shadow.py` sample app responds to it.

- Use the MQTT test client to subscribe to shadow topics and observe updates when you run the sample program.

Before you run this tutorial, you must have:

Set up your AWS account, configured your Raspberry Pi device, and created an AWS IoT thing and policy. You must have also installed the required software, Device SDK, certificate files, and run the sample program in the terminal. For more information, see the previous tutorials [Tutorial: Preparing your Raspberry Pi to run the shadow application \(p. 213\)](#) and [Step 1: Run the shadow.py sample app \(p. 218\)](#). You must complete these tutorials if you haven't already.

In this tutorial, you'll:

- [Step 1: Update desired and reported values using shadow.py sample app \(p. 224\)](#)
- [Step 2: View messages from the shadow.py sample app in the MQTT test client \(p. 225\)](#)
- [Step 3: Troubleshoot errors with Device Shadow interactions \(p. 229\)](#)
- [Step 4: Review the results and next steps \(p. 230\)](#)

This tutorial takes about 45 minutes to complete.

Step 1: Update desired and reported values using shadow.py sample app

In the previous tutorial [Step 1: Run the shadow.py sample app \(p. 218\)](#), you learned how to observe a message published to the Shadow document in the AWS IoT console when you enter a desired value as described in the section [Tutorial: Installing the Device SDK and running the sample application for Device Shadows \(p. 217\)](#).

In the previous example, we set the desired color to yellow. After you enter each value, the terminal prompts you to enter another desired value. If you again enter the same value (yellow), the app recognizes this and prompts you to enter a new desired value.

```
Enter desired value:  
yellow  
Local value is already 'yellow'.  
Enter desired value:
```

Now, say that you enter the color green. AWS IoT responds to the request and updates the reported value to green. This is how the update happens when the desired state is different from the reported state, causing a delta.

How the shadow.py sample app simulates Device Shadow interactions:

1. Enter a desired value (say yellow) in the terminal to publish the desired state.
2. As the desired state is different from the reported state (say the color green), a delta occurs, and the app that is subscribed to the delta receives this message.
3. The app responds to the message and updates its state to the desired value, yellow.
4. The app then publishes an update message with the new reported value of the device's state, yellow.

Following shows the messages displayed in the terminal that shows how the update request is published.

```
Enter desired value:  
green  
Changed local shadow value to 'green'.  
Updating reported shadow value to 'green'...
```

```
Update request published.  
Finished updating reported shadow value to 'green'.
```

In the AWS IoT console, the Shadow document reflects the updated value to `green` for both the `reported` and `desired` fields, and the version number is incremented by 1. For example, if the previous version number was displayed as 10, the current version number will display as 11.

Note

Deleting a shadow doesn't reset the version number to 0. You'll see that the shadow version is incremented by 1 when you publish an update request or create another shadow with the same name.

Edit the Shadow document to observe delta events

The `shadow.py` sample app is also subscribed to delta events, and responds when there is a change to the desired value. For example, you can change the desired value to the color `red`. To do this, in the AWS IoT console, edit the Shadow document by clicking **Edit** and then set the desired value to `red` in the JSON, while keeping the `reported` value to `green`. Before you save the changes, keep the terminal on the Raspberry Pi open as you'll see messages displayed in the terminal when the change occurs.

```
{
  "desired": {
    "welcome": "aws-iot",
    "color": "red"
  },
  "reported": {
    "welcome": "aws-iot",
    "color": "green"
  }
}
```

After you save the new value, the `shadow.py` sample app responds to this change and displays messages in the terminal indicating the delta. You should then see the following messages appear below the prompt for entering the desired value.

```
Enter desired value:  
Received shadow delta event.  
Delta reports that desired value is 'red'. Changing local value...  
Changed local shadow value to 'red'.  
Updating reported shadow value to 'red'...  
Finished updating reported shadow value to 'red'.  
Enter desired value:  
Update request published.  
Finished updating reported shadow value to 'red'.
```

Step 2: View messages from the `shadow.py` sample app in the MQTT test client

You can use the **MQTT test client** in the **AWS IoT console** to monitor MQTT messages that are passed in your AWS account. By subscribing to reserved MQTT topics used by the Device Shadow service, you can observe the messages received by the topics when running the sample app.

If you haven't already used the MQTT test client, you can review [View MQTT messages with the AWS IoT MQTT client \(p. 62\)](#). This helps you learn how to use the **MQTT test client** in the **AWS IoT console** to view MQTT messages as they pass through the message broker.

1. Open the MQTT test client

Open the [MQTT test client in the AWS IoT console](#) in a new window so that you can observe the messages received by the MQTT topics without losing the configuration of your MQTT test client.

The MQTT test client doesn't retain any subscriptions or message logs if you leave it to go to another page in the console. For this section of the tutorial, you can have the Shadow document of your AWS IoT thing and the MQTT test client open in separate windows to more easily observe the interaction with Device Shadows.

2. Subscribe to the MQTT reserved Shadow topics

You can use the MQTT test client to enter the names of the Device Shadow's MQTT reserved topics and subscribe to them to receive updates when running the shadow.py sample app. To subscribe to the topics:

- a. In the **MQTT test client** in the **AWS IoT console**, choose **Subscribe to a topic**.
- b. In the **Topic filter** section, enter: `$aws/things/thingname/shadow/update/#`. Here, `thingname` is the name of the thing resource that you created earlier (for example, `My_light_bulb`).
- c. Keep the default values for the additional configuration settings, and then choose **Subscribe**.

By using the `#` wildcard in the topic subscription, you can subscribe to multiple MQTT topics at the same time and observe all the messages that are exchanged between the device and its Shadow in a single window. For more information about the wildcard characters and their use, see [MQTT topics \(p. 93\)](#).

3. Run shadow.py sample program and observe messages

In your command line window of the Raspberry Pi, if you've disconnected the program, run the sample app again and watch the messages in the **MQTT test client** in the **AWS IoT console**.

- a. Run the following command to restart the sample program. Replace `your-iot-thing-name` and `your-iot-endpoint` with the names of the AWS IoT thing that you created earlier (for example, `My_light_bulb`), and the endpoint to interact with the device.

```
cd ~/aws-iot-device-sdk-python-v2/samples
python3 shadow.py --root-ca ~/certs/Amazon-root-CA-1.pem --cert ~/certs/
device.pem.crt --key ~/certs/private.pem.key --endpoint your-iot-endpoint --thing-
name your-iot-thing-name
```

The shadow.py sample app then runs and retrieves the current shadow state. If you've deleted the shadow or cleared the current states, the program sets the current value to off and then prompts you to enter a desired value.

```
Connecting to a3qEXAMPLEfffp-ats.iot.us-west-2.amazonaws.com with client ID
'test-0c8ae2ff-cc87-49d2-a82a-ae7ba1d0ca5a'...
Connected!
Subscribing to Delta events...
Subscribing to Update responses...
Subscribing to Get responses...
Requesting current shadow state...
Launching thread to read user input...
Finished getting initial shadow state.
Shadow document lacks 'color' property. Setting defaults...
Changed local shadow value to 'off'.
Updating reported shadow value to 'off'...
Update request published.
Finished updating reported shadow value to 'off'...
Enter desired value:
```

On the other hand, if the program was running and you restarted it, you'll see the latest color value reported in the terminal. In the MQTT test client, you'll see an update to the topics `$aws/things/thingname/shadow/get` and `$aws/things/thingname/shadow/get/accepted`.

Suppose that the latest color reported was green. Following shows the contents of the `$aws/things/thingname/shadow/get/accepted` JSON file.

```
{  
  "state": {  
    "desired": {  
      "welcome": "aws-iot",  
      "color": "green"  
    },  
    "reported": {  
      "welcome": "aws-iot",  
      "color": "green"  
    }  
  },  
  "metadata": {  
    "desired": {  
      "welcome": {  
        "timestamp": 1620156892  
      },  
      "color": {  
        "timestamp": 1620161643  
      }  
    },  
    "reported": {  
      "welcome": {  
        "timestamp": 1620156892  
      },  
      "color": {  
        "timestamp": 1620161643  
      }  
    }  
  },  
  "version": 10,  
  "timestamp": 1620173908  
}
```

- b. Enter a desired value in the terminal, such as yellow. The `shadow.py` sample app responds and displays the following messages in the terminal that show the change in the `reported` value to yellow.

```
Enter desired value:  
yellow  
Changed local shadow value to 'yellow'.  
Updating reported shadow value to 'yellow'...  
Update request published.  
Finished updating reported shadow value to 'yellow'.
```

In the **MQTT test client** in the **AWS IoT console**, under **Subscriptions**, you see that the following topics received a message:

- `$aws/things/thingname/shadow/update`: shows that both desired and updated values change to the color yellow.
- `$aws/things/thingname/shadow/update/accepted`: shows the current values of the desired and reported states and their metadata and version information.
- `$aws/things/thingname/shadow/update/documents`: shows the previous and current values of the desired and reported states and their metadata and version information.

As the document **\$aws/things/*thingname*/shadow/update/documents** also contains information that is contained in the other two topics, we can review it to see the state information. The previous state shows the reported value set to green, its metadata and version information, and the current state that shows the reported value updated to yellow.

```
{  
  "previous": {  
    "state": {  
      "desired": {  
        "welcome": "aws-iot",  
        "color": "green"  
      },  
      "reported": {  
        "welcome": "aws-iot",  
        "color": "green"  
      }  
    },  
    "metadata": {  
      "desired": {  
        "welcome": {  
          "timestamp": 1617297888  
        },  
        "color": {  
          "timestamp": 1617297898  
        }  
      },  
      "reported": {  
        "welcome": {  
          "timestamp": 1617297888  
        },  
        "color": {  
          "timestamp": 1617297898  
        }  
      }  
    },  
    "version": 10  
  },  
  "current": {  
    "state": {  
      "desired": {  
        "welcome": "aws-iot",  
        "color": "yellow"  
      },  
      "reported": {  
        "welcome": "aws-iot",  
        "color": "yellow"  
      }  
    },  
    "metadata": {  
      "desired": {  
        "welcome": {  
          "timestamp": 1617297888  
        },  
        "color": {  
          "timestamp": 1617297904  
        }  
      },  
      "reported": {  
        "welcome": {  
          "timestamp": 1617297888  
        },  
        "color": {  
          "timestamp": 1617297904  
        }  
      }  
    }  
  }  
}
```

```

        "timestamp": 1617297904
    }
},
"version": 11
},
"timestamp": 1617297904
}

```

- c. Now, if you enter another desired value, you see further changes to the reported values and message updates received by these topics. The version number also increments by 1. For example, if you enter the value green, the previous state reports the value yellow and the current state reports the value green.

4. Edit Shadow document to observe delta events

To observe changes to the delta topic, edit the Shadow document in the AWS IoT console. For example, you can change the desired value to the color red. To do this, in the AWS IoT console, choose **Edit** and then set the desired value to red in the JSON, while keeping the reported value set to green. Before you save the change, keep the terminal open as you'll see the delta message reported in the terminal.

```

{
"desired": {
    "welcome": "aws-iot",
    "color": "red"
},
"reported": {
    "welcome": "aws-iot",
    "color": "green"
}
}

```

The shadow.py sample app responds to this change and displays messages in the terminal indicating the delta. In the MQTT test client, the update topics will have received a message showing changes to the desired and reported values.

You also see that the topic **\$aws/things/*thingname*/shadow/update/delta** received a message. To see the message, choose this topic, which is listed under **Subscriptions**.

```

{
"version": 13,
"timestamp": 1617318480,
"state": {
    "color": "red"
},
"metadata": {
    "color": {
        "timestamp": 1617318480
    }
}
}

```

Step 3: Troubleshoot errors with Device Shadow interactions

When you run the Shadow sample app, you might encounter issues with observing interactions with the Device Shadow service.

If the program runs successfully and prompts you to enter a desired value, you should be able to observe the Device Shadow interactions by using the Shadow document and the MQTT test client as

described previously. However, if you're unable to see the interactions, here are some things you can check:

- **Check the thing name and its shadow in the AWS IoT console**

If you don't see the messages in the Shadow document, review the command and make sure it matches the thing name in the **AWS IoT console**. You can also check whether you have a classic shadow by choosing your thing resource and then choosing **Shadows**. This tutorial focuses primarily on interactions with the classic shadow.

You can also confirm that the device you used is connected to the internet. In the **AWS IoT console**, choose the thing you created earlier, and then choose **Interact**. On the thing details page, you should see a message here that says: This thing already appears to be connected.

- **Check the MQTT reserved topics you subscribed to**

If you don't see the messages appear in the MQTT test client, check whether the topics you subscribed to are formatted correctly. MQTT Device Shadow topics have a format `$aws/things/thingname/shadow/` and might have update, get, or delete following it depending on actions you want to perform on the shadow. This tutorial uses the topic `$aws/things/thingname/shadow/#` so make sure you entered it correctly when subscribing to the topic in the **Topic filter** section of the test client.

As you enter the topic name, make sure that the `thingname` is the same as the name of the AWS IoT thing that you created earlier. You can also subscribe to additional MQTT topics to see if an update has been successfully performed. For example, you can subscribe to the topic `$aws/things/thingname/shadow/update/rejected` to receive a message whenever an update request failed so that you can debug connection issues. For more information about the reserved topics, see [the section called "Shadow topics" \(p. 106\)](#) and [Device Shadow MQTT topics \(p. 619\)](#).

Step 4: Review the results and next steps

In this tutorial, you learned how to:

- Use the `shadow.py` sample app to specify desired states and update the shadow's current state.
- Edit the Shadow document to observe delta events and how the `shadow.py` sample app responds to it.
- Use the MQTT test client to subscribe to shadow topics and observe updates when you run the sample program.

Next steps

You can subscribe to additional MQTT reserved topics to observe updates to the shadow application. For example, if you only subscribe to the topic `$aws/things/thingname/shadow/update/accepted`, you'll see only the current state information when an update is successfully performed.

You can also subscribe to additional shadow topics to debug issues or learn more about the Device Shadow interactions and also debug any issues with the Device Shadow interactions. For more information, see [the section called "Shadow topics" \(p. 106\)](#) and [Device Shadow MQTT topics \(p. 619\)](#).

You can also choose to extend your application by using named shadows or by using additional hardware connected with the Raspberry Pi for the LEDs and observe changes to their state using messages sent from the terminal.

For more information about the Device Shadow service and using the service in devices, apps, and services, see [AWS IoT Device Shadow service \(p. 591\)](#), [Using shadows in devices \(p. 594\)](#), and [Using shadows in apps and services \(p. 597\)](#).

Tutorial: Creating a custom authorizer for AWS IoT Core

This tutorial demonstrates the steps to create, validate, and use Custom Authentication by using the AWS CLI. Optionally, using this tutorial, you can use Postman to send data to AWS IoT Core by using the HTTP Publish API.

This tutorial shows you how to create a sample Lambda function that implements the authorization and authentication logic and a custom authorizer using the **create-authorizer** call with token signing enabled. The authorizer is then validated using the **test-invoke-authorizer**, and finally you can send data to AWS IoT Core by using the HTTP Publish API to a test MQTT topic. Sample request will specify the authorizer to invoke by using the **x-amz-customauthorizer-name** header and pass the **token-key-name** and **x-amz-customauthorizer-signature** in request headers.

What you'll learn in this tutorial:

- How to create a Lambda function to be a custom authorizer handler
- How to create a custom authorizer using the AWS CLI with token signing enabled
- How to test your custom authorizer using the **test-invoke-authorizer** command
- How to publish an MQTT topic by using [Postman](#) and validate the request with your custom authorizer

This tutorial takes about 60 minutes to complete.

In this tutorial, you'll:

- [Step 1: Create a Lambda function for your custom authorizer \(p. 232\)](#)
- [Step 2: Create a public and private key pair for your custom authorizer \(p. 234\)](#)
- [Step 3: Create a customer authorizer resource and its authorization \(p. 235\)](#)
- [Step 4: Test the authorizer by calling test-invoke-authorizer \(p. 237\)](#)
- [Step 5: Test publishing MQTT message using Postman \(p. 239\)](#)
- [Step 6: View messages in MQTT test client \(p. 240\)](#)
- [Step 7: Review the results and next steps \(p. 241\)](#)
- [Step 8: Clean up \(p. 241\)](#)

Before you start this tutorial, make sure that you have:

- [Set up your AWS account \(p. 17\)](#)

You'll need your AWS account and AWS IoT console to complete this tutorial.

The account you use for this tutorial works best when it includes at least these AWS managed policies:

- [IAMFullAccess](#)
- [AWSIoTFullAccess](#)
- [AWSLambda_FullAccess](#)

Important

The IAM policies used in this tutorial are more permissive than you should follow in a production implementation. In a production environment, make sure that your account and resource policies grant only the necessary permissions.

When you create IAM policies for production, determine what access users and roles need, and then design the policies that allow them to perform only those tasks.

For more information, see [Security best practices in IAM](#)

- **Installed the AWS CLI**

For information about how to install the AWS CLI, see [Installing the AWS CLI](#). This tutorial requires AWS CLI version aws-cli/2.1.3 Python/3.7.4 Darwin/18.7.0 exe/x86_64 or later.

- **OpenSSL tools**

The examples in this tutorial use [LibreSSL 2.6.5](#). You can also use [OpenSSL v1.1.1i](#) tools for this tutorial.

- **Reviewed the AWS Lambda overview**

If you haven't used AWS Lambda before, review [AWS Lambda](#) and [Getting started with Lambda](#) to learn its terms and concepts.

- **Reviewed how to build requests in Postman**

For more information, see [Building requests](#).

- **Removed custom authorizers from previous tutor**

Your AWS account can have only a limited number of custom authorizers configured at one time. For information about how to remove a custom authorizer, see [the section called "Step 8: Clean up" \(p. 241\)](#).

Step 1: Create a Lambda function for your custom authorizer

Custom authentication in AWS IoT Core uses [authorizer resources](#) that you create to authenticate and authorize clients. The function you'll create in this section authenticates and authorizes clients as they connect to AWS IoT Core and access AWS IoT resources.

The Lambda function does the following:

- If a request comes from **test-invoke-authorizer**, it returns an IAM policy with a Deny action.
- If a request comes from Passport using HTTP and the `actionToken` parameter has a value of `allow`, it returns an IAM policy with an Allow action. Otherwise, it returns an IAM policy with a Deny action.

To create the Lambda function for your custom authorizer

1. In the [Lambda](#) console, open [Functions](#).
2. Choose **Create function**.
3. Confirm **Author from scratch** is selected.
4. Under **Basic information**:
 - a. In **Function name**, enter `custom-auth-function`.
 - b. In **Runtime**, confirm **Node.js 14.x**
5. Choose **Create function**.

Lambda creates a Node.js function and an [execution role](#) that grants the function permission to upload logs. The Lambda function assumes the execution role when you invoke your function and uses the execution role to create credentials for the AWS SDK and to read data from event sources.

6. To see the function's code and configuration in the [AWS Cloud9](#) editor, choose **custom-auth-function** in the designer window, and then choose **index.js** in the navigation pane of the editor.

For scripting languages such as Node.js, Lambda includes a basic function that returns a success response. You can use the [AWS Cloud9](#) editor to edit your function as long as your source code doesn't exceed 3 MB.

7. Replace the **index.js** code in the editor with the following code:

```

// A simple Lambda function for an authorizer. It demonstrates
// How to parse a CLI and Http password to generate a response.

exports.handler = function(event, context, callback) {

    //Http parameter to initiate allow/deny request
    const HTTP_PARAM_NAME='actionToken';
    const ALLOW_ACTION = 'Allow';
    const DENY_ACTION = 'Deny';

    //Event data passed to Lambda function
    var event_str = JSON.stringify(event);
    console.log('Complete event :'+ event_str);

    //Read protocolData from the event json passed to Lambda function
    var protocolData = event.protocolData;
    console.log('protocolData value--> ' + protocolData);

    //Get the dynamic account ID from function's ARN to be used
    // as full resource for IAM policy
    var ACCOUNT_ID = context.invokedFunctionArn.split(":")[4];
    console.log("ACCOUNT_ID---"+ACCOUNT_ID);

    //Get the dynamic region from function's ARN to be used
    // as full resource for IAM policy
    var REGION = context.invokedFunctionArn.split(":")[3];
    console.log("REGION---"+REGION);

    //protocolData data will be undefined if testing is done via CLI.
    // This will help to test the set up.
    if (protocolData === undefined) {

        //If CLI testing, pass deny action as this is for testing purpose only.
        console.log('Using the test-invoke-authorizer cli for testing only');
        callback(null, generateAuthResponse(DENY_ACTION,ACCOUNT_ID,REGION));

    } else{

        //Http Testing from Postman
        //Get the query string from the request
        var queryString = event.protocolData.http.queryString;
        console.log('queryString values -- ' + queryString);
        /*          global URLSearchParams          */
        const params = new URLSearchParams(queryString);
        var action = params.get(HTTP_PARAM_NAME);

        if(action!=null && action.toLowerCase() === 'allow'){

            callback(null, generateAuthResponse(ALLOW_ACTION,ACCOUNT_ID,REGION));

        }else{

            callback(null, generateAuthResponse(DENY_ACTION,ACCOUNT_ID,REGION));

        }

    }

};

// Helper function to generate the authorization IAM response.
var generateAuthResponse = function(effect,ACCOUNT_ID,REGION) {

    var full_resource = "arn:aws:iot:"+ REGION + ":" + ACCOUNT_ID + ":"*";

```

```
        console.log("full_resource---"+full_resource);

        var authResponse = {};
        authResponse.isAuthenticated = true;
        authResponse.principalId = 'principalId';

        var policyDocument = {};
        policyDocument.Version = '2012-10-17';
        policyDocument.Statement = [];
        var statement = {};
        statement.Action = 'iot:;';
        statement.Effect = effect;
        statement.Resource = full_resource;
        policyDocument.Statement[0] = statement;
        authResponse.policyDocuments = [policyDocument];
        authResponse.disconnectAfterInSeconds = 3600;
        authResponse.refreshAfterInSeconds = 600;

        console.log('custom auth policy function called from http');
        console.log('authResponse --> ' + JSON.stringify(authResponse));
        console.log(authResponse.policyDocuments[0]);

        return authResponse;
    }
}
```

8. Choose **Deploy**.
9. After **Changes deployed** appears above the editor:
 - a. Scroll to the **Function overview** section above the editor.
 - b. Copy the **Function ARN** and save it to use later in this tutorial.
10. Test your function.
 - a. Choose the **Test** tab.
 - b. Using the default test settings, choose **Invoke**.
 - c. If the test succeeded, in the **Execution results**, open the **Details** view. You should see the policy document that the function returned.

If the test failed or you don't see a policy document, review the code to find and correct the errors.

Step 2: Create a public and private key pair for your custom authorizer

Your custom authorizer requires a public and private key to authenticate it. The commands in this section use OpenSSL tools to create this key pair.

To create the public and private key pair for your custom authorizer

1. Create the private key file.

```
openssl genrsa -out private-key.pem 4096
```

2. Verify the private key file you just created.

```
openssl rsa -check -in private-key.pem -noout
```

If the command doesn't display any errors, the private key file is valid.

3. Create the public key file.

```
openssl rsa -in private-key.pem -pubout -out public-key.pem
```

4. Verify the public key file.

```
openssl pkey -inform PEM -pubin -in public-key.pem -noout
```

If the command doesn't display any errors, the public key file is valid.

Step 3: Create a customer authorizer resource and its authorization

The AWS IoT custom authorizer is the resource that ties together all the elements created in the previous steps. In this section, you'll create a custom authorizer resource and give it permission to run the Lambda function you created earlier. You can create a custom authorizer resource by using the AWS IoT console, the AWS CLI, or the AWS API.

For this tutorial, you only need to create one custom authorizer. This section describes how to create by using the AWS IoT console and the AWS CLI, so you can use the method that is most convenient for you. There's no difference between the custom authorizer resources created by either method.

Create a customer authorizer resource

Choose one of these options to create your custom authorizer resource

- [Create a custom authorizer by using the AWS IoT console \(p. 235\)](#)
- [Create a custom authorizer using the AWS CLI \(p. 236\)](#)

To create a custom authorizer (console)

1. Open the [Custom authorizer page of the AWS IoT console](#), and choose **Create**.
2. In **Create custom authorizer**:
 - a. In **Name your custom authorizer**, enter **my-new-authorizer**.
 - b. In **Authorizer function**, choose the Lambda function you created earlier.
 - c. In **Token validation - optional**:
 - i. Check **Enable token signing**.
 - ii. In **Token header name (optional)**, enter **tokenKeyName**.
 - iii. In **Key name**, enter: **FirstKey**.
 - iv. In **Value**, enter the contents of the public-key.pem file. Be sure to include the lines from the file with **-----BEGIN PUBLIC KEY-----** and **-----END PUBLIC KEY-----** and don't add or remove any line feeds, carriage returns, or other characters from the file contents. The string that you enter should look something like this example.

```
-----BEGIN PUBLIC KEY-----  
MIICIJANBgkqhkiG9w0BAQEFAOCAg8AMIIICCgKCAgEAyEBzOk4vhN+3Lgs1vEWt  
sLCqNmt5Damas3bmiTRVq2gjRJ6KXGTGQChqArAJwL1a9dkS9+maaXC3vc6xzx9z  
QPu/vQOe5tyzz1MsKdmtFGxMqQ3qjEXAMPLEOmqyUKPP5mff58k6ePSfxAnzBH0q  
lg2HioefrpU5OSAnpuRAjYKofKjbc2Vrn6N2G7hv+IfTBvCElf0csals/Rk4phD5  
oa4Y0GHISRnevypg5C8n9Rrz91PWGqP6M/q5DNJJXjMyleG92hQgu1N696bn5Dw8  
FhedszFa6b2x6xrItZFzewNQkPMLMFhNrQIIyvshtT/F1LVCS5+v8AQ8UGGdfZmv  
QeqAMAF7WgagDMXcfgKSVU8yid2sIm56qsCLMvD2Sq8Lgzpey9N5ON1o1Cvldwvc
```

```
KrJJtgwW6hVqRGuShnownLpgG86M6neZ5sRMbVNZO8OzcobLngJ0Ibw9KkcUdklW
gvZ6HEJqBY2XE70iEXAMPLETPHzhqvK6Ei1HGxpHsXx6BNft582J1VpgYjXha8oa
/NN7l7zbj/euAb41IVtmX8JrD9z613d1iM5L8HluJlUzn62Q+VeNV2tdA7MfpfMC
8btGYladFAnitThaz6+F0VSBJPu7pZQoLnqyEp5zLMtF+kFl2yOBmGAP0RBivRd9
JWBUCG0bqcLQPeQyjbXSOfUCAwEAAQ==
-----END PUBLIC KEY-----
```

3. Check **Activate authorizer**.
4. Choose **Create authorizer**.
5. If the custom authorizer resource was created, you'll see the list of custom authorizers and your new custom authorizer should appear in the list and you can continue to the next section to test it.

If you see an error, review the error and try to create your custom authorizer again and double-check the entries. Note that each custom authorizer resource must have a unique name.

To create a custom authorizer (AWS CLI)

1. Substitute your values for `authorizer-function-arn` and `token-signing-public-keys`, and then run the following command:

```
aws iot create-authorizer \
--authorizer-name "my-new-authorizer" \
--token-key-name "tokenKeyName" \
--status ACTIVE \
--no-signing-disabled \
--authorizer-function-arn "arn:aws:lambda:Region:57EXAMPLE833:function:custom-auth-function" \
--token-signing-public-keys FirstKey="-----BEGIN PUBLIC KEY-----
MIICIJANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCAgEAvgEBzOk4vhN+3Lgs1vEWt
sLCqNm5Damas3bmiTRvg2gjRJ6KXGTGQChqArAJwL1a9dkS9+maaXC3vc6zxz9z
QPu/vQOe5tyzz1MsKdmtFGxMqQ3qjEXAMPLE0mqyUKPP5mff58k6ePSfXAnzBH0q
lg2HioefrpU5OSAnpuRAjYKofKjbc2Vrn6N2G7hV+JfTBvCElf0csals/Rk4phD5
oa4Y0GHISRnevypg5C8n9Rrz91PWGqP6M/q5DNJJKjMyleG92h0gu1N696bn5Dw8
FhedszFa6b2x6xrItZFzewNQkPMLMFhNrQIIyvshT/F1LVCS5+v8AQ8UGGDfZmv
QeqAMAF7WgagDMXcfgKVSVU8yid2sIm56gsCLMvD2Sq8Lgzpey9N5ON1o1Cvldwvc
KrJJtgwW6hVqRGuShnownLpgG86M6neZ5sRMbVNZO8OzcobLngJ0Ibw9KkcUdklW
gvZ6HEJqBY2XE70iEXAMPLETPHzhqvK6Ei1HGxpHsXx6BNft582J1VpgYjXha8oa
/NN7l7zbj/euAb41IVtmX8JrD9z613d1iM5L8HluJlUzn62Q+VeNV2tdA7MfpfMC
8btGYladFAnitThaz6+F0VSBJPu7pZQoLnqyEp5zLMtF+kFl2yOBmGAP0RBivRd9
JWBUCG0bqcLQPeQyjbXSOfUCAwEAAQ==
-----END PUBLIC KEY-----"
```

Where:

- The `authorizer-function-arn` value is the Amazon Resource Name (ARN) of the Lambda function you created for your custom authorizer.
- The `token-signing-public-keys` value includes the name of the key, `FirstKey`, and the contents of the `public-key.pem` file. Be sure to include the lines from the file with `-----BEGIN PUBLIC KEY-----` and `-----END PUBLIC KEY-----` and don't add or remove any line feeds, carriage returns, or other characters from the file contents.

Note: be careful entering the public key as any alteration to the public key value makes it unusable.

2. If the custom authorizer is created, the command returns the name and ARN of the new resource, such as the following.

```
{ "authorizerName": "my-new-authorizer", "authorizerArn": "arn:aws:iot:Region:57EXAMPLE833:authorizer/my-new-authorizer"
```

}

Save the `authorizerArn` value for use in the next step.

Remember that each custom authorizer resource must have a unique name.

Authorize the custom authorizer resource

In this section, you'll grant permission the custom authorizer resource that you just created permission to run the Lambda function

Grant permission to your Lambda function using the AWS CLI

1. After inserting your values, enter the following command. Note that the `statement-id` value must be unique. Replace `Id-1234` with another value if you have run this tutorial before or if you get a `ResourceConflictException` error.

```
aws lambda add-permission \
--function-name "custom-auth-function" \
--principal "iot.amazonaws.com" \
--action "lambda:InvokeFunction" \
--statement-id "Id-1234" \
--source-arn authorizerArn
```

2. If the command succeeds, it returns a permission statement, such as this example. You can continue to the next section to test the custom authorizer.

```
{  
  "Statement": "{\"Sid\": \"Id-1234\", \"Effect\": \"Allow\", \"Principal\": {\"Service\": \"iot.amazonaws.com\"}, \"Action\": \"lambda:InvokeFunction\", \"Resource\": \"arn:aws:lambda:Region:57EXAMPLE833:function:custom-auth-function\", \"Condition\": {\"ArnLike\": {\"AWS:SourceArn\": \"arn:aws:lambda:Region:57EXAMPLE833:function:custom-auth-function\"}}}"  
}
```

If the command doesn't succeed, it returns an error, such as this example. You'll need to review and correct the error before you continue.

```
An error occurred (AccessDeniedException) when calling the AddPermission operation:  
User: arn:aws:iam::57EXAMPLE833:user/EXAMPLE-1 is not authorized to perform:  
lambda:AddPer  
mission on resource: arn:aws:lambda:Region:57EXAMPLE833:function:custom-auth-function
```

Step 4: Test the authorizer by calling test-invoke-authorizer

With all the resources defined, in this section, you'll call `test-invoke-authorizer` from the command line to test the authorization pass.

Note that when invoking the authorizer from the command line, `protocolData` is not defined, so the authorizer will always return a DENY document. This test does, however, confirm that your custom authorizer and Lambda function are configured correctly--even if it doesn't fully test the Lambda function.

To test your custom authorizer and its Lambda function by using the AWS CLI

1. In the directory that has the `private-key.pem` file you created in a previous step, run the following command.

```
echo -n "tokenKeyValue" | openssl dgst -sha256 -sign private-key.pem | openssl base64 -A
```

This command creates a signature string to use in the next step. The signature string looks something like this:

```
dBykzlb+fo+JmSGdwoGr8dyC2qB/IyLefJJr+rbCvmu9Jl4KHA9DG+v
+MMWu09YSA86+64Y3Gt4tOykpZqn9mn
VB1wyxp+0bDZh8hmqUAUH3fw13fPjBvCa4cwNuLQNqBZzbCvsluv7i2IMjEg
+CPY0zrWt1jr9BikgGPDXWkjaceeh
bQHHTo357TegKs9pP30Uf4TrxypNmFswA5k7QIC01n4bIyRTm900yZ94R4bdJshNig1JePgnuOBvMGCEFE09jGjj
szEHfgAUAQIWXiVGQj16BU1xKpTGSiTAWheLKUjITOEXAMPLECK3aHKYKY
+d1vTvdthKtYHBq8MjhzJ0kggbt29V
QJCb8RilN/P5+vcVniSXWPplyB5jkYs9UvG08REoy64AtizfUhvSul/r/F3VV8ITtOp3aXiUtcspACi6ca
+tsDuX
f3LzCwQQF/YSUy02u5XkWn+sto6KCkpNlkD0wU8gl3+kOzxrthnQ8gEajd5Iylx230iqcXo3osjPha7JDyWM5o
+K
EWckTe91I1mokDr5sJ4JXixvnJTVSx1li49IalW4en1DAkc1a0s2U2UNm236EXAMPLElotyh7h
+f1FeloZlAWQFH
xRlxPsPqiVKs1ZIUClaZWprh/orDJplpiWfBgBIOgokJIDGP9gwhXIIk7zWrGmWpMK9o=
```

Copy this signature string to use in the next step. Be careful not to include any extra characters or leave any out.

2. In this command, replace the `token-signature` value with the signature string from the previous step and run this command to test your authorizer.

```
aws iot test-invoke-authorizer \
--authorizer-name my-new-authorizer \
--token tokenKeyValue \
--token-signature dBykzlb+fo+JmSGdwoGr8dyC2qB/IyLefJJr
+rbCvmu9Jl4KHA9DG+v+MMWu09YSA86+64Y3Gt4tOykpZqn9mnVB1wyxp
+0bDZh8hmqUAUH3fw13fPjBvCa4cwNuLQNqBZzbCvsluv7i2IMjEg
+CPY0zrWt1jr9BikgGPDXWkjaceehbQHHTo357TegKs9pP30Uf4TrxypNmFswA5k7QIC01n4bIyRTm900yZ94R4bdJshNig1JePg
+d1vTvdthKtYHBq8MjhzJ0kggbt29VQJCb8RilN/P5+vcVniSXWPplyB5jkYs9UvG08REoy64AtizfUhvSul/
r/F3VV8ITtOp3aXiUtcspACi6ca+tsDuXf3LzCwQQF/YSUy02u5XkWn
+sto6KCkpNlkD0wU8gl3+kOzxrthnQ8gEajd5Iylx230iqcXo3osjPha7JDyWM5o
+KEWckTe91I1mokDr5sJ4JXixvnJTVSx1li49IalW4en1DAkc1a0s2U2UNm236EXAMPLElotyh7h
+f1FeloZlAWQFHxRlxPsPqiVKs1ZIUClaZWprh/orDJplpiWfBgBIOgokJIDGP9gwhXIIk7zWrGmWpMK9o=
```

If the command is successful, it returns the information generated by your customer authorizer function, such as this example.

```
{
  "isAuthenticated": true,
  "principalId": "principalId",
  "policyDocuments": [
    {
      "Version": "2012-10-17",
      "Statement": [
        {
          "Action": "iot:*,"
        },
        {
          "Effect": "Deny",
          "Resource": "arn:aws:iot:Region:57EXAMPLE833:/*"
        }
      ]
    },
    {
      "refreshAfterInSeconds": 600,
      "disconnectAfterInSeconds": 3600
    }
}
```

If the command returns an error, review the error and double-check the commands you used in this section.

Step 5: Test publishing MQTT message using Postman

- To get your device data endpoint from the command line, call [describe-endpoint](#) as shown here

```
aws iot describe-endpoint --output text --endpoint-type iot:Data-ATS
```

Save this address for use as the *device_data_endpoint_address* in a later step.

- Open a new Postman window and create a new HTTP POST request.

- From your computer, open the Postman app.
- In Postman, in the **File** menu, choose **New...**
- In the **New** dialog box, choose **Request**.
- In Save request,
 - In **Request name** enter **Custom authorizer test request**.
 - In **Select a collection or folder to save to:** choose or create a collection into which to save this request.
 - Choose **Save to collection_name**.

- Create the POST request to test your custom authorizer.

- In the request method selector next to the URL field, choose **POST**.
- In the URL field, create the URL for your request by using the following URL with the *device_data_endpoint_address* from the [describe-endpoint](#) command in a previous step.

```
https://device_data_endpoint_address:443/topics/test/cust-auth/topic?  
qos=0&actionToken=allow
```

Note that this URL includes the `actionToken=allow` query parameter that will tell your Lambda function to return a policy document that allows access to AWS IoT. After you enter the URL, the query parameters also appear in the **Params** tab of Postman.

- In the **Auth** tab, in the **Type** field, choose **No Auth**.
- In the Headers tab:
 - If there's a **Host** key that's checked, uncheck this one.
 - At the bottom of the list of headers add these new headers and confirm they are checked. Replace the **Host** value with your *device_data_endpoint_address* and the **x-amz-customauthorizer-signature** value with the signature string that you used with the `test-invoke-authorize` command in the previous section.

Key	Value
x-amz-customauthorizer-name	my-new-authorizer
Host	<i>device_data_endpoint_address</i>
tokenKeyName	tokenKeyValue
x-amz-customauthorizer-signature	<i>dBwykzlb+fo+JmSGdwoGr8dyC2qB/ IyLefJJr+rbCvmu9Jl4KHAA9DG+V +MMWu09YSA86+64Y3Gt4tOykPZqn9mnVB1wyxp +0bDZh8hmqUAUH3fwi3fPjBvCa4cwNuLQNqBZZbCvslu +CPY0zrWt1jr9BikgGPdxWkjaaehbQHHTo357TegKs9p +d1vTvdthKtYHBq8MjhzJ0kggbt29VQJCb8RilN/</i>

Key	Value
	P5+vcVniSXWPplyB5jkYs9UvG08REoy64AtizfUhvSul r/F3VV8ITtQp3aXiUtcspACi6ca +tsDuXf3LzCwQQF/YSUy02u5XkWn +sto6KCkpNlkD0wU8gl3+kOzxrthnQ8gEajd5Iylx230 +KEWckTe91I1mokDr5sJ4JXixvnJTVSx1li49IalW4en +fLFelozlAWQFHxRlxspqivVKS1ZIUClazWprh/ orDJplpiWfBgBI0gokJIDGP9gwhXIIk7zWrGmWpMK9o=

- e. In the Body tab:
 - i. In the data format option box, choose **Raw**.
 - ii. In the data type list, choose **JavaScript**.
 - iii. In the text field, enter this JSON message payload for your test message:

```

{
  "data_mode": "test",
  "vibration": 200,
  "temperature": 40
}
```

- 4. Choose **Send** to send the request.

If the request was successful, it returns:

```

{
  "message": "OK",
  "traceId": "ff35c33f-409a-ea90-b06f-fbEXAMPLE25c"
}
```

The successful response indicates that your custom authorizer allowed the connection to AWS IoT and that the test message was delivered to broker in AWS IoT Core.

If it returns an error, review error message, the *device_data_endpoint_address*, the signature string, and the other header values.

Keep this request in Postman for use in the next section.

Step 6: View messages in MQTT test client

In the previous step, you sent simulated device messages to AWS IoT by using Postman. The successful response indicated that your custom authorizer allowed the connection to AWS IoT and that the test message was delivered to broker in AWS IoT Core. In this section, you'll use the MQTT test client in the AWS IoT console to see the message contents from that message as other devices and services might.

To see the test messages authorized by your custom authorizer

1. In the AWS IoT console, open the [MQTT test client](#).
2. In the **Subscribe to topic** tab, in **Topic filter**, enter **test/cust-auth/topic**, which is the message topic used in the Postman example from the previous section.
3. Choose **Subscribe**.

Keep this window visible for the next step.

4. In Postman, in the request you created for the previous section, choose **Send**.

Review the response to make sure it was successful. If not, troubleshoot the error as the previous section describes.

5. In the **MQTT test client**, you should see a new entry that shows the message topic and, if expanded, the message payload from the request you sent from Postman.

If you don't see your messages in the **MQTT test client**, here are some things to check:

- Make sure your Postman request returned successfully. If AWS IoT rejects the connection and returns an error, the message in the request doesn't get passed to the message broker.
- Make sure the AWS account and AWS Region used to open the AWS IoT console are the same as you're using in the Postman URL.
- Make sure you've entered the topic correctly in the **MQTT test client**. The topic filter is case-sensitive. If in doubt, you can also subscribe to the **#** topic, which subscribes to all MQTT messages that pass through the message broker the AWS account and AWS Region used to open the AWS IoT console.

Step 7: Review the results and next steps

In this tutorial:

- You created a Lambda function to be a custom authorizer handler
- You created a custom authorizer with token signing enabled
- You tested your custom authorizer using the **test-invoke-authorizer** command
- You published an MQTT topic by using [Postman](#) and validate the request with your custom authorizer
- You used the **MQTT test client** to view the messages sent from your Postman test

Next steps

After you send some messages from Postman to verify that the custom authorizer is working, try experimenting to see how changing different aspects of this tutorial affect the results. Here are some examples to get you started.

- Change the signature string so that it's no longer valid to see how unauthorized connection attempts are handled. You should get an error response, such as this one, and the message should not appear in the **MQTT test client**.

```
{  
  "message": "Forbidden",  
  "traceId": "15969756-a4a4-917c-b47a-5433e25b1356"  
}
```

- To learn more about how to find errors that might occur while you're developing and using AWS IoT rules, see [Monitoring AWS IoT \(p. 400\)](#).

Step 8: Clean up

If you'd like repeat this tutorial, you might need to remove some of your custom authorizers. Your AWS account can have only a limited number of custom authorizers configured at one time and you can get a `LimitExceeded` exception when you try to add a new one without removing an existing custom authorizer.

To remove a custom authorizer (console)

1. Open the [Custom authorizer page of the AWS IoT console](#), and in the list of custom authorizers, find the custom authorizer to remove.
2. Open the Custom authorizer details page and, from the **Actions** menu, choose **Edit**.
3. Uncheck the **Activate authorizer**, and then choose **Update**.

You can't delete a custom authorizer while it's active.
4. From the Custom authorizer details page, open the **Actions** menu, and choose **Delete**.

To remove a custom authorizer (AWS CLI)

1. List the custom authorizers that you have installed and find the name of the custom authorizer you want to delete.

```
aws iot list-authorizers
```

2. Set the custom authorizer to `inactive` by running this command after replacing `Custom_Auth_Name` with the `authorizerName` of the custom authorizer to delete.

```
aws iot update-authorizer --status INACTIVE --authorizer-name Custom_Auth_Name
```

3. Delete the custom authorizer by running this command after replacing `Custom_Auth_Name` with the `authorizerName` of the custom authorizer to delete.

```
aws iot delete-authorizer --authorizer-name Custom_Auth_Name
```

Tutorial: Monitoring soil moisture with AWS IoT and Raspberry Pi

This tutorial shows you how to use a [Raspberry Pi](#), a moisture sensor, and AWS IoT to monitor the soil moisture level for a house plant or garden. The Raspberry Pi runs code that reads the moisture level and temperature from the sensor and then sends the data to AWS IoT. You create a rule in AWS IoT that sends an email to an address subscribed to an Amazon SNS topic when the moisture level falls below a threshold.

Note

This tutorial might not be up to date. Some references might have been superseded since this topic was originally published.

Contents

- [Prerequisites \(p. 243\)](#)
- [Setting up AWS IoT \(p. 243\)](#)
 - [Step 1: Create the AWS IoT policy \(p. 243\)](#)
 - [Step 2: Create the AWS IoT thing, certificate, and private key \(p. 245\)](#)
 - [Step 3: Create an Amazon SNS topic and subscription \(p. 245\)](#)
 - [Step 4: Create an AWS IoT rule to send an email \(p. 245\)](#)
- [Setting up your Raspberry Pi and moisture sensor \(p. 246\)](#)

Prerequisites

To complete this tutorial, you need:

- An AWS account.
- An IAM user with administrator permissions.
- A development computer running Windows, macOS, Linux, or Unix to access the [AWS IoT console](#).
- A [Raspberry Pi 3B or 4B](#) running the latest [Raspbian OS](#). For installation instructions, see [Installing operating system images](#) on the Raspberry Pi website.
- A monitor, keyboard, mouse, and Wi-Fi network or Ethernet connection for your Raspberry Pi.
- A Raspberry Pi-compatible moisture sensor. The sensor used in this tutorial is an [Adafruit STEMMA I2C Capacitive Moisture Sensor](#) with a [JST 4-pin to female socket cable header](#).

Setting up AWS IoT

To complete this tutorial, you need to create the following resources. To connect a device to AWS IoT, you create an IoT thing, a device certificate, and an AWS IoT policy.

- An AWS IoT thing.

A thing represents a physical device (in this case, your Raspberry Pi) and contains static metadata about the device.

- A device certificate.

All devices must have a device certificate to connect to and authenticate with AWS IoT.

- An AWS IoT policy.

Each device certificate has one or more AWS IoT policies associated with it. These policies determine which AWS IoT resources the device can access.

- An AWS IoT root CA certificate.

Devices and other clients use an AWS IoT root CA certificate to authenticate the AWS IoT server with which they are communicating. For more information, see [Server authentication \(p. 280\)](#).

- An AWS IoT rule.

A rule contains a query and one or more rule actions. The query extracts data from device messages to determine if the message data should be processed. The rule action specifies what to do if the data matches the query.

- An Amazon SNS topic and topic subscription.

The rule listens for moisture data from your Raspberry Pi. If the value is below a threshold, it sends a message to the Amazon SNS topic. Amazon SNS sends that message to all email addresses subscribed to the topic.

Step 1: Create the AWS IoT policy

Create an AWS IoT policy that allows your Raspberry Pi to connect and send messages to AWS IoT.

1. In the [AWS IoT console](#), if a **Get started** button appears, choose it. Otherwise, in the navigation pane, expand **Secure**, and then choose **Policies**.
2. If a **You don't have any policies yet** dialog box appears, choose **Create a policy**. Otherwise, choose **Create**.

3. Enter a name for the AWS IoT policy (for example, **MoistureSensorPolicy**).
4. In the **Add statements** section, replace the existing policy with the following JSON. Replace **region** and **account** with your AWS Region and AWS account number.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iot:Connect",
            "Resource": "arn:aws:iot:region:account:client/RaspberryPi"
        },
        {
            "Effect": "Allow",
            "Action": "iot:Publish",
            "Resource": [
                "arn:aws:iot:region:account:topic/$aws/things/RaspberryPi/shadow/update",
                "arn:aws:iot:region:account:topic/$aws/things/RaspberryPi/shadow/delete",
                "arn:aws:iot:region:account:topic/$aws/things/RaspberryPi/shadow/get"
            ]
        },
        {
            "Effect": "Allow",
            "Action": "iot:Receive",
            "Resource": [
                "arn:aws:iot:region:account:topic/$aws/things/RaspberryPi/shadow/update/accepted",
                "arn:aws:iot:region:account:topic/$aws/things/RaspberryPi/shadow/delete/accepted",
                "arn:aws:iot:region:account:topic/$aws/things/RaspberryPi/shadow/get/accepted",
                "arn:aws:iot:region:account:topic/$aws/things/RaspberryPi/shadow/update/rejected",
                "arn:aws:iot:region:account:topic/$aws/things/RaspberryPi/shadow/delete/rejected"
            ]
        },
        {
            "Effect": "Allow",
            "Action": "iot:Subscribe",
            "Resource": [
                "arn:aws:iot:region:account:topicfilter/$aws/things/RaspberryPi/shadow/update/accepted",
                "arn:aws:iot:region:account:topicfilter/$aws/things/RaspberryPi/shadow/delete/accepted",
                "arn:aws:iot:region:account:topicfilter/$aws/things/RaspberryPi/shadow/get/accepted",
                "arn:aws:iot:region:account:topicfilter/$aws/things/RaspberryPi/shadow/update/rejected",
                "arn:aws:iot:region:account:topicfilter/$aws/things/RaspberryPi/shadow/delete/rejected"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:GetThingShadow",
                "iot:UpdateThingShadow",
                "iot:DeleteThingShadow"
            ],
            "Resource": "arn:aws:iot:region:account:thing/RaspberryPi"
        }
    ]
}
```

5. Choose **Create**.

Step 2: Create the AWS IoT thing, certificate, and private key

Create a thing in the AWS IoT registry to represent your Raspberry Pi.

1. In the [AWS IoT console](#), in the navigation pane, choose **Manage**, and then choose **Things**.
2. If a **You don't have any things yet** dialog box is displayed, choose **Register a thing**. Otherwise, choose **Create**.
3. On the **Creating AWS IoT things** page, choose **Create a single thing**.
4. On the **Add your device to the device registry** page, enter a name for your IoT thing (for example, **RaspberryPi**), and then choose **Next**. You can't change the name of a thing after you create it. To change a thing's name, you must create a new thing, give it the new name, and then delete the old thing.
5. On the **Add a certificate for your thing** page, choose **Create certificate**.
6. Choose the **Download** links to download the certificate, private key, and root CA certificate.

Important

This is the only time you can download your certificate and private key.

7. To activate the certificate, choose **Activate**. The certificate must be active for a device to connect to AWS IoT.
8. Choose **Attach a policy**.
9. For **Add a policy for your thing**, choose **MoistureSensorPolicy**, and then choose **Register Thing**.

Step 3: Create an Amazon SNS topic and subscription

Create an Amazon SNS topic and subscription.

1. From the [AWS SNS console](#), in the navigation pane, choose **Topics**, and then choose **Create topic**.
2. Enter a name for the topic (for example, **MoistureSensorTopic**).
3. Enter a display name for the topic (for example, **Moisture Sensor Topic**). This is the name displayed for your topic in the Amazon SNS console.
4. Choose **Create topic**.
5. In the Amazon SNS topic detail page, choose **Create subscription**.
6. For **Protocol**, choose **Email**.
7. For **Endpoint**, enter your email address.
8. Choose **Create subscription**.
9. Open your email client and look for a message with the subject **MoistureSensorTopic**. Open the email and click the **Confirm subscription** link.

Important

You won't receive any email alerts from this Amazon SNS topic until you confirm the subscription.

You should receive an email message with the text you typed.

Step 4: Create an AWS IoT rule to send an email

An AWS IoT rule defines a query and one or more actions to take when a message is received from a device. The AWS IoT rules engine listens for messages sent by devices and uses the data in the messages to determine if some action should be taken. For more information, see [Rules for AWS IoT \(p. 443\)](#).

In this tutorial, your Raspberry Pi publishes messages on `aws/things/RaspberryPi/shadow/update`. This is an internal MQTT topic used by devices and the Thing Shadow service. The Raspberry Pi publishes messages that have the following form:

```
{  
    "reported": {  
        "moisture" : moisture-reading,  
        "temp" : temperature-reading  
    }  
}
```

You create a query that extracts the moisture and temperature data from the incoming message. You also create an Amazon SNS action that takes the data and sends it to Amazon SNS topic subscribers if the moisture reading is below a threshold value.

Create an Amazon SNS rule

1. In the [AWS IoT console](#), in the navigation pane, choose **Act**. If a **You don't have any rules yet** dialog box appears, choose **Create a rule**. Otherwise, choose **Create**.
2. In the **Create a rule** page, enter a name for your rule (for example, **MoistureSensorRule**).
3. For **Description**, provide a short description for this rule (for example, **Sends an alert when soil moisture level readings are too low**).
4. Under **Rule query statement**, choose SQL version **2016-03-23**, and enter the following AWS IoT SQL query statement:

```
SELECT * FROM '$aws/things/RaspberryPi/shadow/update/accepted' WHERE  
state.reported.moisture < 400
```

This statement triggers the rule action when the moisture reading is less than 400.

Note

You might have to use a different value. After you have the code running on your Raspberry Pi, you can see the values that you get from your sensor by touching the sensor, placing it in water, or placing it in a planter.

5. Under **Set one or more actions**, choose **Add action**.
6. On the **Select an action** page, choose **Send a message as an SNS push notification**.
7. Scroll to the bottom of the page, and then choose **Configure action**.
8. On the **Configure action** page, for **SNS target**, choose **Select**, and then choose **LowMoistureTopic**.
9. For **Message format**, choose **RAW**.
10. Under **Choose or create a role to grant AWS IoT access to perform this action**, choose **Create role**. Enter a name for the role (for example, **LowMoistureTopicRole**), and then choose **Create role**.
11. Choose **Add action**.
12. Choose **Create rule**.

Setting up your Raspberry Pi and moisture sensor

Insert your microSD card into the Raspberry Pi, connect your monitor, keyboard, mouse, and, if you're not using Wi-Fi, Ethernet cable. Do not connect the power cable yet.

Connect the JST jumper cable to the moisture sensor. The other side of the jumper has four wires:

- Green: I2C SCL
- White: I2C SDA
- Red: power (3.5 V)
- Black: ground

Hold the Raspberry Pi with the Ethernet jack on the right. In this orientation, there are two rows of GPIO pins at the top. Connect the wires from the moisture sensor to the bottom row of pins in the following order. Starting at the left-most pin, connect red (power), white (SDA), and green (SCL). Skip one pin, and then connect the black (ground) wire. For more information, see [Python Computer Wiring](#).

Attach the power cable to the Raspberry Pi and plug the other end into a wall socket to turn it on.

Configure your Raspberry Pi

1. On **Welcome to Raspberry Pi**, choose **Next**.
2. Choose your country, language, timezone, and keyboard layout. Choose **Next**.
3. Enter a password for your Raspberry Pi, and then choose **Next**.
4. Choose your Wi-Fi network, and then choose **Next**. If you aren't using a Wi-Fi network, choose **Skip**.
5. Choose **Next** to check for software updates. When the updates are complete, choose **Restart** to restart your Raspberry Pi.

After your Raspberry Pi starts up, enable the I2C interface.

1. In the upper left corner of the Raspbian desktop, click the Raspberry icon, choose **Preferences**, and then choose **Raspberry Pi Configuration**.
2. On the **Interfaces** tab, for **I2C**, choose **Enable**.
3. Choose **OK**.

The libraries for the Adafruit STEMMA moisture sensor are written for CircuitPython. To run them on a Raspberry Pi, you need to install the latest version of Python 3.

1. Run the following commands from a command prompt to update your Raspberry Pi software:

```
sudo apt-get update
```

```
sudo apt-get upgrade
```

2. Run the following command to update your Python 3 installation:

```
sudo pip3 install --upgrade setuptools
```

3. Run the following command to install the Raspberry Pi GPIO libraries:

```
pip3 install RPI.GPIO
```

4. Run the following command to install the Adafruit Blinka libraries:

```
pip3 install adafruit-blinka
```

For more information, see [Installing CircuitPython Libraries on Raspberry Pi](#).

5. Run the following command to install the Adafruit Seesaw libraries:

```
sudo pip3 install adafruit-circuitpython-seesaw
```

6. Run the following command to install the AWS IoT Device SDK for Python:

```
pip3 install AWSIoTPythonSDK
```

Your Raspberry Pi now has all of the required libraries. Create a file called **moistureSensor.py** and copy the following Python code into the file:

```
from adafruit_seesaw.seesaw import Seesaw
from AWSIoTPythonSDK.MQTTLib import AWSIoTMQTTShadowClient
from board import SCL, SDA

import logging
import time
import json
import argparse
import busio

# Shadow JSON schema:
#
# {
#     "state": {
#         "desired":{
#             "moisture":<INT VALUE>,
#             "temp":<INT VALUE>
#         }
#     }
# }

# Function called when a shadow is updated
def customShadowCallback_Update(payload, responseStatus, token):

    # Display status and data from update request
    if responseStatus == "timeout":
        print("Update request " + token + " time out!")

    if responseStatus == "accepted":
        payloadDict = json.loads(payload)
        print("~~~~~")
        print("Update request with token: " + token + " accepted!")
        print("moisture: " + str(payloadDict["state"]["reported"]["moisture"]))
        print("temperature: " + str(payloadDict["state"]["reported"]["temp"]))
        print("~~~~~\n\n")

    if responseStatus == "rejected":
        print("Update request " + token + " rejected!")

# Function called when a shadow is deleted
def customShadowCallback_Delete(payload, responseStatus, token):

    # Display status and data from delete request
    if responseStatus == "timeout":
        print("Delete request " + token + " time out!")

    if responseStatus == "accepted":
        print("~~~~~")
        print("Delete request with token: " + token + " accepted!")
        print("~~~~~\n\n")

    if responseStatus == "rejected":
        print("Delete request " + token + " rejected!")

# Read in command-line parameters
def parseArgs():

    parser = argparse.ArgumentParser()
    parser.add_argument("-e", "--endpoint", action="store", required=True, dest="host",
    help="Your device data endpoint")
```

```

        parser.add_argument("-r", "--rootCA", action="store", required=True, dest="rootCAPath",
help="Root CA file path")
        parser.add_argument("-c", "--cert", action="store", dest="certificatePath",
help="Certificate file path")
        parser.add_argument("-k", "--key", action="store", dest="privateKeyPath", help="Private
key file path")
        parser.add_argument("-p", "--port", action="store", dest="port", type=int, help="Port
number override")
        parser.add_argument("-n", "--thingName", action="store", dest="thingName",
default="Bot", help="Targeted thing name")
        parser.add_argument("-id", "--clientId", action="store", dest="clientId",
default="basicShadowUpdater", help="Targeted client id")

    args = parser.parse_args()
    return args

# Configure logging
# AWSIoTMQTTShadowClient writes data to the log
def configureLogging():

    logger = logging.getLogger("AWSIoTPythonSDK.core")
    logger.setLevel(logging.DEBUG)
    streamHandler = logging.StreamHandler()
    formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
    streamHandler.setFormatter(formatter)
    logger.addHandler(streamHandler)

# Parse command line arguments
args = parseArgs()

if not args.certificatePath or not args.privateKeyPath:
    parser.error("Missing credentials for authentication.")
    exit(2)

# If no --port argument is passed, default to 8883
if not args.port:
    args.port = 8883

# Init AWSIoTMQTTShadowClient
myAWSIoTMQTTShadowClient = None
myAWSIoTMQTTShadowClient = AWSIoTMQTTShadowClient(args.clientId)
myAWSIoTMQTTShadowClient.configureEndpoint(args.host, args.port)
myAWSIoTMQTTShadowClient.configureCredentials(args.rootCAPath, args.privateKeyPath,
args.certificatePath)

# AWSIoTMQTTShadowClient connection configuration
myAWSIoTMQTTShadowClient.configureAutoReconnectBackoffTime(1, 32, 20)
myAWSIoTMQTTShadowClient.configureConnectDisconnectTimeout(10) # 10 sec
myAWSIoTMQTTShadowClient.configureMQTTOperationTimeout(5) # 5 sec

# Initialize Raspberry Pi's I2C interface
i2c_bus = busio.I2C(SCL, SDA)

# Intialize SeeSaw, Adafruit's Circuit Python library
ss = Seesaw(i2c_bus, addr=0x36)

# Connect to AWS IoT
myAWSIoTMQTTShadowClient.connect()

# Create a device shadow handler, use this to update and delete shadow document
deviceShadowHandler = myAWSIoTMQTTShadowClient.createShadowHandlerWithName(args.thingName,
True)

```

```
# Delete current shadow JSON doc
deviceShadowHandler.shadowDelete(customShadowCallback_Delete, 5)

# Read data from moisture sensor and update shadow
while True:

    # read moisture level through capacitive touch pad
    moistureLevel = ss.moisture_read()

    # read temperature from the temperature sensor
    temp = ss.get_temp()

    # Display moisture and temp readings
    print("Moisture Level: {}".format(moistureLevel))
    print("Temperature: {}".format(temp))

    # Create message payload
    payload = {"state":{"reported":{"moisture":str(moistureLevel),"temp":str(temp)}}}

    # Update shadow
    deviceShadowHandler.shadowUpdate(json.dumps(payload), customShadowCallback_Update, 5)
    time.sleep(1)
```

Save the file to a place you can find it. Run `moistureSensor.py` from the command line with the following parameters:

`endpoint`

Your custom AWS IoT endpoint. For more information, see [Device Shadow REST API \(p. 615\)](#).

`rootCA`

The full path to your AWS IoT root CA certificate.

`cert`

The full path to your AWS IoT device certificate.

`key`

The full path to your AWS IoT device certificate private key.

`thingName`

Your thing name (in this case, `RaspberryPi`).

`clientId`

The MQTT client ID. Use `RaspberryPi`.

The command line should look like this:

```
python3 moistureSensor.py --endpoint your-endpoint --rootCA ~/certs/
AmazonRootCA1.pem --cert ~/certs/raspberrypi-certificate.pem.crt --key ~/certs/
raspberrypi-private.pem.key --thingName RaspberryPi --clientId RaspberryPi
```

Try touching the sensor, putting it in a planter, or putting it in a glass of water to see how the sensor responds to various levels of moisture. If needed, you can change the threshold value in the `MoistureSensorRule`. When the moisture sensor reading goes below the value specified in your rule's SQL query statement, AWS IoT publishes a message to the Amazon SNS topic. You should receive an email message that contains the moisture and temperature data.

After you have verified receipt of email messages from Amazon SNS, press **CTRL+C** to stop the Python program. It is unlikely that the Python program will send enough messages to incur charges, but it is a best practice to stop the program when you are done.

Managing devices with AWS IoT

AWS IoT provides a registry that helps you manage *things*. A thing is a representation of a specific device or logical entity. It can be a physical device or sensor (for example, a light bulb or a switch on a wall). It can also be a logical entity like an instance of an application or physical entity that does not connect to AWS IoT but is related to other devices that do (for example, a car that has engine sensors or a control panel).

Information about a thing is stored in the registry as JSON data. Here is an example thing:

```
{  
    "version": 3,  
    "thingName": "MyLightBulb",  
    "defaultClientId": "MyLightBulb",  
    "thingTypeName": "LightBulb",  
    "attributes": {  
        "model": "123",  
        "wattage": "75"  
    }  
}
```

Things are identified by a name. Things can also have attributes, which are name-value pairs you can use to store information about the thing, such as its serial number or manufacturer.

A typical device use case involves the use of the thing name as the default MQTT client ID. Although we do not enforce a mapping between a thing's registry name and its use of MQTT client IDs, certificates, or shadow state, we recommend you choose a thing name and use it as the MQTT client ID for both the registry and the Device Shadow service. This provides organization and convenience to your IoT fleet without removing the flexibility of the underlying device certificate model or shadows.

You do not need to create a thing in the registry to connect a device to AWS IoT. Adding things to the registry allows you to manage and search for devices more easily.

How to manage things with the registry

You use the AWS IoT console, AWS IoT API, or the AWS CLI to interact with the registry. The following sections show how to use the CLI to work with the registry.

When naming your thing objects:

- You should not use personally identifiable information in your thing name. The thing name can appear in unencrypted communications and reports.
- You should not use a colon character (:) in a thing name. The colon character is used as a delimiter by other AWS IoT services and this can cause them to parse strings with thing names incorrectly.

Create a thing

The following command shows how to use the AWS IoT **CreateThing** command from the CLI to create a thing. You can't change a thing's name after you create it. To change a thing's name, you must create a new thing, give it the new name, and then delete the old thing.

```
$ aws iot create-thing --thing-name "MyLightBulb" --attribute-payload "{\"attributes\":{\"wattage\":\"75\", \"model\":\"123\"}}"
```

The **CreateThing** command displays the name and Amazon Resource Name (ARN) of your new thing:

```
{  
    "thingArn": "arn:aws:iot:us-east-1:123456789012:thing/MyLightBulb",  
    "thingName": "MyLightBulb",  
    "thingId": "12345678abcdefghijklmnopqrstuvwxyz"  
}
```

Note

We do not recommend using personally identifiable information in your thing names.

List things

You can use the **ListThings** command to list all things in your account:

```
$ aws iot list-things
```

```
{  
    "things": [  
        {  
            "attributes": {  
                "model": "123",  
                "wattage": "75"  
            },  
            "version": 1,  
            "thingName": "MyLightBulb"  
        },  
        {  
            "attributes": {  
                "numOfStates": "3"  
            },  
            "version": 11,  
            "thingName": "MyWallSwitch"  
        }  
    ]  
}
```

You can use the **ListThings** command to search for all things of a specific thing type:

```
$ aws iot list-things --thing-type-name "LightBulb"
```

```
{  
    "things": [  
        {  
            "thingTypeName": "LightBulb",  
            "attributes": {  
                "model": "123",  
                "wattage": "75"  
            },  
            "version": 1,  
            "thingName": "MyRGBLight"  
        },  
        {  
            "thingTypeName": "LightBulb",  
            "attributes": {  
                "model": "123",  
                "wattage": "75"  
            },  
            "version": 1,  
            "thingName": "MySmartBulb"  
        }  
    ]  
}
```

```
        "model": "123",
        "wattage": "75"
    },
    "version": 1,
    "thingName": "MySecondLightBulb"
}
]
```

You can use the **ListThings** command to search for all things that have an attribute with a specific value. This command searches only searchable attributes.

```
$ aws iot list-things --attribute-name "wattage" --attribute-value "75"
```

```
{
    "things": [
        {
            "thingTypeName": "StopLight",
            "attributes": {
                "model": "123",
                "wattage": "75"
            },
            "version": 3,
            "thingName": "MyLightBulb"
        },
        {
            "thingTypeName": "LightBulb",
            "attributes": {
                "model": "123",
                "wattage": "75"
            },
            "version": 1,
            "thingName": "MyRGBLight"
        },
        {
            "thingTypeName": "LightBulb",
            "attributes": {
                "model": "123",
                "wattage": "75"
            },
            "version": 1,
            "thingName": "MySecondLightBulb"
        }
    ]
}
```

If [fleet indexing \(p. 732\)](#) is enabled, you can use the **search-index** command to search on searchable and non-searchable thing attributes, device shadow values, and connectivity values. For more information about what you can query by using the **search-index** command, see [Example thing queries \(p. 750\)](#) and the CLI reference on the **search-index** command.

Describe things

You can use the **DescribeThing** command to display more detailed information about a thing:

```
$ aws iot describe-thing --thing-name "MyLightBulb"
{
    "version": 3,
    "thingName": "MyLightBulb",
```

```
"thingArn": "arn:aws:iot:us-east-1:123456789012:thing/MyLightBulb",
"thingId": "12345678abcdefghijklmnpqrstuvwxyz",
"defaultClientId": "MyLightBulb",
"thingTypeName": "StopLight",
"attributes": {
    "model": "123",
    "wattage": "75"
}
}
```

Update a thing

You can use the **UpdateThing** command to update a thing. Note that this command updates only the thing's attributes. You can't change a thing's name. To change a thing's name, you must create a new thing, give it the new name, and then delete the old thing.

```
$ aws iot update-thing --thing-name "MyLightBulb" --attribute-payload "{\"attributes\": {\"wattage\":\"150\", \"model\":\"456\"}}"
```

The **UpdateThing** command does not produce output. You can use the **DescribeThing** command to see the result:

```
$ aws iot describe-thing --thing-name "MyLightBulb"
{
    "attributes": {
        "model": "456",
        "wattage": "150"
    },
    "version": 2,
    "thingName": "MyLightBulb"
}
```

Delete a thing

You can use the **DeleteThing** command to delete a thing:

```
$ aws iot delete-thing --thing-name "MyThing"
```

This command returns successfully with no error if the deletion is successful or you specify a thing that doesn't exist.

Attach a principal to a thing

A physical device must have an X.509 certificate to communicate with AWS IoT. You can associate the certificate on your device with the thing in the registry that represents your device. To attach a certificate to your thing, use the **AttachThingPrincipal** command:

```
$ aws iot attach-thing-principal --thing-name "MyLightBulb" --principal "arn:aws:iot:us-east-1:123456789012:cert/a0c01f5835079de0a7514643d68ef8414ab739a1e94ee4162977b02b12842847"
```

The **AttachThingPrincipal** command does not produce any output.

Detach a principal from a thing

You can use the **DetachThingPrincipal** command to detach a certificate from a thing:

```
$ aws iot detach-thing-principal --thing-name "MyLightBulb" --principal "arn:aws:iot:us-east-1:123456789012:cert/a0c01f5835079de0a7514643d68ef8414ab739a1e94ee4162977b02b12842847"
```

The **DetachThingPrincipal** command does not produce any output.

Thing types

Thing types allow you to store description and configuration information that is common to all things associated with the same thing type. This simplifies the management of things in the registry. For example, you can define a LightBulb thing type. All things associated with the LightBulb thing type share a set of attributes: serial number, manufacturer, and wattage. When you create a thing of type LightBulb (or change the type of an existing thing to LightBulb) you can specify values for each of the attributes defined in the LightBulb thing type.

Although thing types are optional, their use makes it easier to discover things.

- Things with a thing type can have up to 50 attributes.
- Things without a thing type can have up to three attributes.
- A thing can be associated with only one thing type.
- There is no limit on the number of thing types you can create in your account.

Thing types are immutable. You cannot change a thing type name after it has been created. You can deprecate a thing type at any time to prevent new things from being associated with it. You can also delete thing types that have no things associated with them.

Create a thing type

You can use the **CreateThingType** command to create a thing type:

```
$ aws iot create-thing-type  
    --thing-type-name "LightBulb" --thing-type-properties  
    "thingTypeDescription=light bulb type, searchableAttributes=wattage,model"
```

The **CreateThingType** command returns a response that contains the thing type and its ARN:

```
{  
    "thingTypeName": "LightBulb",  
    "thingTypeId": "df9c2d8c-894d-46a9-8192-9068d01b2886",  
    "thingTypeArn": "arn:aws:iot:us-west-2:123456789012:thingtype/LightBulb"  
}
```

List thing types

You can use the **ListThingTypes** command to list thing types:

```
$ aws iot list-thing-types
```

The **ListThingTypes** command returns a list of the thing types defined in your AWS account:

```
{
```

```
"thingTypes": [
    {
        "thingTypeName": "LightBulb",
        "thingTypeProperties": {
            "searchableAttributes": [
                "wattage",
                "model"
            ],
            "thingTypeDescription": "light bulb type"
        },
        "thingTypeMetadata": {
            "deprecated": false,
            "creationDate": 1468423800950
        }
    }
]
```

Describe a thing type

You can use the **DescribeThingType** command to get information about a thing type:

```
$ aws iot describe-thing-type --thing-type-name "LightBulb"
```

The **DescribeThingType** command returns information about the specified type:

```
{
    "thingTypeProperties": {
        "searchableAttributes": [
            "model",
            "wattage"
        ],
        "thingTypeDescription": "light bulb type"
    },
    "thingTypeId": "df9c2d8c-894d-46a9-8192-9068d01b2886",
    "thingTypeArn": "arn:aws:iot:us-west-2:123456789012:thingtype/LightBulb",
    "thingTypeName": "LightBulb",
    "thingTypeMetadata": {
        "deprecated": false,
        "creationDate": 1544466338.399
    }
}
```

Associate a thing type with a thing

You can use the **CreateThing** command to specify a thing type when you create a thing:

```
$ aws iot create-thing --thing-name "MyLightBulb" --thing-type-name "LightBulb" --attribute-payload "{\"attributes\": {\"wattage\": \"75\", \"model\": \"123\"}}"
```

You can use the **UpdateThing** command at any time to change the thing type associated with a thing:

```
$ aws iot update-thing --thing-name "MyLightBulb"
--thing-type-name "LightBulb" --attribute-payload "{\"attributes\": {\"wattage\": \"75\", \"model\": \"123\"}}"
```

You can also use the **UpdateThing** command to disassociate a thing from a thing type.

Deprecate a thing type

Thing types are immutable. They cannot be changed after they are defined. You can, however, deprecate a thing type to prevent users from associating any new things with it. All existing things associated with the thing type are unchanged.

To deprecate a thing type, use the **DeprecateThingType** command:

```
$ aws iot deprecate-thing-type --thing-type-name "myThingType"
```

You can use the **DescribeThingType** command to see the result:

```
$ aws iot describe-thing-type --thing-type-name "StopLight":
```

```
{  
    "thingTypeName": "StopLight",  
    "thingTypeProperties": {  
        "searchableAttributes": [  
            "wattage",  
            "numOfLights",  
            "model"  
        ],  
        "thingTypeDescription": "traffic light type",  
    },  
    "thingTypeMetadata": {  
        "deprecated": true,  
        "creationDate": 1468425854308,  
        "deprecationDate": 1468446026349  
    }  
}
```

Deprecating a thing type is a reversible operation. You can undo a deprecation by using the **--undo-deprecate** flag with the **DeprecateThingType** CLI command:

```
$ aws iot deprecate-thing-type --thing-type-name "myThingType" --undo-deprecate
```

You can use the **DescribeThingType** CLI command to see the result:

```
$ aws iot describe-thing-type --thing-type-name "StopLight":
```

```
{  
    "thingTypeName": "StopLight",  
    "thingTypeArn": "arn:aws:iot:us-east-1:123456789012:thingtype/StopLight",  
    "thingTypeId": "12345678abcdefghijklmnpqrstuvwxyz",  
    "thingTypeProperties": {  
        "searchableAttributes": [  
            "wattage",  
            "numOfLights",  
            "model"  
        ],  
        "thingTypeDescription": "traffic light type",  
    },  
    "thingTypeMetadata": {  
        "deprecated": false,  
        "creationDate": 1468425854308,  
    }  
}
```

}

Delete a thing type

You can delete thing types only after they have been deprecated. To delete a thing type, use the [DeleteThingType](#) command:

```
$ aws iot delete-thing-type --thing-type-name "StopLight"
```

Note

You must wait five minutes after you deprecate a thing type before you can delete it.

Static thing groups

Static thing groups allow you to manage several things at once by categorizing them into groups. Static thing groups contain a group of things that are managed by using the console, CLI, or the API. [Dynamic thing groups \(p. 267\)](#), on the other hand, contain things that match a specified query. Static thing groups can also contain other static thing groups — you can build a hierarchy of groups. You can attach a policy to a parent group and it is inherited by its child groups, and by all of the things in the group and in its child groups. This makes control of permissions easy for large numbers of things.

Here are the things you can do with static thing groups:

- Create, describe or delete a group.
- Add a thing to a group, or to more than one group.
- Remove a thing from a group.
- List the groups you have created.
- List all child groups of a group (its direct and indirect descendants.)
- List the things in a group, including all the things in its child groups.
- List all ancestor groups of a group (its direct and indirect parents.)
- Add, delete or update the attributes of a group. (Attributes are name-value pairs you can use to store information about a group.)
- Attach or detach a policy to or from a group.
- List the policies attached to a group.
- List the policies inherited by a thing (by virtue of the policies attached to its group, or one of its parent groups.)
- Configure logging options for things in a group. See [Configure AWS IoT logging \(p. 400\)](#).
- Create jobs that are sent to and executed on every thing in a group and its child groups. See [Jobs \(p. 637\)](#).

Here are some limitations of static thing groups:

- A group can have at most one direct parent.
- If a group is a child of another group, you must specify this at the time it is created.
- You can't change a group's parent later, so be sure to plan your group hierarchy and create a parent group before you create any child groups it contains.
- The number of groups to which a thing can belong is [limited](#).

- You cannot add a thing to more than one group in the same hierarchy. (In other words, you cannot add a thing to two groups that share a common parent.)
- You cannot rename a group.
- Thing group names can't contain international characters, such as û, é and ñ.
- You should not use personally identifiable information in your thing group name. The thing group name can appear in unencrypted communications and reports.
- You should not use a colon character (:) in a thing group name. The colon character is used as a delimiter by other AWS IoT services and this can cause them to parse strings with thing group names incorrectly.

Attaching and detaching policies to groups can enhance the security of your AWS IoT operations in a number of significant ways. The per-device method of attaching a policy to a certificate, which is then attached to a thing, is time consuming and makes it difficult to quickly update or change policies across a fleet of devices. Having a policy attached to the thing's group saves steps when it is time to rotate the certificates on a thing. And policies are dynamically applied to things when they change group membership, so you aren't required to re-create a complex set of permissions each time a device changes membership in a group.

Create a static thing group

Use the **CreateThingGroup** command to create a static thing group:

```
$ aws iot create-thing-group --thing-group-name LightBulbs
```

The **CreateThingGroup** command returns a response that contains the static thing group's name, ID, and ARN:

```
{  
    "thingGroupName": "LightBulbs",  
    "thingGroupId": "abcdefghijklmnopqrstuvwxyz12345678qrstuvwxyz",  
    "thingGroupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/LightBulbs"  
}
```

Note

We do not recommend using personally identifiable information in your thing group names.

Here is an example that specifies a parent of the static thing group when it is created:

```
$ aws iot create-thing-group --thing-group-name RedLights --parent-group-name LightBulbs
```

As before, the **CreateThingGroup** command returns a response that contains the static thing group's name,, ID, and ARN:

```
{  
    "thingGroupName": "RedLights",  
    "thingGroupId": "abcdefghijklmnopqrstuvwxyz12345678qrstuvwxyz",  
    "thingGroupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/RedLights",  
}
```

Important

Keep in mind the following limits when creating thing group hierarchies:

- A thing group can have only one direct parent.
- The number of direct child groups a thing group can have is [limited](#).

- The maximum depth of a group hierarchy is [limited](#).
- The number of attributes a thing group can have is [limited](#). (Attributes are name-value pairs you can use to store information about a group.) The lengths of each attribute name and each value are also [limited](#).

Describe a thing group

You can use the **DescribeThingGroup** command to get information about a thing group:

```
$ aws iot describe-thing-group --thing-group-name RedLights
```

The **DescribeThingGroup** command returns information about the specified group:

```
{  
    "thingGroupName": "RedLights",  
    "thingGroupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/RedLights",  
    "thingGroupId": "12345678abcdefghijklmnopqrstuvwxyz",  
    "version": 1,  
    "thingGroupMetadata": {  
        "creationDate": 1478299948.882  
        "parentGroupName": "Lights",  
        "rootToParentThingGroups": [  
            {  
                "groupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/ShinyObjects",  
                "groupName": "ShinyObjects"  
            },  
            {  
                "groupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/LightBulbs",  
                "groupName": "LightBulbs"  
            }  
        ]  
    },  
    "thingGroupProperties": {  
        "attributePayload": {  
            "attributes": {  
                "brightness": "3400_lumens"  
            }  
        },  
        "thingGroupDescription": "string"  
    },  
}
```

Add a thing to a static thing group

You can use the **AddThingToThingGroup** command to add a thing to a static thing group:

```
$ aws iot add-thing-to-thing-group --thing-name MyLightBulb --thing-group-name RedLights
```

The **AddThingToThingGroup** command does not produce any output.

Important

You can add a thing to a maximum of 10 groups. But you cannot add a thing to more than one group in the same hierarchy. (In other words, you cannot add a thing to two groups which share a common parent.)

If a thing belongs to as many thing groups as possible, and one or more of those groups is a dynamic thing group, you can use the [overrideDynamicGroups](#) flag to make static groups take priority over dynamic groups.

Remove a thing from a static thing group

You can use the **RemoveThingFromThingGroup** command to remove a thing from a group:

```
$ aws iot remove-thing-from-thing-group --thing-name MyLightBulb --thing-group-name RedLights
```

The **RemoveThingFromThingGroup** command does not produce any output.

List things in a thing group

You can use the **ListThingsInThingGroup** command to list the things that belong to a group:

```
$ aws iot list-things-in-thing-group --thing-group-name LightBulbs
```

The **ListThingsInThingGroup** command returns a list of the things in the given group:

```
{  
    "things": [  
        "TestThingA"  
    ]  
}
```

With the **--recursive** parameter, you can list things belonging to a group and those in any of its child groups:

```
$ aws iot list-things-in-thing-group --thing-group-name LightBulbs --recursive
```

```
{  
    "things": [  
        "TestThingA",  
        "MyLightBulb"  
    ]  
}
```

Note

This operation is [eventually consistent](#). In other words, changes to the thing group might not be reflected immediately.

List thing groups

You can use the **ListThingGroups** command to list your account's thing groups:

```
$ aws iot list-thing-groups
```

The **ListThingGroups** command returns a list of the thing groups in your AWS account:

```
{  
    "thingGroups": [  
        {  
            "groupName": "LightBulbs",  
            "groupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/LightBulbs"  
        },  
    ]  
}
```

```
{  
    "groupName": "RedLights",  
    "groupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/RedLights"  
},  
{  
    "groupName": "RedLEDLights",  
    "groupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/RedLEDLights"  
},  
{  
    "groupName": "RedIncandescentLights",  
    "groupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/  
RedIncandescentLights"  
}  
]  
}
```

Use the optional filters to list those groups that have a given group as parent (`--parent-group`) or groups whose name begins with a given prefix (`--name-prefix-filter`). The `--recursive` parameter allows you to list all children groups, not just direct child groups of a thing group:

```
$ aws iot list-thing-groups --parent-group LightBulbs
```

In this case, the **ListThingGroups** command returns a list of the direct child groups of the thing group defined in your AWS account:

```
{  
    "childGroups": [  
        {  
            "groupName": "RedLights",  
            "groupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/RedLights"  
        }  
    ]  
}
```

Use the `--recursive` parameter with the **ListThingGroups** command to list all child groups of a thing group, not just direct children:

```
$ aws iot list-thing-groups --parent-group LightBulbs --recursive
```

The **ListThingGroups** command returns a list of all child groups of the thing group:

```
{  
    "childGroups": [  
        {  
            "groupName": "RedLights",  
            "groupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/RedLights"  
        },  
        {  
            "groupName": "RedLEDLights",  
            "groupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/RedLEDLights"  
        },  
        {  
            "groupName": "RedIncandescentLights",  
            "groupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/  
RedIncandescentLights"  
        }  
    ]  
}
```

```
    ]  
}
```

Note

This operation is [eventually consistent](#). In other words, changes to the thing group might not be reflected immediately.

List groups for a thing

You can use the **ListThingGroupsForThing** command to list the groups a thing belongs to, including any parent groups:

```
$ aws iot list-thing-groups-for-thing --thing-name MyLightBulb
```

The **ListThingGroupsForThing** command returns a list of the thing groups this thing belongs to, including any parent groups:

```
{  
  "thingGroups": [  
    {  
      "groupName": "LightBulbs",  
      "groupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/LightBulbs"  
    },  
    {  
      "groupName": "RedLights",  
      "groupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/RedLights"  
    },  
    {  
      "groupName": "ReplaceableObjects",  
      "groupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/ReplaceableObjects"  
    }  
  ]  
}
```

Update a static thing group

You can use the **UpdateThingGroup** command to update the attributes of a static thing group:

```
$ aws iot update-thing-group --thing-group-name "LightBulbs" --thing-group-properties  
  "thingGroupDescription=\\"this is a test group\\", attributePayload=\\"{\\"attributes  
\\\"=\\\"Owner\\\"=\\"150\\\", \\\"modelNames\\\"=\\"456\\\"}\\\"}"
```

The **UpdateThingGroup** command returns a response that contains the group's version number after the update:

```
{  
  "version": 4  
}
```

Note

The number of attributes that a thing can have is [limited](#).

Delete a thing group

To delete a thing group, use the **DeleteThingGroup** command:

```
$ aws iot delete-thing-group --thing-group-name "RedLights"
```

The **DeleteThingGroup** command does not produce any output.

Important

If you try to delete a thing group that has child thing groups, you receive an error:

```
A client error (InvalidRequestException) occurred when calling the
DeleteThingGroup
operation: Cannot delete thing group : RedLights when there are still child groups
attached to it.
```

You must delete any child groups first before you delete the group.

You can delete a group that has child things, but any permissions granted to the things by membership in the group no longer apply. Before deleting a group that has a policy attached, check carefully that removing those permissions would not stop the things in the group from being able to function properly. Also, note that commands that show which groups a thing belongs to (for example, **ListGroupsForThing**) might continue to show the group while records in the cloud are being updated.

Attach a policy to a static thing group

You can use the **AttachPolicy** command to attach a policy to a static thing group and so, by extension, to all things in that group and things in any of its child groups:

```
$ aws iot attach-policy \
--target "arn:aws:iot:us-west-2:123456789012:thinggroup/LightBulbs" \
--policy-name "myLightBulbPolicy"
```

The **AttachPolicy** command does not produce any output

Important

You can attach a maximum number of two policies to a group.

Note

We do not recommend using personally identifiable information in your policy names.

The **--target** parameter can be a thing group ARN (as above), a certificate ARN, or an Amazon Cognito Identity. For more information about policies, certificates and authentication, see [Authentication \(p. 279\)](#).

For more information, see [AWS IoT Core policies](#).

Detach a policy from a static thing group

You can use the **DetachPolicy** command to detach a policy from a group and so, by extension, to all things in that group and things in any of its child groups:

```
$ aws iot detach-policy --target "arn:aws:iot:us-west-2:123456789012:thinggroup/LightBulbs"
--policy-name "myLightBulbPolicy"
```

The **DetachPolicy** command does not produce any output.

List the policies attached to a static thing group

You can use the **ListAttachedPolicies** command to list the policies attached to a static thing group:

```
$ aws iot list-attached-policies --target "arn:aws:iot:us-west-2:123456789012:thinggroup/RedLights"
```

The `--target` parameter can be a thing group ARN (as above), a certificate ARN, or an Amazon Cognito identity.

Add the optional `--recursive` parameter to include all policies attached to the group's parent groups.

The **ListAttachedPolicies** command returns a list of policies:

```
{  
    "policies": [  
        "MyLightBulbPolicy"  
        ...  
    ]  
}
```

List the groups for a policy

You can use the **ListTargetsForPolicy** command to list the targets, including any groups, that a policy is attached to:

```
$ aws iot list-targets-for-policy --policy-name "MyLightBulbPolicy"
```

Add the optional `--page-size number` parameter to specify the maximum number of results to be returned for each query, and the `--marker string` parameter on subsequent calls to retrieve the next set of results, if any.

The **ListTargetsForPolicy** command returns a list of targets and the token to use to retrieve more results:

```
{  
    "nextMarker": "string",  
    "targets": [ "string" ... ]  
}
```

Get effective policies for a thing

You can use the **GetEffectivePolicies** command to list the policies in effect for a thing, including the policies attached to any groups the thing belongs to (whether the group is a direct parent or indirect ancestor):

```
$ aws iot get-effective-policies \  
--thing-name "MyLightBulb" \  
--principal "arn:aws:iot:us-east-1:123456789012:cert/  
a0c01f5835079de0a7514643d68ef8414ab739a1e94ee4162977b02b12842847"
```

Use the `--principal` parameter to specify the ARN of the certificate attached to the thing. If you are using Amazon Cognito identity authentication, use the `--cognito-identity-pool-id` parameter and, optionally, add the `--principal` parameter to specify an Amazon Cognito identity. If you specify only the `--cognito-identity-pool-id`, the policies associated with that identity pool's role for unauthenticated users are returned. If you use both, the policies associated with that identity pool's role for authenticated users are returned.

The `--thing-name` parameter is optional and can be used instead of the `--principal` parameter. When used, the policies attached to any group the thing belongs to, and the policies attached to any parent groups of these groups (up to the root group in the hierarchy) are returned.

The **GetEffectivePolicies** command returns a list of policies:

```
{
    "effectivePolicies": [
        {
            "policyArn": "string",
            "policyDocument": "string",
            "policyName": "string"
        }
        ...
    ]
}
```

Test authorization for MQTT actions

You can use the **TestAuthorization** command to test whether an [MQTT](#) action (Publish, Subscribe) is allowed for a thing:

```
aws iot test-authorization \
--principal "arn:aws:iot:us-east-1:123456789012:cert/
a0c01f5835079de0a7514643d68ef8414ab739a1e94ee4162977b02b12842847" \
--auth-infos "{\"actionType\": \"PUBLISH\", \"resources\": [ \"arn:aws:iot:us-
east-1:123456789012:topic/my/topic\"]}"
```

Use the `--principal` parameter to specify the ARN of the certificate attached to the thing. If using Amazon Cognito Identity authentication, specify a Cognito Identity as the `--principal` or use the `--cognito-identity-pool-id` parameter, or both. (If you specify only the `--cognito-identity-pool-id` then the policies associated with that identity pool's role for unauthenticated users are considered. If you use both, the policies associated with that identity pool's role for authenticated users are considered.

Specify one or more MQTT actions you want to test by listing sets of resources and action types following the `--auth-infos` parameter. The `actionType` field should contain "PUBLISH", "SUBSCRIBE", "RECEIVE", or "CONNECT". The `resources` field should contain a list of resource ARNs. See [AWS IoT Core policies \(p. 314\)](#) for more information.

You can test the effects of adding policies by specifying them with the `--policy-names-to-add` parameter. Or you can test the effects of removing policies by them with the `--policy-names-to-skip` parameter.

You can use the optional `--client-id` parameter to further refine your results.

The **TestAuthorization** command returns details on actions that were allowed or denied for each set of `--auth-infos` queries you specified:

```
{
    "authResults": [
        {
            "allowed": {
                "policies": [
                    {
                        "policyArn": "string",
                        "policyName": "string"
                    }
                ]
            },
            "authDecision": "string",
            "authInfo": {
                "actionType": "string",
                "resources": [ "string" ]
            }
        }
    ]
}
```

```
        },
        "denied": {
            "explicitDeny": {
                "policies": [
                    {
                        "policyArn": "string",
                        "policyName": "string"
                    }
                ]
            },
            "implicitDeny": {
                "policies": [
                    {
                        "policyArn": "string",
                        "policyName": "string"
                    }
                ]
            }
        },
        "missingContextValues": [ "string" ]
    }
}
```

Dynamic thing groups

Dynamic thing groups update group membership through search queries. Using dynamic thing groups, you can change the way you interact with things depending on their connectivity, registry, or shadow data.

Because dynamic thing groups are tied to your fleet index, you must enable the fleet indexing service to use them. You can preview the things in a dynamic thing group before you create the group with a fleet indexing search query. For more information, see [Fleet indexing service \(p. 732\)](#) and [Query syntax \(p. 749\)](#).

You can specify a dynamic thing group as a target for a job. Only things that meet the criteria that define the dynamic thing group perform the job.

For example, suppose that you want to update the firmware on your devices, but, to minimize the chance that the update is interrupted, you only want to update firmware on devices with battery life greater than 80%. You can create a dynamic thing group that only includes devices with a reported battery life above 80%, and you can use that dynamic thing group as the target for your firmware update job. Only devices that meet your battery life criteria receive the firmware update. As devices reach the 80% battery life criteria, they are added to the dynamic thing group and receive the firmware update.

For more information about specifying thing groups as job targets, see [CreateJob](#).

Dynamic thing groups differ from static thing groups in the following ways:

- Thing membership is not explicitly defined. To create a dynamic thing group, you must define a query string that defines group membership.
- Dynamic thing groups cannot be part of a hierarchy.
- Dynamic thing groups cannot have policies applied to them.
- You use a different set of commands to create, update, and delete dynamic thing groups. For all other operations, the same commands that you use to interact with static thing groups can be used to interact with dynamic thing groups.
- The number of dynamic groups that a single account can have is [limited](#).

- You should not use personally identifiable information in your thing group name. The thing group name can appear in unencrypted communications and reports.
- You should not use a colon character (:) in a thing group name. The colon character is used as a delimiter by other AWS IoT services and this can cause them to parse strings with thing group names incorrectly.

For more information about static thing groups, see [Static thing groups \(p. 258\)](#).

As an example, suppose we create a dynamic group that contains all rooms in a warehouse whose temperature is greater than 60 degrees Fahrenheit. When a room's temperature is 61 degrees or higher, it is added to the RoomTooWarm dynamic thing group. All rooms in the RoomTooWarm dynamic thing group have cooling fans turned on. When a room's temperature falls to 60 degrees or lower, it is removed from the dynamic thing group and its fan would be turned off.

Create a dynamic thing group

Use the **CreateDynamicThingGroup** command to create a dynamic thing group. To create a dynamic thing group for the room too warm scenario you would use the **create-dynamic-thing-group** CLI command:

```
$ aws iot create-dynamic-thing-group --thing-group-name "RoomTooWarm" --query-string "attributes.temperature>60"
```

Note

We do not recommend using personally identifiable information in your dynamic thing group names.

The **CreateDynamicThingGroup** command returns a response that contains the index name, query string, query version, thing group name, thing group ID, and thing group ARN:

```
{  
    "indexName": "AWS_Things",  
    "queryVersion": "2017-09-30",  
    "thingGroupName": "RoomTooWarm",  
    "thingGroupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/RoomTooWarm",  
    "queryString": "attributes.temperature>60\n",  
    "thingGroupId": "abcdefghijklmnopqrstuvwxyz12345678ijklmnop12345678qrstuvwxyz"  
}
```

Dynamic thing group creation is not instantaneous. The dynamic thing group backfill takes time to complete. When a dynamic thing group is created, the status of the group is set to **BUILDING**. When the backfill is complete, the status changes to **ACTIVE**. To check the status of your dynamic thing group, use the [DescribeThingGroup](#) command.

Describe a dynamic thing group

Use the **DescribeThingGroup** command to get information about a dynamic thing group:

```
$ aws iot describe-thing-group --thing-group-name "RoomTooWarm"
```

The **DescribeThingGroup** command returns information about the specified group:

```
{  
    "status": "ACTIVE",
```

```
"indexName": "AWS_Things",
"thingGroupName": "RoomTooWarm",
"thingGroupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/RoomTooWarm",
"queryString": "attributes.temperature>60\n",
"version": 1,
"thingGroupMetadata": {
    "creationDate": 1548716921.289
},
"thingGroupProperties": {},
"queryVersion": "2017-09-30",
"thingGroupId": "84dd9b5b-2b98-4c65-84e4-be0e1ecf4fd8"
}
```

Running **DescribeThingGroup** on a dynamic thing group returns attributes that are specific to dynamic thing groups, such as the queryString and the status.

The status of a dynamic thing group can take the following values:

ACTIVE

The dynamic thing group is ready for use.

BUILDING

The dynamic thing group is being created, and thing membership is being processed.

REBUILDING

The dynamic thing group's membership is being updated, following the adjustment of the group's search query.

Note

After you create a dynamic thing group, you can use the group, regardless of its status. Only dynamic thing groups with an ACTIVE status include all of the things that match the search query for that dynamic thing group. Dynamic thing groups with BUILDING and REBUILDING statuses might not include all of the things that match the search query.

Update a dynamic thing group

Use the **UpdateDynamicThingGroup** command to update the attributes of a dynamic thing group, including the group's search query. The following command updates the thing group description and the query string changing the membership criteria to temperature > 65:

```
$ aws iot update-dynamic-thing-group --thing-group-name "RoomTooWarm" --thing-group-properties "thingGroupDescription=\"This thing group contains rooms warmer than 65F.\" --query-string "attributes.temperature>65"
```

The **UpdateDynamicThingGroup** command returns a response that contains the group's version number after the update:

```
{
    "version": 2
}
```

Dynamic thing group updates are not instantaneous. The dynamic thing group backfill takes time to complete. When a dynamic thing group is updated, the status of the group changes to REBUILDING while the group updates its membership. When the backfill is complete, the status changes to ACTIVE. To check the status of your dynamic thing group, use the **DescribeThingGroup** command.

Delete a dynamic thing group

Use the **DeleteDynamicThingGroup** command to delete a dynamic thing group:

```
$ aws iot delete-dynamic-thing-group --thing-group-name "RoomTooWarm"
```

The **DeleteDynamicThingGroup** command does not produce any output.

Commands that show which groups a thing belongs to (for example, **ListGroupsForThing**) might continue to show the group while records in the cloud are being updated.

Limitations and conflicts

Dynamic thing groups share these limitations with static thing groups:

- The number of attributes a thing group can have is [limited](#).
- The number of groups to which a thing can belong is [limited](#).
- Thing groups cannot be renamed.
- Thing group names cannot contain international characters, such as û, é, and ñ.

When using dynamic thing groups, keep the following in mind.

The fleet indexing service must be enabled

The fleet indexing service must be enabled and the fleet indexing backfill must be complete before you can create and use dynamic thing groups. Expect a delay after you enable the fleet indexing service. The backfill can take some time to complete. The more things that you have registered, the longer the backfill process takes. After you enable the fleet indexing service for dynamic thing groups, you cannot disable it until you delete all of your dynamic thing groups.

Note

If you have permissions to query the fleet index, you can access the data of things across the entire fleet.

The number of dynamic thing groups is limited

The number of dynamic groups is [limited](#).

Successful commands can log errors

When creating or updating a dynamic thing group, it's possible that some things might be eligible to be in a dynamic thing group yet not be added to it. The command to create or update a dynamic thing group, however, still succeeds in those cases while logging an error and generating an [AddThingToDynamicThingGroupsFailed metric \(p. 411\)](#).

An [error log entry](#) in the CloudWatch log is created for each thing when an eligible thing cannot be added to a dynamic thing group or a thing is removed from a dynamic thing group to add it to another group. When a thing cannot be added to a dynamic group, an [AddThingToDynamicThingGroupsFailed metric \(p. 411\)](#) is also created; however, a single metric can represent multiple log entries.

When a thing becomes eligible to be added to a dynamic thing group, the following is considered:

- Is the thing already in as many groups as it can be? (See [limits](#))

- **NO:** The thing is added to the dynamic thing group.
- **YES:** Is the thing a member of any dynamic thing groups?
 - **NO:** The thing can't be added to the dynamic thing group, an error is logged, and an [AddThingToDynamicThingGroupsFailed metric \(p. 411\)](#) is generated.
 - **YES:** Is the dynamic thing group to join older than any dynamic thing group that the thing is already a member of?
 - **NO:** The thing can't be added to the dynamic thing group, an error is logged, and an [AddThingToDynamicThingGroupsFailed metric \(p. 411\)](#) is generated.
 - **YES:** Remove the thing from the most recent dynamic thing group it is a member of, log an error, and add the thing to the dynamic thing group. This generates an error and an [AddThingToDynamicThingGroupsFailed metric \(p. 411\)](#) for the dynamic thing group from which the thing was removed.

When a thing in a dynamic thing group no longer meets the search query, it is removed from the dynamic thing group. Likewise, when a thing is updated to meet a dynamic thing group's search query, it is then added to the group as previously described. These additions and removals are normal and do not produce error log entries.

With `overrideDynamicGroups` enabled, static groups take priority over dynamic groups

The number of groups to which a thing can belong is [limited](#). When you update thing membership by using the [AddThingToThingGroup](#) or [UpdateThingGroupsForThing](#) commands, adding the `--overrideDynamicGroups` parameter gives static thing groups priority over dynamic thing groups.

When adding a thing to a static thing group, the following is considered:

- Does the thing already belong to the maximum number of groups?
 - **NO:** The thing is added to the static thing group.
 - **YES:** Is the thing in any dynamic groups?
 - **NO:** The thing cannot be added to the thing group. The command raises an exception.
 - **YES:** Was `--overrideDynamicGroups` enabled?
 - **NO:** The thing cannot be added to the thing group. The command raises an exception.
 - **YES:** The thing is removed from the most recently created dynamic thing group, an error is logged, and an [AddThingToDynamicThingGroupsFailed metric \(p. 411\)](#) is generated for the dynamic thing group from which the thing was removed. Then, the thing is added to the static thing group.

Older dynamic thing groups take priority over newer ones

The number of groups to which a thing can belong is [limited](#). When a thing becomes eligible to be added to a dynamic thing group because of a create or update operation, and the thing is already in as many groups as it can be, it can be removed from another dynamic thing group to enable this addition. For more information about how this occurs, see [Successful commands can log errors \(p. 270\)](#) and [With `overrideDynamicGroups` enabled, static groups take priority over dynamic groups \(p. 271\)](#) for examples.

When a thing is removed from a dynamic thing group, an error is logged, and an event is raised.

You cannot apply policies to dynamic thing groups

Attempting to apply a policy to a dynamic thing group generates an exception.

Dynamic thing group membership is eventually consistent

Only the final state of a thing is evaluated for the registry. Intermediary states can be skipped if states are updated rapidly. Avoid associating a rule or job, with a dynamic thing group whose membership depends on an intermediary state.

Tagging your AWS IoT resources

To help you manage and organize your thing groups, thing types, topic rules, jobs, scheduled audits and security profiles you can optionally assign your own metadata to each of these resources in the form of tags. This section describes tags and shows you how to create them.

To help you manage your costs related to things, you can create [billing groups \(p. 276\)](#) that contain things. You can then assign tags that contain your metadata to each of these billing groups. This section also discusses billing groups and the commands available to create and manage them.

Tag basics

You can use tags to categorize your AWS IoT resources in different ways (for example, by purpose, owner, or environment). This is useful when you have many resources of the same type — you can quickly identify a resource based on the tags you've assigned to it. Each tag consists of a key and optional value, both of which you define. For example, you can define a set of tags for your thing types that helps you track devices by type. We recommend that you create a set of tag keys that meets your needs for each kind of resource. Using a consistent set of tag keys makes it easier for you to manage your resources.

You can search for and filter resources based on the tags you add or apply. You can also use billing group tags to categorize and track your costs. You can also use tags to control access to your resources as described in [Using tags with IAM policies \(p. 274\)](#).

For ease of use, the Tag Editor in the AWS Management Console provides a central, unified way to create and manage your tags. For more information, see [Working with Tag Editor](#) in [Working with the AWS Management Console](#).

You can also work with tags using the AWS CLI and the AWS IoT API. You can associate tags with thing groups, thing types, topic rules, jobs, security profiles, policies, and billing groups when you create them by using the `Tags` field in the following commands:

- [CreateBillingGroup](#)
- [CreateDestination](#)
- [CreateDeviceProfile](#)
- [CreateDynamicThingGroup](#)
- [CreateJob](#)
- [CreateOTAUpdate](#)
- [CreatePolicy](#)
- [CreateScheduledAudit](#)
- [CreateSecurityProfile](#)
- [CreateServiceProfile](#)
- [CreateStream](#)
- [CreateThingGroup](#)
- [CreateThingType](#)
- [CreateTopicRule](#)

- [CreateWirelessGateway](#)
- [CreateWirelessDevice](#)

You can add, modify, or delete tags for existing resources that support tagging by using the following commands:

- [TagResource](#)
- [ListTagsForResource](#)
- [UntagResource](#)

You can edit tag keys and values, and you can remove tags from a resource at any time. You can set the value of a tag to an empty string, but you can't set the value of a tag to null. If you add a tag that has the same key as an existing tag on that resource, the new value overwrites the old value. If you delete a resource, any tags associated with the resource are also deleted.

Tag restrictions and limitations

The following basic restrictions apply to tags:

- Maximum number of tags per resource — 50
- Maximum key length — 127 Unicode characters in UTF-8
- Maximum value length — 255 Unicode characters in UTF-8
- Tag keys and values are case sensitive.
- Do not use the `aws :` prefix in your tag names or values. It's reserved for AWS use. You can't edit or delete tag names or values with this prefix. Tags with this prefix don't count against your tags per resource limit.
- If your tagging schema is used across multiple services and resources, remember that other services might have restrictions on allowed characters. Allowed characters include letters, spaces, and numbers representable in UTF-8, and the following special characters: `+ - = . _ : / @`.

Using tags with IAM policies

You can apply tag-based resource-level permissions in the IAM policies you use for AWS IoT API actions. This gives you better control over what resources a user can create, modify, or use. You use the Condition element (also called the Condition block) with the following condition context keys and values in an IAM policy to control user access (permissions) based on a resource's tags:

- Use `aws:ResourceTag/tag-key: tag-value` to allow or deny user actions on resources with specific tags.
- Use `aws:RequestTag/tag-key: tag-value` to require that a specific tag be used (or not used) when making an API request to create or modify a resource that allows tags.
- Use `aws:TagKeys: [tag-key, ...]` to require that a specific set of tag keys be used (or not used) when making an API request to create or modify a resource that allows tags.

Note

The condition context keys and values in an IAM policy apply only to those AWS IoT actions where an identifier for a resource capable of being tagged is a required parameter. For example, the use of [DescribeEndpoint](#) is not allowed or denied on the basis of condition context keys and values because no taggable resource (thing groups, thing types, topic rules, jobs, or security profile) is referenced in this request. For more information about AWS IoT resources that are

taggable and condition keys they support, read [Actions, resources, and condition keys for AWS IoT](#).

For more information about using tags, see [Controlling Access Using Tags](#) in the *AWS Identity and Access Management User Guide*. The [IAM JSON Policy Reference](#) section of that guide has detailed syntax, descriptions, and examples of the elements, variables, and evaluation logic of JSON policies in IAM.

The following example policy applies two tag-based restrictions for the ThingGroup actions. An IAM user restricted by this policy:

- Can't create a thing group the tag "env=prod" (in the example, see the line "aws:RequestTag/env" : "prod").
- Can't modify or access a thing group that has an existing tag "env=prod" (in the example, see the line "aws:ResourceTag/env" : "prod").

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Deny",  
            "Action": "iot:CreateThingGroup",  
            "Resource": "*",  
            "Condition": {  
                "StringEquals": {  
                    "aws:RequestTag/env": "prod"  
                }  
            }  
        },  
        {  
            "Effect": "Deny",  
            "Action": [  
                "iot:CreateThingGroup",  
                "iot>DeleteThingGroup",  
                "iot:DescribeThingGroup",  
                "iot:UpdateThingGroup"  
            ],  
            "Resource": "*",  
            "Condition": {  
                "StringEquals": {  
                    "aws:ResourceTag/env": "prod"  
                }  
            }  
        },  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:CreateThingGroup",  
                "iot>DeleteThingGroup",  
                "iot:DescribeThingGroup",  
                "iot:UpdateThingGroup"  
            ],  
            "Resource": "*"  
        }  
    ]  
}
```

You can also specify multiple tag values for a given tag key by enclosing them in a list, like this:

```
"StringEquals" : {  
    "aws:ResourceTag/env" : ["dev", "test"]
```

}

Note

If you allow or deny users access to resources based on tags, you must consider explicitly denying users the ability to add those tags to or remove them from the same resources. Otherwise, it's possible for a user to circumvent your restrictions and gain access to a resource by modifying its tags.

Billing groups

AWS IoT doesn't allow you to directly apply tags to individual things, but it does allow you to place things in billing groups and to apply tags to these. For AWS IoT, allocation of cost and usage data based on tags is limited to billing groups.

AWS IoT Core for LoRaWAN resources, such as wireless devices and gateways, can't be added to billing groups. However, they can be associated with AWS IoT things, which can be added to billing groups.

The following commands are available:

- [AddThingToBillingGroup](#) adds a thing to a billing group.
- [CreateBillingGroup](#) creates a billing group.
- [DeleteBillingGroup](#) deletes the billing group.
- [DescribeBillingGroup](#) returns information about a billing group.
- [ListBillingGroups](#) lists the billing groups you have created.
- [ListThingsInBillingGroup](#) lists the things you have added to the given billing group.
- [RemoveThingFromBillingGroup](#) removes the given thing from the billing group.
- [UpdateBillingGroup](#) updates information about the billing group.
- [CreateThing](#) allows you to specify a billing group for the thing when you create it.
- [DescribeThing](#) returns the description of a thing including the billing group the thing belongs to, if any.

The AWS IoT Wireless API provides these actions to associate wireless devices and gateways with AWS IoT things.

- [AssociateWirelessDeviceWithThing](#)
- [AssociateWirelessGatewayWithThing](#)

Viewing cost allocation and usage data

You can use billing group tags to categorize and track your costs. When you apply tags to billing groups (and so to the things they include), AWS generates a cost allocation report as a comma-separated value (CSV) file with your usage and costs aggregated by your tags. You can apply tags that represent business categories (such as cost centers, application names, or owners) to organize your costs across multiple services. For more information about using tags for cost allocation, see [Use Cost Allocation Tags](#) in the [AWS Billing and Cost Management User Guide](#).

Note

To accurately associate usage and cost data with those things you have placed in billing groups, each device or application must:

- Be registered as a thing in AWS IoT. For more information, see [Managing devices with AWS IoT \(p. 251\)](#).

- Connect to the AWS IoT message broker through MQTT using only the thing's name as the client ID. For more information, see [the section called "Device communication protocols" \(p. 77\)](#).
- Authenticate using a client certificate associated with the thing.

The following pricing dimensions are available for billing groups (based on the activity of things associated with the billing group):

- Connectivity (based on the thing name used as the client ID to connect).
- Messaging (based on messages inbound from, and outbound to, a thing; MQTT only).
- Shadow operations (based on the thing whose message triggered a shadow update).
- Rules triggered (based on the thing whose inbound message triggered the rule; does not apply to those rules triggered by MQTT lifecycle events).
- Thing index updates (based on the thing that was added to the index).
- Remote actions (based on the thing updated).
- [Detect \(p. 879\)](#) reports (based on the thing whose activity is reported).

Cost and usage data based on tags (and reported for a billing group) doesn't reflect the following activities:

- Device registry operations (including updates to things, thing groups, and thing types). For more information, see [Managing devices with AWS IoT \(p. 251\)](#).
- Thing group index updates (when adding a thing group).
- Index search queries.
- [Device provisioning \(p. 704\)](#).
- [Audit \(p. 818\)](#) reports.

Security in AWS IoT

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The [shared responsibility model](#) describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the [AWS compliance programs](#). To learn about the compliance programs that apply to AWS IoT, see [AWS Services in Scope by Compliance Program](#).
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

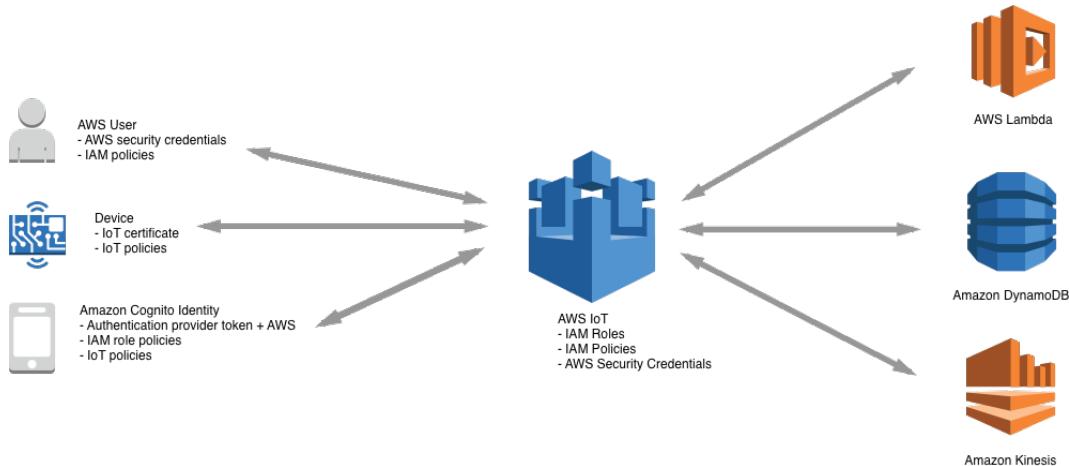
This documentation helps you understand how to apply the shared responsibility model when using AWS IoT. The following topics show you how to configure AWS IoT to meet your security and compliance objectives. You also learn how to use other AWS services that help you to monitor and secure your AWS IoT resources.

Topics

- [AWS IoT security \(p. 278\)](#)
- [Authentication \(p. 279\)](#)
- [Authorization \(p. 312\)](#)
- [Data protection in AWS IoT Core \(p. 361\)](#)
- [Identity and access management for AWS IoT \(p. 364\)](#)
- [Logging and Monitoring \(p. 390\)](#)
- [Compliance validation for AWS IoT Core \(p. 391\)](#)
- [Resilience in AWS IoT Core \(p. 391\)](#)
- [Using AWS IoT Core with interface VPC endpoints \(p. 392\)](#)
- [Infrastructure security in AWS IoT \(p. 394\)](#)
- [Security monitoring of production fleets or devices with AWS IoT Core \(p. 395\)](#)
- [Security best practices in AWS IoT Core \(p. 395\)](#)
- [AWS training and certification \(p. 399\)](#)

AWS IoT security

Each connected device or client must have a credential to interact with AWS IoT. All traffic to and from AWS IoT is sent securely over Transport Layer Security (TLS). AWS cloud security mechanisms protect data as it moves between AWS IoT and other AWS services.



- You are responsible for managing device credentials (X.509 certificates, AWS credentials, Amazon Cognito identities, federated identities, or custom authentication tokens) and policies in AWS IoT. For more information, see [Key management in AWS IoT \(p. 363\)](#). You are responsible for assigning unique identities to each device and managing the permissions for each device or group of devices.
- Your devices connect to AWS IoT using X.509 certificates or Amazon Cognito identities over a secure TLS connection. During research and development, and for some applications that make API calls or use WebSockets, you can also authenticate using IAM users and groups or custom authentication tokens. For more information, see [IAM users, groups, and roles \(p. 300\)](#).
- When using AWS IoT authentication, the message broker is responsible for authenticating your devices, securely ingesting device data, and granting or denying access permissions you specify for your devices using AWS IoT policies.
- When using custom authentication, a custom authorizer is responsible for authenticating your devices and granting or denying access permissions you specify for your devices using AWS IoT or IAM policies.
- The AWS IoT rules engine forwards device data to other devices or other AWS services according to rules you define. It uses AWS Identity and Access Management to securely transfer data to its final destination. For more information, see [Identity and access management for AWS IoT \(p. 364\)](#).

Authentication

Authentication is a mechanism where you verify the identity of a client or a server. Server authentication is the process where devices or other clients ensure they are communicating with an actual AWS IoT endpoint. Client authentication is the process where devices or other clients authenticate themselves with AWS IoT.

AWS training and certification

Take the following course to learn about authentication in AWS IoT: [Deep Dive into AWS IoT Authentication and Authorization](#).

X.509 Certificate overview

X.509 certificates are digital certificates that use the [X.509 public key infrastructure standard](#) to associate a public key with an identity contained in a certificate. X.509 certificates are issued by a trusted entity called a certification authority (CA). The CA maintains one or more special certificates called CA certificates that it uses to issue X.509 certificates. Only the certification authority has access to

CA certificates. X.509 certificate chains are used both for server authentication by clients and client authentication by the server.

Server authentication

When your device or other client attempts to connect to AWS IoT Core, the AWS IoT Core server will send an X.509 certificate that your device uses to authenticate the server. Authentication takes place at the TLS layer through validation of the [X.509 certificate chain \(p. 282\)](#). This is the same method used by your browser when you visit an HTTPS URL. If you want to use certificates from your own certificate authority, see [Manage your CA certificates \(p. 286\)](#).

When your devices or other clients establish a TLS connection to an AWS IoT Core endpoint, AWS IoT Core presents a certificate chain that the devices use to verify that they're communicating with AWS IoT Core and not another server impersonating AWS IoT Core. The chain that is presented depends on a combination of the type of endpoint the device is connecting to and the [cipher suite \(p. 362\)](#) that the client and AWS IoT Core negotiated during the TLS handshake.

Endpoint types

AWS IoT Core supports two different data endpoint types, `iot:Data` and `iot:Data-ATS`. `iot:Data` endpoints present a certificate signed by the [VeriSign Class 3 Public Primary G5 root CA certificate](#). `iot:Data-ATS` endpoints present a server certificate signed by an [Amazon Trust Services](#) CA.

Certificates presented by ATS endpoints are cross signed by Starfield. Some TLS client implementations require validation of the root of trust and require that the Starfield CA certificates are installed in the client's trust stores.

Warning

Using a method of certificate pinning that hashes the whole certificate (including the issuer name, and so on) is not recommended because this will cause certificate verification to fail because the ATS certificates we provide are cross signed by Starfield and have a different issuer name.

Use `iot:Data-ATS` endpoints unless your device requires Symantec or Verisign CA certificates. Symantec and Verisign certificates have been deprecated and are no longer supported by most web browsers.

You can use the `describe-endpoint` command to create your ATS endpoint.

```
aws iot describe-endpoint --endpoint-type iot:Data-ATS
```

The `describe-endpoint` command returns an endpoint in the following format.

```
account-specific-prefix.iot.your-region.amazonaws.com
```

The first time `describe-endpoint` is called, an endpoint is created. All subsequent calls to `describe-endpoint` return the same endpoint.

For backward-compatibility, AWS IoT Core still supports Symantec endpoints. For more information, see [How AWS IoT Core is Helping Customers Navigate the Upcoming Distrust of Symantec Certificate Authorities](#). Devices operating on ATS endpoints are fully interoperable with devices operating on Symantec endpoints in the same account and do not require any re-registration.

Note

To see your `iot:Data-ATS` endpoint in the AWS IoT Core console, choose **Settings**. The console displays only the `iot:Data-ATS` endpoint. By default, the `describe-endpoint` command displays the `iot:Data` endpoint for backward compatibility. To see the `iot:Data-ATS` endpoint, specify the `--endpointType` parameter, as in the previous example.

Creating an `IotDataPlaneClient` with the AWS SDK for Java

By default, the [AWS SDK for Java - Version 2](#) creates an `IotDataPlaneClient` by using an `iot:Data` endpoint. To create a client that uses an `iot:Data-ATS` endpoint, you must do the following.

- Create an `iot:Data-ATS` endpoint by using the [DescribeEndpoint](#) API.
- Specify that endpoint when you create the `IotDataPlaneClient`.

The following example performs both of these operations.

```
public void setup() throws Exception {
    IotClient client =
        IotClient.builder().credentialsProvider(CREDENTIALS_PROVIDER_CHAIN).region(Region.US_EAST_1).build();
    String endpoint = client.describeEndpoint(r -> r.endpointType("iot:Data-
ATS")).endpointAddress();
    Iot iot = IotDataPlaneClient.builder()
        .credentialsProvider(CREDENTIALS_PROVIDER_CHAIN)
        .endpointOverride(URI.create("https://" + endpoint))
        .region(Region.US_EAST_1)
        .build();
}
```

CA certificates for server authentication

Depending on which type of data endpoint you are using and which cipher suite you have negotiated, AWS IoT Core server authentication certificates are signed by one of the following root CA certificates:

VeriSign Endpoints (legacy)

- RSA 2048 bit key: [VeriSign Class 3 Public Primary G5 root CA certificate](#)

Amazon Trust Services Endpoints (preferred)

Note

You might need to right click these links and select **Save link as...** to save these certificates as files.

- RSA 2048 bit key: [Amazon Root CA 1](#).
- RSA 4096 bit key: Amazon Root CA 2. Reserved for future use.
- ECC 256 bit key: [Amazon Root CA 3](#).
- ECC 384 bit key: Amazon Root CA 4. Reserved for future use.

These certificates are all cross-signed by the [Starfield Root CA Certificate](#). All new AWS IoT Core regions, beginning with the May 9, 2018 launch of AWS IoT Core in the Asia Pacific (Mumbai) Region, serve only ATS certificates.

Server authentication guidelines

There are many variables that can affect a device's ability to validate the AWS IoT Core server authentication certificate. For example, devices may be too memory constrained to hold all possible root CA certificates, or devices may implement a non-standard method of certificate validation. For these reasons we suggest following these guidelines:

- We recommend that you use your ATS endpoint and install all supported Amazon Root CA certificates.
- If you cannot store all of these certificates on your device and if your devices do not use ECC-based validation, you can omit the [Amazon Root CA 3](#) and [Amazon Root CA 4](#) ECC certificates. If your devices

do not implement RSA-based certificate validation, you can omit the [Amazon Root CA 1](#) and [Amazon Root CA 2](#) RSA certificates. You might need to right click these links and select **Save link as...** to save these certificates as files.

- If you are experiencing server certificate validation issues when connecting to your ATS endpoint, try adding the relevant cross-signed Amazon Root CA certificate to your trust store. You might need to right click these links and select **Save link as...** to save these certificates as files.
 - [Cross-signed Amazon Root CA 1](#)
 - [Cross-signed Amazon Root CA 2 - Reserved for future use.](#)
 - [Cross-signed Amazon Root CA 3](#)
 - [Cross-signed Amazon Root CA 4 - Reserved for future use.](#)
- If you are experiencing server certificate validation issues, your device may need to explicitly trust the root CA. Try adding the [Starfield Root CA Certificate](#) to your trust store.
- If you still experience issues after executing the steps above, please contact [AWS Developer Support](#).

Note

CA certificates have an expiration date after which they cannot be used to validate a server's certificate. CA certificates might have to be replaced before their expiration date. Make sure that you can update the root CA certificates on all of your devices or clients to help ensure ongoing connectivity and to keep up to date with security best practices.

Note

When connecting to AWS IoT Core in your device code, pass the certificate into the API you are using to connect. The API you use will vary by SDK. For more information, see the [AWS IoT Core Device SDKs \(p. 1127\)](#).

Client authentication

AWS IoT supports three types of identity principals for device or client authentication:

- [X.509 client certificates \(p. 282\)](#)
- [IAM users, groups, and roles \(p. 300\)](#)
- [Amazon Cognito identities \(p. 300\)](#)

These identities can be used with devices, mobile, web, or desktop applications. They can even be used by a user typing AWS IoT command line interface (CLI) commands. Typically, AWS IoT devices use X.509 certificates, while mobile applications use Amazon Cognito identities. Web and desktop applications use IAM or federated identities. AWS CLI commands use IAM. For more information about IAM identities, see [Identity and access management for AWS IoT \(p. 364\)](#).

X.509 client certificates

X.509 certificates provide AWS IoT with the ability to authenticate client and device connections. Client certificates must be registered with AWS IoT before a client can communicate with AWS IoT. A client certificate can be registered in multiple AWS accounts in the same AWS Region to facilitate moving devices between your AWS accounts in the same region. See [Using X.509 client certificates in multiple AWS accounts with multi-account registration \(p. 283\)](#) for more information.

We recommend that each device or client be given a unique certificate to enable fine-grained client management actions, including certificate revocation. Devices and clients must also support rotation and replacement of certificates to help ensure smooth operation as certificates expire.

For information about using X.509 certificates to support more than a few devices, see [Device provisioning \(p. 704\)](#) to review the different certificate management and provisioning options that AWS IoT supports.

AWS IoT supports these types of X.509 client certificates:

- X.509 certificates generated by AWS IoT
- X.509 certificates signed by a CA registered with AWS IoT.
- X.509 certificates signed by a CA that is not registered with AWS IoT.

This section describes how to manage X.509 certificates in AWS IoT. You can use the AWS IoT console or AWS CLI to perform these certificate operations:

- [Create AWS IoT client certificates \(p. 284\)](#)
- [Create your own client certificates \(p. 285\)](#)
- [Register a client certificate \(p. 290\)](#)
- [Activate or deactivate a client certificate \(p. 294\)](#)
- [Revoke a client certificate \(p. 296\)](#)

For more information about the AWS CLI commands that perform these operations, see [AWS IoT CLI Reference](#).

Using X.509 client certificates

X.509 certificates authenticate client and device connections to AWS IoT. X.509 certificates provide several benefits over other identification and authentication mechanisms. X.509 certificates enable asymmetric keys to be used with devices. For example, you could burn private keys into secure storage on a device so that sensitive cryptographic material never leaves the device. X.509 certificates provide stronger client authentication over other schemes, such as user name and password or bearer tokens, because the private key never leaves the device.

AWS IoT authenticates client certificates using the TLS protocol's client authentication mode. TLS support is available in many programming languages and operating systems and is commonly used for encrypting data. In TLS client authentication, AWS IoT requests an X.509 client certificate and validates the certificate's status and AWS account against a registry of certificates. It then challenges the client for proof of ownership of the private key that corresponds to the public key contained in the certificate. AWS IoT requires clients to send the [Server Name Indication \(SNI\) extension](#) to the Transport Layer Security (TLS) protocol. For more information on configuring the SNI extension, see [Transport security in AWS IoT \(p. 362\)](#).

X.509 certificates can be verified against a trusted certificate authority (CA). You can create client certificates that use the Amazon Root CA and you can use your own client certificates signed by another CA. For more information about using your own X.509 certificates, see [Create your own client certificates \(p. 285\)](#).

The date and time when certificates signed by a CA certificate expire are set when the certificate is created. X.509 certificates generated by AWS IoT expire at midnight UTC on December 31, 2049 (2049-12-31T23:59:59Z). For more information about using the AWS IoT console to create certificates that use the Amazon Root CA, see [Create AWS IoT client certificates \(p. 284\)](#).

Using X.509 client certificates in multiple AWS accounts with multi-account registration

Multi-account registration makes it possible to move devices between your AWS accounts in the same Region or in different Regions. With this, you can register, test, and configure a device in a pre-production account, and then register and use the same device and device certificate in a production account. You can also [register the client certificate on the device \(the device certificates\) without a CA \(p. 292\)](#) that is registered with AWS IoT.

Note

Certificates used for multi-account registration are supported on the `iot:Data-ATS`, `iot:Data (legacy)`, `iot:Jobs`, and `iot:CredentialProvider` endpoint types. For more information about AWS IoT device endpoints, see [AWS IoT device data and service endpoints \(p. 74\)](#).

Devices that use multi-account registration must send the [Server Name Indication \(SNI\) extension](#) to the Transport Layer Security (TLS) protocol and provide the complete endpoint address in the `host_name` field, when they connect to AWS IoT. AWS IoT uses the endpoint address in `host_name` to route the connection to the correct AWS IoT account. Existing devices that don't send a valid endpoint address in `host_name` will continue to work, but they will not be able to use the features that require this information. For more information about the SNI extension and to learn how to identify the endpoint address for the `host_name` field, see [Transport security in AWS IoT \(p. 362\)](#).

To use multi-account registration

1. Do not register the CA that signed the device certificates with AWS IoT.
2. Register the device certificates without a CA. See [Register a client certificate signed by an unregistered CA \(CLI\) \(p. 292\)](#).
3. Use the correct `host_name` in the SNI extension to TLS when the device connects to AWS IoT. See [Transport security in AWS IoT \(p. 362\)](#).

Certificate signing algorithms supported by AWS IoT

AWS IoT supports the following certificate-signing algorithms:

- SHA256WITHRSA
- SHA384WITHRSA
- SHA512WITHRSA
- DSA_WITH_SHA256
- ECDSA-WITH-SHA256
- ECDSA-WITH-SHA384
- ECDSA-WITH-SHA512

Create AWS IoT client certificates

AWS IoT provides client certificates that are signed by the Amazon Root certificate authority (CA).

This topic describes how to create a client certificate signed by the Amazon Root certificate authority and download the certificate files. After you create the client certificate files, you must install them on the client.

Note

Each X.509 client certificate provided by AWS IoT holds issuer and subject attributes that are set at the time of certificate creation. The certificate attributes are immutable only after the certificate is created.

You can use the AWS IoT console or the AWS CLI to create an AWS IoT certificate signed by the Amazon Root certificate authority.

[Create an AWS IoT certificate \(console\)](#)

To create an AWS IoT certificate using the AWS IoT console

1. Sign in to the AWS Management Console, and open the [AWS IoT console](#).

2. In the left navigation pane, choose **Secure**, choose **Certificates**, and then choose **Create**.
3. Choose **One-click certificate creation (recommended) - Create certificate**.
4. From the **Certificate created!** page, download the client certificate files for the thing, public key, and private key to a secure location.

If you also need the Amazon Root CA certificate file, this page also has the link to the page from where you can download it.

5. A client certificate has now been created and registered with AWS IoT. You must activate the certificate before you use it in a client.

Choose **Activate** to activate the client certificate now. If you don't want to activate the certificate now, [Activate a client certificate \(console\) \(p. 294\)](#) describes how to activate the certificate later.

6. If you want to attach a policy to the certificate, choose **Attach a policy**.

If you don't want to attach a policy now, choose **Done** to finish. You can attach a policy later.

After you complete the procedure, install the certificate files on the client.

Create an AWS IoT certificate (CLI)

The AWS CLI provides the [create-keys-and-certificate](#) command to create client certificates signed by the Amazon Root certificate authority. This command, however, does not download the Amazon Root CA certificate file. You can download the Amazon Root CA certificate file from [CA certificates for server authentication \(p. 281\)](#).

This command creates private key, public key, and X.509 certificate files and registers and activates the certificate with AWS IoT.

```
aws iot create-keys-and-certificate \
--set-as-active \
--certificate-pemoutfile certificate_filename \
--public-key-outfile public_key_filename \
--private-key-outfile private_key_filename
```

If you don't want to activate the certificate when you create and register it, this command creates private key, public key, and X.509 certificate files and registers the certificate, but it does not activate it. [Activate a client certificate \(CLI\) \(p. 295\)](#) describes how to activate the certificate later.

```
aws iot create-keys-and-certificate \
--no-set-as-active \
--certificate-pemoutfile certificate_filename \
--public-key-outfile public_key_filename \
--private-key-outfile private_key_filename
```

Install the certificate files on the client.

Create your own client certificates

AWS IoT supports client certificates signed by other root certificate authorities (CA). You can register client certificates signed by another root CA; however, if you want the device or client to register its client certificate when it first connects to AWS IoT, the root CA must be registered with AWS IoT.

Note

A CA certificate can be registered by only one account in a Region.

For more information about using X.509 certificates to support more than a few devices, see [Device provisioning \(p. 704\)](#) to review the different certificate management and provisioning options that AWS IoT supports.

Topics

- [Manage your CA certificates \(p. 286\)](#)
- [Create a client certificate using your CA certificate \(p. 289\)](#)

Manage your CA certificates

This section describes common tasks for managing your own certificate authority (CA) certificates.

- [Create a CA certificate \(p. 286\)](#), if you need one.
- [Register your CA certificate \(p. 286\)](#)
- [Deactivate a CA certificate \(p. 288\)](#)

Create a CA certificate

If you do not have a CA certificate, you can use [OpenSSL v1.1.1i](#) tools to create one.

Note

You can't perform this procedure in the AWS IoT console.

To create a CA certificate using [OpenSSL v1.1.1i](#) tools

1. Generate a key pair.

```
openssl genrsa -out root_CA_key_filename 2048
```

2. Use the private key from the key pair to generate a CA certificate.

```
openssl req -x509 -new -nodes \
    -key root_CA_key_filename \
    -sha256 -days 1024 \
    -out root_CA_pem_filename
```

Register your CA certificate

You might need to register your certificate authority (CA) with AWS IoT if you are using client certificates signed by a CA that AWS IoT doesn't recognize.

If you want clients to automatically register their client certificates with AWS IoT when they first connect, the CA that signed the client certificates must be registered with AWS IoT. Otherwise, you don't need to register the CA certificate that signed the client certificates.

Note

A CA certificate can be registered by only one account in a Region.

Register a CA certificate (console)

Note

Make sure you have the root CA's certificate file and private key file before you begin. This procedure also requires using the command line interface to run [OpenSSL v1.1.1i](#) commands.

To register a CA certificate using the AWS IoT console

1. Sign in to the AWS Management Console and open the [AWS IoT console](#).
2. In the left navigation pane, choose **Secure**, choose **CAs**, and then **Register**.
3. In **Select a CA**, choose **Register CA**.
4. In **Register a CA certificate**, follow the steps displayed.

Steps 1 - 4 are performed in the command line interface.

Steps 5 and 6 require the files created in Steps 3 and 4.

5. If you want to activate this certificate when you register it, check **Activate CA certificate**.

The CA certificate must be active before you can register any client certificates that are signed by it.

6. If you want to enable this certificate to automatically register client certificates signed by this certificate, select **Enable auto-registration of device certificates**.
7. Choose **Register CA certificate** to complete the registration.

The CA certificate appears in the list of certificate authorities with its current status.

Register a CA certificate (CLI)

Note

Make sure you have the root CA's certificate file and private key file before you begin.

To register a CA certificate using the AWS CLI

1. Use [get-registration-code](#) to get a registration code from AWS IoT. Save the `registrationCode` returned to use as the `Common Name` of the private key verification certificate.

```
aws iot get-registration-code
```

2. Generate a key pair for the private key verification certificate:

```
openssl genrsa -out verification_cert_key_filename 2048
```

3. Create a certificate signing request (CSR) for the private key verification certificate. Set the `Common Name` field of the certificate to the `registrationCode` returned by [get-registration-code](#).

```
openssl req -new \
    -key verification_cert_key_filename \
    -out verification_cert_csr_filename
```

You are prompted for some information, including the `Common Name` for the certificate.

```
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:
State or Province Name (full name) []:
Locality Name (for example, city) []:
Organization Name (for example, company) []:
Organizational Unit Name (for example, section) []:
Common Name (e.g. server FQDN or YOUR name) []:your_registration_code
Email Address []:
```

```
Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

4. Use the CSR to create a private key verification certificate:

```
openssl x509 -req \
-in verification_cert_csr_filename \
-CA root_CA_pem_filename \
-CAkey root_CA_key_filename \
-CAcreateserial \
-out verification_cert_pem_filename \
-days 500 -sha256
```

5. Register the CA certificate with AWS IoT. Pass in the CA certificate filename and the private key verification certificate filename to the [register-ca-certificate](#) command:

```
aws iot register-ca-certificate \
--ca-certificate file://root_CA_pem_filename \
--verification-cert file://verification_cert_pem_filename
```

This command returns the *certificateId*, if successful.

6. At this point, the CA certificate has been registered with AWS IoT, but is not active. The CA certificate must be active before you can register any client certificates that are signed by it.

This step activates the CA certificate.

Use the [update-certificate](#) CLI command to activate the CA certificate:

```
aws iot update-ca-certificate \
--certificate-id certificateId \
--new-status ACTIVE
```

Use the [describe-ca-certificate](#) command to see the status of the CA certificate.

Deactivate a CA certificate

When a certificate authority (CA) certificate is enabled for automatic client certificate registration, AWS IoT checks the CA certificate used to sign the client certificate to make sure the CA is ACTIVE. If the CA certificate is INACTIVE, AWS IoT doesn't allow the client certificate to be registered.

By setting the CA certificate as INACTIVE, you prevent any new client certificates issued by the CA from being registered automatically.

Note

Any registered client certificates that were signed by the compromised CA certificate continue to work until you explicitly revoke each one of them.

Deactivate a CA certificate (console)

To deactivate a CA certificate using the AWS IoT console

1. Sign in to the AWS Management Console and open the [AWS IoT console](#).
2. In the left navigation pane, choose **Secure**, choose **CAs**.
3. In the list of certificate authorities, find the one that you want to deactivate, and open the option menu by using the ellipsis icon.

4. On the option menu, choose **Deactivate**.

The certificate authority should show as **Inactive** in the list.

Note

The AWS IoT console does not provide a way to list the certificates that were signed by the CA you deactivated. For an AWS CLI option to list those certificates, see [Deactivate a CA certificate \(CLI\) \(p. 289\)](#).

Deactivate a CA certificate (CLI)

The AWS CLI provides the [update-ca-certificate](#) command to deactivate a CA certificate.

```
aws iot update-ca-certificate \  
  --certificate-id certificateId \  
  --new-status INACTIVE
```

Use the [list-certificates-by-ca](#) command to get a list of all registered client certificates that were signed by the specified CA. For each client certificate signed by the specified CA certificate, use the [update-certificate](#) command to revoke the client certificate to prevent it from being used.

Use the [describe-ca-certificate](#) command to see the status of the CA certificate.

Create a client certificate using your CA certificate

You can use your own certificate authority (CA) to create client certificates. The client certificate must be registered with AWS IoT before use. For information about the registration options for your client certificates, see [Register a client certificate \(p. 290\)](#).

Create a client certificate (CLI)

Note

You can't perform this procedure in the AWS IoT console.

To create a client certificate using the AWS CLI

1. Generate a key pair.

```
openssl genrsa -out device_cert_key_filename 2048
```

2. Create a CSR for the client certificate.

```
openssl req -new \  
  -key device_cert_key_filename \  
  -out device_cert_csr_filename
```

You are prompted for some information, as shown here:

```
You are about to be asked to enter information that will be incorporated  
into your certificate request.  
What you are about to enter is what is called a Distinguished Name or a DN.  
There are quite a few fields but you can leave some blank  
For some fields there will be a default value,  
If you enter '.', the field will be left blank.  
----  
Country Name (2 letter code) [AU]:  
State or Province Name (full name) []:  
Locality Name (for example, city) []:  
Organization Name (for example, company) []:
```

```
Organizational Unit Name (for example, section) []:
Common Name (e.g. server FQDN or YOUR name) []:
Email Address []:
```

```
Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

3. Create a client certificate from the CSR.

```
openssl x509 -req \
-in device_cert_csr_filename \
-CA root_CA_pem_filename \
-CAkey root_CA_key_filename \
-CACreateserial \
-out device_cert_pem_filename \
-days 500 -sha256
```

At this point, the client certificate has been created, but it has not yet been registered with AWS IoT. For information about how and when to register the client certificate, see [Register a client certificate \(p. 290\)](#).

Register a client certificate

Client certificates must be registered with AWS IoT to enable communications between the client and AWS IoT. You can register each client certificate manually, or you can configure the client certificates to register automatically when the client connects to AWS IoT for the first time.

If you want your clients and devices to register their client certificates when they first connect, you must [Register your CA certificate \(p. 286\)](#) used to sign the client certificate with AWS IoT in the Regions in which you want to use it. The Amazon Root CA is automatically registered with AWS IoT.

Client certificates can be shared by AWS accounts and Regions. The procedures in these topics must be performed in each account and Region in which you want to use the client certificate. The registration of a client certificate in one account or Region is not automatically recognized by another.

Note

Clients that use the Transport Layer Security (TLS) protocol to connect to AWS IoT must support the [Server Name Indication \(SNI\) extension](#) to TLS. For more information, see [Transport security in AWS IoT \(p. 362\)](#).

Topics

- [Register a client certificate manually \(p. 290\)](#)
- [Register a client certificate when the client connects to AWS IoT just-in-time registration \(JITR\) \(p. 293\)](#)

Register a client certificate manually

You can register a client certificate manually by using the AWS IoT console and AWS CLI.

The registration procedure to use depends on whether the certificate will be shared by shared by AWS accounts and Regions. The registration of a client certificate in one account or Region is not automatically recognized by another.

The procedures in this topic must be performed in each account and Region in which you want to use the client certificate. Client certificates can be shared by AWS accounts and Regions, but only if the client certificate is signed by a certificate authority (CA) that is NOT registered with AWS IoT.

Register a client certificate signed by a registered CA (console)

Note

Before you perform this procedure, make sure that you have the client certificate's .pem file and that the client certificate was signed by a CA that you have [registered with AWS IoT \(p. 286\)](#).

To register an existing certificate with AWS IoT using the console

1. Sign in to the AWS Management Console and open the [AWS IoT console](#).
2. In the left navigation pane, choose **Secure**, choose **Certificates**, and then choose **Create**.
3. On **Create a certificate**, locate the **Use my certificate** entry, and choose **Get started**.
4. On **Select a CA**:
 - **If the client certificates are signed by a CA that is registered with AWS IoT**
Choose that CA from the list, and then choose **Next**.
 - **If the client certificates are not signed by a CA that is registered with AWS IoT**
See [Register a client certificate signed by an unregistered CA \(console\) \(p. 291\)](#).
 - **If the client certificates are signed by Amazon's CA**
Don't select any CA, just choose **Next**.

If the client certificates are not signed by a CA that is registered with AWS IoT, see [Register a client certificate signed by an unregistered CA \(console\) \(p. 291\)](#).

5. On **Register existing device certificates**, choose **Select certificates**, and select up to 10 certificate files to register.
6. After closing the file dialog box, select whether you want to activate or revoke the client certificates when you register them.

If you don't activate a certificate when it is registered, [Activate a client certificate \(console\) \(p. 294\)](#) describes how to activate it later.

If a certificate is revoked when it is registered, it can't be activated later.

After you choose the certificate files to register, and select the actions to take after registration, select **Register certificates**.

The client certificates that are registered successfully appear in the list of certificates.

Register a client certificate signed by an unregistered CA (console)

Note

Before you perform this procedure, make sure that you have the client certificate's .pem file.

To register an existing certificate with AWS IoT using the console

1. Sign in to the AWS Management Console and open the [AWS IoT console](#).
2. In the left navigation pane, choose **Secure**, choose **Certificates**, and then choose **Create**.
3. On **Create a certificate**, locate the **Use my certificate** entry, and choose **Get started**.
4. On **Select a CA**, choose **Next**.
5. On **Register existing device certificates**, choose **Select certificates**, and select up to 10 certificate files to register.
6. After closing the file dialog box, select whether you want to activate or revoke the client certificates when you register them.

If you don't activate a certificate when it is registered, [Activate a client certificate \(console\) \(p. 294\)](#) describes how to activate it later.

If a certificate is revoked when it is registered, it can't be activated later.

After you choose the certificate files to register, and select the actions to take after registration, select **Register certificates**.

The client certificates that are registered successfully appear in the list of certificates.

Register a client certificate signed by a registered CA (CLI)

Note

Before you perform this procedure, make sure that you have the certificate authority (CA) .pem and the client certificate's .pem file. The client certificate must be signed by a certificate authority (CA) that you have [registered with AWS IoT \(p. 286\)](#).

Use the **register-certificate** command to register, but not activate, a client certificate.

```
aws iot register-certificate \
--certificate-pem file://device_cert_pem_filename \
--ca-certificate-pem file://ca_cert_pem_filename
```

The client certificate is registered with AWS IoT, but it is not active yet. See [Activate a client certificate \(CLI\) \(p. 295\)](#) for information on how to activate it later.

You can also activate the client certificate when you register it by using this command.

```
aws iot register-certificate \
--set-as-active \
--certificate-pem file://device_cert_pem_filename \
--ca-certificate-pem file://ca_cert_pem_filename
```

For more information about activating the certificate so that it can be used to connect to AWS IoT, see [Activate or deactivate a client certificate \(p. 294\)](#)

Register a client certificate signed by an unregistered CA (CLI)

Note

Before you perform this procedure, make sure that you have the certificate's .pem file.

Use the **register-certificate-without-ca** command to register, but not activate, a client certificate.

```
aws iot register-certificate-without-ca \
--certificate-pem file://device_cert_pem_filename
```

The client certificate is registered with AWS IoT, but it is not active yet. See [Activate a client certificate \(CLI\) \(p. 295\)](#) for information on how to activate it later.

You can also activate the client certificate when you register it by using this command.

```
aws iot register-certificate-without-ca \
--status ACTIVE \
--certificate-pem file://device_cert_pem_filename
```

For more information about activating the certificate so that it can be used to connect to AWS IoT, see [Activate or deactivate a client certificate \(p. 294\)](#).

Register a client certificate when the client connects to AWS IoT just-in-time registration (JITR)

You can configure a CA certificate to enable client certificates it has signed to register with AWS IoT automatically the first time the client connects to AWS IoT.

To register client certificates when a client connects to AWS IoT for the first time, you must enable the CA certificate for automatic registration and configure the first connection by the client to provide the required certificates.

Configure a CA certificate to support automatic registration (console)

To configure a CA certificate to support automatic client certificate registration using the AWS IoT console

1. Sign in to the AWS Management Console and open the [AWS IoT console](#).
2. In the left navigation pane, choose **Secure**, choose **CAs**.
3. In the list of certificate authorities, find the one for which you want to enable automatic registration, and open the option menu by using the ellipsis icon.
4. On the option menu, choose **Enable auto-registration**.

Note

The auto-registration status is not shown in the list of certificate authorities. To see the auto-registration status of a certificate authority, you must open the **Details** page of the certificate authority.

Configure a CA certificate to support automatic registration (CLI)

If you have already registered your CA certificate with AWS IoT, use the **update-ca-certificate** command to set `autoRegistrationStatus` of the CA certificate to `ENABLE`.

```
aws iot update-ca-certificate \
--certificate-id caCertificateId \
--new-auto-registration-status ENABLE
```

If you want to enable `autoRegistrationStatus` when you register the CA certificate, use the **register-ca-certificate** command.

```
aws iot register-ca-certificate \
--allow-auto-registration \
--ca-certificate file://root_CA_pem_filename \
--verification-cert file://verification_cert_pem_filename
```

Use the **describe-ca-certificate** command to see the status of the CA certificate.

Configure the first connection by a client for automatic registration

When a client attempts to connect to AWS IoT for the first time it must present a file that contains your client certificate signed by your CA certificate as part of the TLS handshake.

When the client connects to AWS IoT, use the `client_certificate_filename` file as the certificate file. AWS IoT recognizes the CA certificate as a registered CA certificate, registers the client certificate and sets its status to `PENDING_ACTIVATION`. This means that the client certificate was automatically registered and is awaiting activation. The client certificate's state must be `ACTIVE` before it can be used to connect to AWS IoT.

Note

You can provision devices using AWS IoT Core just-in-time registration (JITR) feature without having to send the entire trust chain on devices' first connection to AWS IoT Core. Presenting

the CA certificate is optional but the device is required to send the [Server Name Indication \(SNI\)](#) extension when they connect.

When AWS IoT automatically registers a certificate or when a client presents a certificate in the PENDING_ACTIVATION status, AWS IoT publishes a message to the following MQTT topic:

`$aws/events/certificates/registered/caCertificateID`

Where *caCertificateID* is the ID of the CA certificate that issued the client certificate.

The message published to this topic has the following structure:

```
{  
    "certificateId": "certificateId",  
    "caCertificateId": "caCertificateID",  
    "timestamp": timestamp,  
    "certificateStatus": "PENDING_ACTIVATION",  
    "awsAccountId": "awsAccountId",  
    "certificateRegistrationTimestamp": "certificateRegistrationTimestamp"  
}
```

You can create a rule that listens on this topic and performs some actions. We recommend that you create a Lambda rule that verifies the client certificate is not on a certificate revocation list (CRL), activates the certificate, and creates and attaches a policy to the certificate. The policy determines which resources the client can access. For more information about how to create a Lambda rule that listens on the `$aws/events/certificates/registered/caCertificateID` topic and performs these actions, see [just-in-time registration of Client Certificates on AWS IoT](#).

If any error or exception occurs during the auto-registration of the client certificates, AWS IoT sends events or messages to your logs in CloudWatch Logs. For more information about setting up the logs for your account, see the [Amazon CloudWatch documentation](#).

Activate or deactivate a client certificate

AWS IoT verifies that a client certificate is active when it authenticates a connection.

You can create and register client certificates without activating them so they can't be used until you want to use them. You can also deactivate active client certificates to disable them temporarily. Finally, you can revoke client certificates to prevent them from any future use.

Activate a client certificate (console)

To activate a client certificate using the AWS IoT console

1. Sign in to the AWS Management Console and open the [AWS IoT console](#).
2. In the left navigation pane, choose **Secure**, choose **Certificates**.
3. In the list of certificates, locate the certificate that you want to activate, and open the option menu by using the ellipsis icon.
4. In the option menu, choose **Activate**.

The certificate should show as **Active** in the list of certificates.

Deactivate a client certificate (console)

To deactivate a client certificate using the AWS IoT console

1. Sign in to the AWS Management Console and open the [AWS IoT console](#).
2. In the left navigation pane, choose **Secure**, choose **Certificates**.

3. In the list of certificates, locate the certificate that you want to deactivate, and open the option menu by using the ellipsis icon.
4. In the option menu, choose **Deactivate**.

The certificate should show as **Inactive** in the list of certificates.

Activate a client certificate (CLI)

The AWS CLI provides the [update-certificate](#) command to activate a certificate.

```
aws iot update-certificate \  
  --certificate-id certificateId \  
  --new-status ACTIVE
```

If the command was successful, the certificate's status will be **ACTIVE**. Run [describe-certificate](#) to see the certificate's status.

```
aws iot describe-certificate \  
  --certificate-id certificateId
```

Deactivate a client certificate (CLI)

The AWS CLI provides the [update-certificate](#) command to deactivate a certificate.

```
aws iot update-certificate \  
  --certificate-id certificateId \  
  --new-status INACTIVE
```

If the command was successful, the certificate's status will be **INACTIVE**. Run [describe-certificate](#) to see the certificate's status.

```
aws iot describe-certificate \  
  --certificate-id certificateId
```

Attach a thing or policy to a client certificate

When you create and register a certificate separate from an AWS IoT thing, it will not have any policies that authorize any AWS IoT operations, nor will it be associated with any AWS IoT thing object. This section describes how to add these relationships to a registered certificate.

Important

To complete these procedures, you must have already created the thing or policy that you want to attach to the certificate.

The certificate authenticates a device with AWS IoT so that it can connect. Attaching the certificate to a thing resource establishes the relationship between the device (by way of the certificate) and the thing resource. To authorize the device to perform AWS IoT actions, such as to allow the device to connect and publish messages, an appropriate policy must be attached to the device's certificate.

Attach a thing to a client certificate (console)

You will need the name of the thing object to complete this procedure.

To attach a thing object to a registered certificate

1. Sign in to the AWS Management Console and open the [AWS IoT console](#).

2. In the left navigation pane, choose **Secure**, choose **Certificates**.
3. In the list of certificates, locate the certificate to which you want to attach a policy, open the certificate's option menu by choosing the ellipsis icon, and choose **Attach thing**.
4. In the pop-up, locate the name of the thing you want to attach to the certificate, choose its check box, and choose **Attach**.

The thing object should now appear in the list of things on the certificate's details page.

[Attach a policy to a client certificate \(console\)](#)

You will need the name of the policy object to complete this procedure.

To attach a policy object to a registered certificate

1. Sign in to the AWS Management Console and open the [AWS IoT console](#).
2. In the left navigation pane, choose **Secure**, choose **Certificates**.
3. In the list of certificates, locate the certificate to which you want to attach a policy, open the certificate's option menu by choosing the ellipsis icon, and choose **Attach policy**.
4. In the pop-up, locate the name of the policy you want to attach to the certificate, choose its check box, and choose **Attach**.

The policy object should now appear in the list of policies on the certificate's details page.

[Attach a thing to a client certificate \(CLI\)](#)

The AWS CLI provides the [attach-thing-principal](#) command to attach a thing object to a certificate.

```
aws iot attach-thing-principal \
  --principal certificateArn \
  --thing-name thingName
```

[Attach a policy to a client certificate \(CLI\)](#)

The AWS CLI provides the [attach-policy](#) command to attach a policy object to a certificate.

```
aws iot attach-policy \
  --target certificateArn \
  --policy-name policyName
```

[Revoke a client certificate](#)

If you detect suspicious activity on a registered client certificate, you can revoke it so that it can't be used again.

[Revoke a client certificate \(console\)](#)

To revoke a client certificate using the AWS IoT console

1. Sign in to the AWS Management Console and open the [AWS IoT console](#).
2. In the left navigation pane, choose **Secure**, choose **Certificates**.
3. In the list of certificates, locate the certificate that you want to revoke, and open the option menu by using the ellipsis icon.
4. In the option menu, choose **Revoke**.

If the certificate was successfully revoked, it will show as **Revoked** in the list of certificates.

Revoke a client certificate (CLI)

The AWS CLI provides the [update-certificate](#) command to revoke a certificate.

```
aws iot update-certificate \  
  --certificate-id certificateId \  
  --new-status REVOKED
```

If the command was successful, the certificate's status will be **REVOKED**. Run [describe-certificate](#) to see the certificate's status.

```
aws iot describe-certificate \  
  --certificate-id certificateId
```

Transfer a certificate to another account

X.509 certificates that belong to one AWS account can be transferred to another AWS account.

To transfer an X.509 certificate from one AWS account to another

1. [the section called "Begin a certificate transfer" \(p. 297\)](#)

The certificate must be deactivated and detached from all policies and things before initiating the transfer.

2. [the section called "Accept or reject a certificate transfer" \(p. 299\)](#)

The receiving account must explicitly accept or reject the transferred certificate. After the receiving account accepts the certificate, the certificate must be activated before use.

3. [the section called "Cancel a certificate transfer" \(p. 299\)](#)

The originating account can cancel a transfer, if the certificate has not been accepted.

Begin a certificate transfer

You can begin to transfer a certificate to another AWS account by using the [AWS IoT console](#) or the AWS CLI.

Begin a certificate transfer (console)

To complete this procedure, you'll need the ID of the certificate that you want to transfer.

Do this procedure from the account with the certificate to transfer.

To begin to transfer a certificate to another AWS account

1. Sign in to the AWS Management Console and open the [AWS IoT console](#).
2. In the left navigation pane, choose **Secure**, choose **Certificates**.

Choose the certificate with an **Active** or **Inactive** status that you want to transfer and open its details page.

3. On the certificate's **Details** page, in the **Actions** menu, if the **Deactivate** option is available, choose the **Deactivate** option to deactivate the certificate.

4. On the certificate's **Details** page, in the left menu, choose **Policies**.

5. On the certificate's **Policies** page, if there are any policies attached to the certificate, detach each one by opening the policy's options menu and choosing **Detach**.
The certificate must not have any attached policies before you continue.
6. On the certificate's **Policies** page, in the left menu, choose **Things**.
7. On the certificate's **Things** page, if there are any things attached to the certificate, detach each one by opening the thing's options menu and choosing **Detach**.
The certificate must not have any attached things before you continue.
8. On the certificate's **Things** page, in the left menu, choose **Details**.
9. On the certificate's **Details** page, in the **Actions** menu, choose **Start transfer** to open the **Start transfer** dialog box.
10. In the **Start transfer** dialog box, enter the AWS account number of the account to receive the certificate and an optional short message.
11. Choose **Start transfer** to transfer the certificate.

The console should display a message that indicates the success or failure of the transfer. If the transfer was started, the certificate's status is updated to **Transferred**.

Begin a certificate transfer (CLI)

To complete this procedure, you'll need the `certificateId` and the `certificateArn` of the certificate that you want to transfer.

Do this procedure from the account with the certificate to transfer.

To begin to transfer a certificate to another AWS account

1. Use the **update-certificate** command to deactivate the certificate.

```
aws iot update-certificate --certificate-id certificateId --new-status INACTIVE
```

2. Detach all policies.

1. Use the **list-attached-policies** command to list the policies attached to the certificate.

```
aws iot list-attached-policies --target certificateArn
```

2. For each attached policy, use the **detach-policy** command to detach the policy.

```
aws iot detach-policy --target certificateArn --policy-name policy-name
```

3. Detach all things.

1. Use the **list-principal-things** command to list the things attached to the certificate.

```
aws iot list-principal-things --principal certificateArn
```

2. For each attached thing, use the **detach-thing-principal** command to detach the thing.

```
aws iot detach-thing-principal --principal certificateArn --thing-name thing-name
```

4. Use the **transfer-certificate** command to start the certificate transfer.

```
aws iot transfer-certificate --certificate-id certificateId --target-aws-account account-id
```

Accept or reject a certificate transfer

You can accept or reject a certificate transferred to your AWS account from another AWS account by using the [AWS IoT console](#) or the AWS CLI.

Accept or reject a certificate transfer (console)

To complete this procedure, you'll need the ID of the certificate that was transferred to your account.

Do this procedure from the account receiving the certificate that was transferred.

To accept or reject a certificate that was transferred to your AWS account

1. Sign in to the AWS Management Console and open the [AWS IoT console](#).
2. In the left navigation pane, choose **Secure**, choose **Certificates**.

Choose the certificate with a status of **Pending transfer** that you want to accept or reject and open its details page.

3. On the certificate's **Details** page, in the **Actions** menu,
 - To accept the certificate, choose **Accept transfer**.
 - To not accept the certificate, choose **Reject transfer**.

Accept or reject a certificate transfer (CLI)

To complete this procedure, you'll need the `certificateId` of the certificate transfer that you want to accept or reject.

Do this procedure from the account receiving the certificate that was transferred.

To accept or reject a certificate that was transferred to your AWS account

1. Use the `accept-certificate-transfer` command to accept the certificate.

```
aws iot accept-certificate-transfer --certificate-id certificateId
```

2. Use the `reject-certificate-transfer` command to reject the certificate.

```
aws iot reject-certificate-transfer --certificate-id certificateId
```

Cancel a certificate transfer

You can cancel a certificate transfer before it has been accepted by using the [AWS IoT console](#) or the AWS CLI.

Cancel a certificate transfer (console)

To complete this procedure, you'll need the ID of the certificate transfer that you want to cancel.

Do this procedure from the account that initiated the certificate transfer.

To cancel a certificate transfer

1. Sign in to the AWS Management Console and open the [AWS IoT console](#).
2. In the left navigation pane, choose **Secure**, choose **Certificates**.

- Choose the certificate with **Transferred** status whose transfer you want to cancel and open its options menu.
3. On the certificate's options menu, choose the **Revoke transfer** option to cancel the certificate transfer.

Important

Be careful not to mistake the **Revoke transfer** option with the **Revoke** option.

The **Revoke transfer** option cancels the certificate transfer, while the **Revoke** option makes the certificate irreversibly unusable by AWS IoT.

[Cancel a certificate transfer \(CLI\)](#)

To complete this procedure, you'll need the *certificateId* of the certificate transfer that you want to cancel.

Do this procedure from the account that initiated the certificate transfer.

Use the **cancel-certificate-transfer** command to cancel the certificate transfer.

```
aws iot cancel-certificate-transfer --certificate-id certificateId
```

IAM users, groups, and roles

IAM users, groups, and roles are the standard mechanisms for managing identity and authentication in AWS. You can use them to connect to AWS IoT HTTP interfaces using the AWS SDK and AWS CLI.

IAM roles also allow AWS IoT to access other AWS resources in your account on your behalf. For example, if you want to have a device publish its state to a DynamoDB table, IAM roles allow AWS IoT to interact with Amazon DynamoDB. For more information, see [IAM Roles](#).

For message broker connections over HTTP, AWS IoT authenticates IAM users, groups, and roles using the Signature Version 4 signing process. For information, see [Signing AWS API Requests](#).

When using AWS Signature Version 4 with AWS IoT, clients must support the following in their TLS implementation:

- TLS 1.2, TLS 1.1, TLS 1.0
- SHA-256 RSA certificate signature validation
- One of the cipher suites from the TLS cipher suite support section

For information, see [Identity and access management for AWS IoT \(p. 364\)](#).

Amazon Cognito identities

Amazon Cognito Identity enables you to create temporary, limited-privilege AWS credentials for use in mobile and web applications. When you use Amazon Cognito Identity, you create identity pools that create unique identities for your users and authenticate them with identity providers like Login with Amazon, Facebook, and Google. You can also use Amazon Cognito identities with your own developer authenticated identities. For more information, see [Amazon Cognito Identity](#).

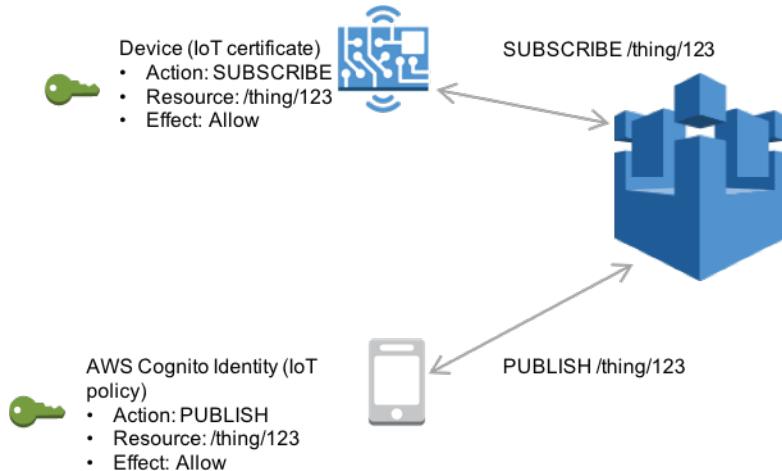
To use Amazon Cognito Identity, you define a Amazon Cognito identity pool that is associated with an IAM role. The IAM role is associated with an IAM policy that grants identities from your identity pool permission to access AWS resources like calling AWS services.

Amazon Cognito Identity creates unauthenticated and authenticated identities. Unauthenticated identities are used for guest users in a mobile or web application who want to use the app without

signing in. Unauthenticated users are granted only those permissions specified in the IAM policy associated with the identity pool.

When you use authenticated identities, in addition to the IAM policy attached to the identity pool, you must attach an AWS IoT policy to an Amazon Cognito Identity by using the [AttachPolicy](#) API and give permissions to an individual user of your AWS IoT application. You can use the AWS IoT policy to assign fine-grained permissions for specific customers and their devices.

Authenticated and unauthenticated users are different identity types. If you don't attach an AWS IoT policy to the Amazon Cognito Identity, an authenticated user fails authorization in AWS IoT and doesn't have access to AWS IoT resources and actions. For more information about creating policies for Amazon Cognito identities, see [Publish/Subscribe policy examples \(p. 332\)](#) and [Authorization with Amazon Cognito identities \(p. 353\)](#).



Custom authentication

AWS IoT Core lets you define custom authorizers so that you can manage your own client authentication and authorization. This is useful when you need to use authentication mechanisms other than the ones that AWS IoT Core natively supports. (For more information about the natively supported mechanisms, see the section called ["Client authentication" \(p. 282\)](#)).

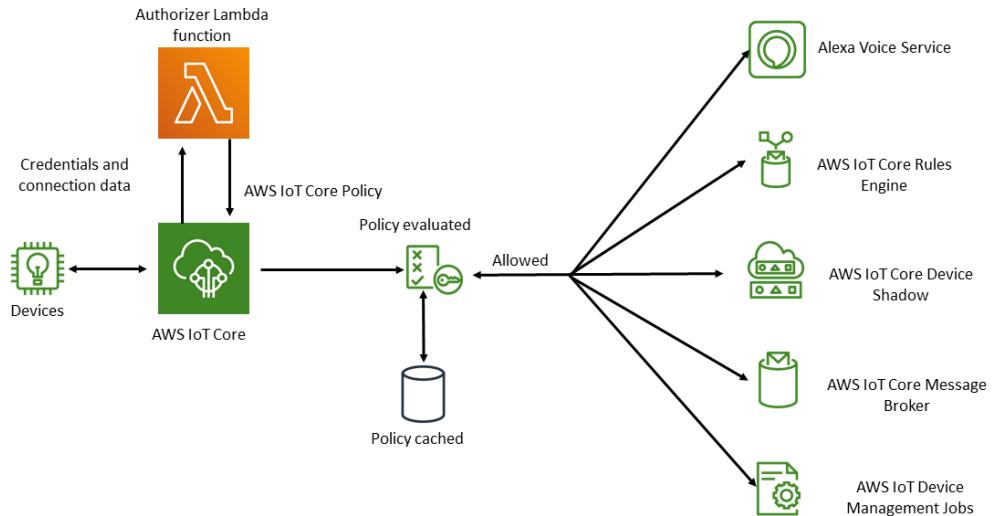
For example, if you are migrating existing devices in the field to AWS IoT Core and these devices use a custom bearer token or MQTT user name and password to authenticate, you can migrate them to AWS IoT Core without having to provision new identities for them. You can use custom authentication with any of the communication protocols that AWS IoT Core supports. For more information about the protocols that AWS IoT Core supports, see the section called ["Device communication protocols" \(p. 77\)](#).

Topics

- [Understanding the custom authentication workflow \(p. 301\)](#)
- [Creating and managing custom authorizers \(p. 303\)](#)
- [Connecting to AWS IoT Core by using custom authentication \(p. 308\)](#)
- [Troubleshooting your authorizers \(p. 310\)](#)

Understanding the custom authentication workflow

Custom authentication enables you to define how to authenticate and authorize clients by using [authorizer resources](#). Each authorizer contains of a reference to a customer-managed Lambda function, an optional public key for validating device credentials, and additional configuration information. The following diagram illustrates the authorization workflow for custom authentication in AWS IoT Core.



AWS IoT Core custom authentication and authorization workflow

The following list explains each step in the custom authentication and authorization workflow.

1. A device connects to a customer's AWS IoT Core data endpoint by using one of the supported [the section called "Device communication protocols" \(p. 77\)](#). The device passes credentials in either the request's header fields or query parameters (for the HTTP Publish or MQTT over WebSockets protocols) or in the user name and password field of the MQTT CONNECT message (for the MQTT and MQTT over WebSockets protocols).
2. AWS IoT Core checks for one of two conditions:
 - The incoming request specifies an authorizer.
 - The AWS IoT Core data endpoint receiving the request has a default authorizer configured for it.
 If AWS IoT Core finds an authorizer in either of these ways, AWS IoT Core triggers the Lambda function associated with the authorizer.
3. (Optional) If you've enabled token signing, AWS IoT Core validates the request signature by using the public key stored in the authorizer before triggering the Lambda function. If validation fails, AWS IoT Core stops the request without invoking the Lambda function.
4. The Lambda function receives the credentials and connection metadata in the request and makes an authentication decision.
5. The Lambda function returns the results of the authentication decision and an AWS IoT Core policy document that specifies what actions are allowed in the connection. The Lambda function also returns information that specifies how often AWS IoT Core revalidates the credentials in the request by invoking the Lambda function.
6. AWS IoT Core evaluates activity on the connection against the policy it has received from the Lambda function.

Scaling considerations

Because a Lambda function handles authentication and authorization for your authorizer, the function is subject to Lambda pricing and service limits, such as concurrent execution rate. For more information

about Lambda pricing, see [Lambda Pricing](#). You can manage the load on your Lambda function by adjusting the `refreshAfterInSeconds` and `disconnectAfterInSeconds` parameters in your Lambda function response. For more information about the contents of your Lambda function response, see the section called “[Defining your Lambda function](#)” (p. 303).

Note

If you leave signing enabled, you can prevent excessive triggering of your Lambda by unrecognized clients. Consider this before you disable signing in your authorizer.

Creating and managing custom authorizers

AWS IoT Core implements custom authentication and authorization schemes by using [authorizer resources](#). Each authorizer consists of the following components:

- *Name*: A unique user-defined string that identifies the authorizer.
- *Lambda function ARN*: The Amazon Resource Name (ARN) of the Lambda function that implements the authorization and authentication logic.
- *Token key name*: The key name used to extract the token from the HTTP headers, query parameters, or MQTT CONNECT user name in order to perform signature validation. This value is required if signing is enabled in your authorizer.
- *Signing disabled flag (optional)*: A Boolean value that specifies whether to disable the signing requirement on credentials. This is useful for scenarios where signing the credentials doesn't make sense, such as authentication schemes that use MQTT user name and password. The default value is `false`, so signing is enabled by default.
- *Token signing public key*: The public key that AWS IoT Core uses to validate the token signature. Its minimum length is 2,048 bits. This value is required if signing is enabled in your authorizer.

Lambda charges you for the number of times your Lambda function runs and for the amount of time it takes for the code in your function to execute. For more information about Lambda pricing, see [Lambda Pricing](#). For more information about creating Lambda functions, see the [Lambda Developer Guide](#).

Note

If you leave signing enabled, you can prevent excessive triggering of your Lambda by unrecognized clients. Consider this before you disable signing in your authorizer.

Defining your Lambda function

When AWS IoT Core invokes your authorizer, it triggers the associated Lambda associated with the authorizer with an event that contains the following JSON object. The example JSON object contains all of the possible fields. Any fields that aren't relevant to the connection request aren't included.

```
{  
    "token" : "aToken",  
    "signatureVerified": Boolean, // Indicates whether the device gateway has validated the  
    // signature.  
    "protocols": ["tls", "http", "mqtt"], // Indicates which protocols to expect for the  
    // request.  
    "protocolData": {  
        "tls" : {  
            "serverName": "serverName" // The server name indication (SNI) host_name  
            string.  
        },  
        "http": {  
            "headers": {  
                "#{name}": "#{value}"  
            },  
            "queryString": "?#{name}=#{value}"  
        }  
    }  
}
```

```

        },
        "mqtt": {
            "username": "myUserName",
            "password": "myPassword", // A base64-encoded string.
            "clientId": "myClientId" // Included in the event only when the device sends
the value.
        }
    },
    "connectionMetadata": {
        "id": UUID // The connection ID. You can use this for logging.
    },
}

```

The Lambda function should use this information to authenticate the incoming connection and decide what actions are permitted in the connection. The function should send a response that contains the following values.

- **isAuthenticated**: A Boolean value that indicates whether the request is authenticated.
- **principalId**: An alphanumeric string that acts as an identifier for the token sent by the custom authorization request. The value must be an alphanumeric string with at least one, and no more than 128, characters and match this regular expression (regex) pattern: ([a-zA-Z0-9]){1,128}.
- **policyDocuments**: A list of JSON-formatted AWS IoT Core policy documents For more information about creating AWS IoT Core policies, see [the section called “AWS IoT Core policies” \(p. 314\)](#). The maximum number of policy documents is 10 policy documents. Each policy document can contain a maximum of 2,048 characters.
- **disconnectAfterInSeconds**: An integer that specifies the maximum duration (in seconds) of the connection to the AWS IoT Core gateway. The minimum value is 300 seconds, and the maximum value is 86,400 seconds.
- **refreshAfterInSeconds**: An integer that specifies the interval between policy refreshes. When this interval passes, AWS IoT Core invokes the Lambda function to allow for policy refreshes. The minimum value is 300 seconds, and the maximum value is 86,400 seconds.

The following JSON object contains an example of a response that your Lambda function can send.

```
{
    "isAuthenticated": true, //A Boolean that determines whether client can connect.
    "principalId": "xxxxxxxx", //A string that identifies the connection in logs.
    "disconnectAfterInSeconds": 86400,
    "refreshAfterInSeconds": 300,
    "policyDocuments": [
        {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Action": "iot:Publish",
                    "Effect": "Allow",
                    "Resource": "arn:aws:iot:us-east-1:<your_aws_account_id>:topic/
customauthtesting"
                }
            ]
        }
    ]
}
```

The **policyDocument** value must contain a valid AWS IoT Core policy document. For more information about AWS IoT Core policies, see [the section called “AWS IoT Core policies” \(p. 314\)](#). In MQTT over TLS and MQTT over WebSockets connections, AWS IoT Core caches this policy for the interval specified in the value of the **refreshAfterInSeconds** field. In the case of HTTP connections the Lambda function

is called for every authorization request unless your device is using HTTP persistent connections (also called HTTP keep-alive or HTTP connection reuse) you can choose to enable caching when configuring the authorizer. During this interval, AWS IoT Core authorizes actions in an established connection against this cached policy without triggering your Lambda function again. If failures occur during custom authentication, AWS IoT Core terminates the connection. AWS IoT Core also terminates the connection if it has been open for longer than the value specified in the `disconnectAfterInSeconds` parameter.

The following JavaScript contains a sample Node.js Lambda function that looks for a password in the MQTT Connect message with a value of `test` and returns a policy that grants permission to connect to AWS IoT Core with a client named `myClientName` and publish to a topic that contains the same client name. If it doesn't find the expected password, it returns a policy that denies those two actions.

```
// A simple Lambda function for an authorizer. It demonstrates
// how to parse an MQTT password and generate a response.

exports.handler = function(event, context, callback) {
    var uname = event.protocolData.mqtt.username;
    var pwd = event.protocolData.mqtt.password;
    var buff = new Buffer(pwd, 'base64');
    var passwd = buff.toString('ascii');
    switch (passwd) {
        case 'test':
            callback(null, generateAuthResponse(passwd, 'Allow'));
        default:
            callback(null, generateAuthResponse(passwd, 'Deny'));
    }
};

// Helper function to generate the authorization response.
var generateAuthResponse = function(token, effect) {
    var authResponse = {};
    authResponse.isAuthenticated = true;
    authResponse.principalId = 'TEST123';

    var policyDocument = {};
    policyDocument.Version = '2012-10-17';
    policyDocument.Statement = [];
    var publishStatement = {};
    var connectStatement = {};
    connectStatement.Action = ["iot:Connect"];
    connectStatement.Effect = effect;
    connectStatement.Resource = ["arn:aws:iot:us-east-1:123456789012:client/myClientName"];
    publishStatement.Action = ["iot:Publish"];
    publishStatement.Effect = effect;
    publishStatement.Resource = ["arn:aws:iot:us-east-1:123456789012:topic/telemetry/
myClientName"];
    policyDocument.Statement[0] = connectStatement;
    policyDocument.Statement[1] = publishStatement;
    authResponse.policyDocuments = [policyDocument];
    authResponse.disconnectAfterInSeconds = 3600;
    authResponse.refreshAfterInSeconds = 300;

    return authResponse;
}
```

This Lambda function returns the following values when it receives the expected value of `test` in the MQTT Connect password field.

```
{
    "password": "password",
```

```

    "isAuthenticated": true,
    "principalId": "principalId",
    "policyDocuments": [
        {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Action": "iot:Connect",
                    "Effect": "Allow",
                    "Resource": "*"
                },
                {
                    "Action": "iot:Publish",
                    "Effect": "Allow",
                    "Resource": "arn:aws:region:accountId:topic/telemetry/${iot:ClientId}"
                },
                {
                    "Action": "iot:Subscribe",
                    "Effect": "Allow",
                    "Resource": "arn:aws:iot:region:accountId:topicfilter/telemetry/${iot:ClientId}"
                },
                {
                    "Action": "iot:Receive",
                    "Effect": "Allow",
                    "Resource": "arn:aws:iot:region:accountId:topic/telemetry/${iot:ClientId}"
                }
            ]
        ],
        "disconnectAfterInSeconds": 3600,
        "refreshAfterInSeconds": 300
    }
}

```

Creating an authorizer

You can create an authorizer by using the [CreateAuthorizer API](#). The following example shows how to do this.

```

aws iot create-authorizer
--authorizer-name MyAuthorizer
--authorizer-function-arn arn:aws:lambda:us-
west-2:<account_id>:function:MyAuthorizerFunction //The ARN of the Lambda function.
[--token-key-name MyAuthorizerToken //The key used to extract the token from headers.
[--token-signing-public-keys FirstKey=
"-----BEGIN PUBLIC KEY-----
[...insert your public key here...]
-----END PUBLIC KEY-----"
[--status ACTIVE]
[--tags <value>]
[--signing-disabled | --no-signing-disabled]

```

You can use the `signing-disabled` parameter to opt out of signature validation for each invocation of your authorizer. We strongly recommend that you do not disable signing unless you have to. Signature validation protects you against excessive invocations of your Lambda function from unknown devices. You can't update the `signing-disabled` status of an authorizer after you create it. To change this behavior, you must create another custom authorizer with a different value for the `signing-disabled` parameter.

Values for the `tokenKeyName` and `tokenSigningPublicKeys` parameters are optional if you have disabled signing. They are required values if signing is enabled.

After you create your Lambda function and the custom authorizer, you must explicitly grant the AWS IoT Core service permission to invoke the function on your behalf. You can do this with the following command.

```
aws lambda add-permission --function-name <lambda_function_name> --principal iot.amazonaws.com --source-arn <authorizer_arn> --statement-id Id-123 --action "lambda:InvokeFunction"
```

Testing your authorizers

You can use the [TestInvokeAuthorizer](#) API to test the invocation and return values of your authorizer. This API enables you to specify protocol metadata and test the signature validation in your authorizer.

The following tabs show how to use the AWS CLI to test your authorizer.

Unix-like

```
aws iot test-invoke-authorizer --authorizer-name NAME_OF_AUTHORIZER \
--token TOKEN_VALUE --token-signature TOKEN_SIGNATURE
```

Windows CMD

```
aws iot test-invoke-authorizer --authorizer-name NAME_OF_AUTHORIZER ^
--token TOKEN_VALUE --token-signature TOKEN_SIGNATURE
```

Windows PowerShell

```
aws iot test-invoke-authorizer --authorizer-name NAME_OF_AUTHORIZER ^
--token TOKEN_VALUE --token-signature TOKEN_SIGNATURE
```

The value of the token-signature parameter is the signed token. To learn how to obtain this value, see [the section called "Signing the token" \(p. 310\)](#).

If your authorizer takes a user name and password, you can pass this information by using the --mqtt-context parameter. The following tabs show how to use the [TestInvokeAuthorizer](#) API to send a JSON object that contains a user name, password, and client name to your custom authorizer.

Unix-like

```
aws iot test-invoke-authorizer --authorizer-name NAME_OF_AUTHORIZER \
--mqtt-context '{"username": "'USER_NAME'", "password": "dGVzdA==",
"clientId": "'CLIENT_NAME'"}'
```

Windows CMD

```
aws iot test-invoke-authorizer --authorizer-name NAME_OF_AUTHORIZER ^
--mqtt-context '{"username": "'USER_NAME'", "password": "dGVzdA==",
"clientId": "'CLIENT_NAME'"}'
```

Windows PowerShell

```
aws iot test-invoke-authorizer --authorizer-name NAME_OF_AUTHORIZER --mqtt-context '{"username": "USER_NAME", "password": "dGVzdA==", "clientId": "CLIENT_NAME"}'
```

The password must be base64-encoded. The following example shows how to encode a password in a Unix-like environment.

```
echo -n PASSWORD | base64
```

Managing custom authorizers

You can manage your authorizers by using the following APIs.

- [ListAuthorizers](#): Show all authorizers in your account.
- [DescribeAuthorizer](#): Displays properties of the specified authorizer. These values include creation date, last modified date, and other attributes.
- [SetDefaultAuthorizer](#): Specifies the default authorizer for your AWS IoT Core data endpoints. AWS IoT Core uses this authorizer if a device doesn't pass AWS IoT Core credentials and doesn't specify an authorizer. For more information about using AWS IoT Core credentials, see [the section called "Client authentication" \(p. 282\)](#).
- [UpdateAuthorizer](#): Changes the status, token key name, or public keys for the specified authorizer.
- [DeleteAuthorizer](#): Deletes the specified authorizer.

Note

You can't update an authorizer's signing requirement. This means that you can't disable signing in an existing authorizer that requires it. You also can't require signing in an existing authorizer that doesn't require it.

Connecting to AWS IoT Core by using custom authentication

Devices can connect to AWS IoT Core by using custom authentication with any protocol that AWS IoT Core supports for device messaging. For more information about supported communication protocols, see [the section called "Device communication protocols" \(p. 77\)](#). The connection data that you pass to your authorizer Lambda function depends on the protocol you use. For more information about creating your authorizer Lambda function, see [the section called "Defining your Lambda function" \(p. 303\)](#). The following sections explain how to connect to authenticate by using each supported protocol.

HTTP

Devices sending data to AWS IoT Core by using the [HTTP Publish API](#) can pass credentials either through request headers or query parameters in their HTTP POST requests. Devices can specify an authorizer to invoke by using the `x-amz-customauthorizer-name` header or query parameter. If you have token signing enabled in your authorizer, you must pass the `token-key-name` and `x-amz-customauthorizer-signature` in either request headers or query parameters. The `token-signature` value must be URL-encoded.

The following example requests show how you pass these parameters in both request headers and query parameters.

```
//Passing credentials via headers
```

```
POST /topics/topic?qos=qos HTTP/1.1
Host: your-endpoint
x-amz-customauthorizer-signature: token-signature
token-key-name: token-value
x-amz-customauthorizer-name: authorizer-name

//Passing credentials via query parameters
POST /topics/topic?qos=qos&x-amz-customauthorizer-signature=token-signature&token-key-name=token-value HTTP/1.1
```

MQTT

Devices connecting to AWS IoT Core by using an MQTT connection can pass credentials through the `username` and `password` fields of MQTT messages. The `username` value can also optionally contain a query string that passes additional values (including a token, signature, and authorizer name) to your authorizer. You can use this query string if you want to use a token-based authentication scheme instead of `username` and `password` values.

Note

Data in the `password` field is base64-encoded by AWS IoT Core. Your Lambda function must decode it.

The following example contains a `username` string that contains extra parameters that specify a token and signature.

```
username?x-amz-customauthorizer-name=authorizer-name&x-amz-customauthorizer-signature=token-signature&token-key-name=token-value
```

In order to invoke an authorizer, devices connecting to AWS IoT Core by using MQTT and custom authentication must connect on port 443. They also must pass the Application Layer Protocol Negotiation (ALPN) TLS extension with a value of `mqtt` and the Server Name Indication (SNI) extension with the host name of their AWS IoT Core data endpoint. For more information about these values, see the section called “Device communication protocols” (p. 77). The V2 [AWS IoT Device SDKs, Mobile SDKs, and AWS IoT Device Client](#) (p. 1127) can configure both of these extensions.

MQTT over WebSockets

Devices connecting to AWS IoT Core by using MQTT over WebSockets can pass credentials in one of the two following ways.

- Through request headers or query parameters in the HTTP UPGRADE request to establish the WebSockets connection.
- Through the `username` and `password` fields in the MQTT CONNECT message.

If you pass credentials through the MQTT connect message, the ALPN and SNI TLS extensions are required. For more information about these extensions, see the section called “MQTT” (p. 309). The following example demonstrates how to pass credentials through the HTTP Upgrade request.

```
GET /mqtt HTTP/1.1
Host: your-endpoint
Upgrade: WebSocket
Connection: Upgrade
x-amz-customauthorizer-signature: token-signature
token-key-name: token-value
sec-WebSocket-Key: any random base64 value
sec-websocket-protocol: mqtt
sec-WebSocket-Version: websocket version
```

Signing the token

You must sign the token with the private key of the public-private key pair that you used in the `create-authorizer` call. The following examples show how to create the token signature by using a UNIX-like command and JavaScript. They use the SHA-256 hash algorithm to encode the signature.

Command line

```
echo -n TOKEN_VALUE | openssl dgst -sha256 -sign PEM encoded RSA private key | openssl base64
```

JavaScript

```
const crypto = require('crypto')

const key = "PEM encoded RSA private key"

const k = crypto.createPrivateKey(key)
let sign = crypto.createSign('SHA256')
sign.write(t)
sign.end()
const s = sign.sign(k, 'base64')
```

Troubleshooting your authorizers

This topic walks through common issues that can cause problems in custom authentication workflows and steps for resolving them. To troubleshoot issues most effectively, enable CloudWatch logs for AWS IoT Core and set the log level to **DEBUG**. You can enable CloudWatch logs in the AWS IoT Core console (<https://console.aws.amazon.com/iot/>). For more information about enabling and configuring logs for AWS IoT Core, see [the section called “Configure AWS IoT logging” \(p. 400\)](#).

Note

If you leave the log level at **DEBUG** for long periods of time, CloudWatch might store large amounts of logging data. This can increase your CloudWatch charges. Consider using resource-based logging to increase the verbosity for only devices in a particular thing group. For more information about resource-based logging, see [the section called “Configure AWS IoT logging” \(p. 400\)](#). Also, when you’re done troubleshooting, reduce the log level to a less verbose level.

Before you start troubleshooting, review [the section called “Understanding the custom authentication workflow” \(p. 301\)](#) for a high-level view of the custom authentication process. This helps you understand where to look for the source of a problem.

This topic discusses the following two areas for you to investigate.

- Issues related to your authorizer’s Lambda function.
- Issues related to your device.

Check for issues in your authorizer’s Lambda function

Perform the following steps to make sure that your devices’ connection attempts are invoking your Lambda function.

1. Verify which Lambda function is associated with your authorizer.

You can do this by calling the [DescribeAuthorizer](#) API or by clicking on the desired authorizer in the **Secure** section of the AWS IoT Core console.

2. Check the invocation metrics for the Lambda function. Perform the following steps to do this.
 - a. Open the AWS Lambda console (<https://console.aws.amazon.com/lambda/>) and select the function that is associated with your authorizer.
 - b. Choose the **Monitor** tab and view metrics for the time frame that is relevant to your problem.
3. If you see no invocations, verify that AWS IoT Core has permission to invoke your Lambda function. If you see invocations, skip to the next step. Perform the following steps to verify that your Lambda function has the required permissions.
 - a. Choose the **Permissions** tab for your function in the AWS Lambda console.
 - b. Find the **Resource-based Policy** section at the bottom of the page. If your Lambda function has the required permissions, the policy looks like the following example.

```
{
    "Version": "2012-10-17",
    "Id": "default",
    "Statement": [
        {
            "Sid": "Id123",
            "Effect": "Allow",
            "Principal": {
                "Service": "iot.amazonaws.com"
            },
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-east-1:111111111111:function:FunctionName",
            "Condition": {
                "ArnLike": {
                    "AWS:SourceArn": "arn:aws:iot:us-east-1:111111111111:authorizer/  
AuthorizerName"
                },
                "StringEquals": {
                    "AWS:SourceAccount": "111111111111"
                }
            }
        }
    ]
}
```

- c. This policy grants the `InvokeFunction` permission on your function to the AWS IoT Core principal. If you don't see it, you'll have to add it by using the [AddPermission](#) API. The following example shows you how to do this by using the AWS CLI.

```
aws lambda add-permission --function-name FunctionName --principal  
iot.amazonaws.com --source-arn AuthorizerARN --statement-id Id-123 --action  
"lambda:InvokeFunction"
```

4. If you see invocations, verify that there are no errors. An error might indicate that the Lambda function isn't properly handling the connection event that AWS IoT Core sends to it.

For information about handling the event in your Lambda function, see [the section called "Defining your Lambda function" \(p. 303\)](#). You can use the test feature in the AWS Lambda console (<https://console.aws.amazon.com/lambda/>) to hard-code test values in the function to make sure that the function is handling events correctly.

5. If you see invocations with no errors, but your devices are not able to connect (or publish, subscribe, and receive messages), the issue might be that the policy that your Lambda function returns doesn't

give permissions for the actions that your devices are trying to take. Perform the following steps to determine whether anything is wrong with the policy that the function returns.

- a. Use an Amazon CloudWatch Logs Insights query to scan logs over a short period of time to check for failures. The following example query sorts events by timestamp and looks for failures.

```
display clientId, eventType, status, @timestamp | sort @timestamp desc | filter status = "Failure"
```

- b. Update your Lambda function to log the data that it's returning to AWS IoT Core and the event that triggers the function. You can use these logs to inspect the policy that the function creates.

Investigating device issues

If you find no issues with invoking your Lambda function or with the policy that the function returns, look for problems with your devices' connection attempts. Malformed connection requests can cause AWS IoT Core not to trigger your authorizer. Connection problems can occur at both the TLS and application layers.

Possible TLS layer issues:

- Customers must pass either a hostname header (HTTP, MQTT over WebSockets) or the Server Name Indication TLS extension (HTTP, MQTT over WebSockets, MQTT) in all custom authentication requests. In both cases, the value passed must match one of your account's AWS IoT Core data endpoints. These are the endpoints that are returned when you perform the following CLI commands.
 - `aws iot describe-endpoint --endpoint-type iot:data-ats`
 - `aws iot describe-endpoint --endpoint-type (for legacy VeriSign endpoints)`
- Devices that use custom authentication for MQTT connections must also pass the Application Layer Protocol Negotiation (ALPN) TLS extension with a value of `mqtt`.
- Custom authentication is currently available only on port 443.

Possible application layer issues:

- If signing is enabled (the `signingDisabled` field is false in your authorizer), look for the following signature issues.
 - Make sure that you're passing the token signature in either the `x-amz-customauthorizer-signatureheader` or in a query string parameter.
 - Make sure that the service isn't signing a value other than the token.
 - Make sure that you pass the token in the header or query parameter that you specified in the `token-key-name` field in your authorizer.
- Make sure that the authorizer name you pass in the `x-amz-customauthorizer-name` header or query string parameter is valid or that you have a default authorizer defined for your account.

Authorization

Authorization is the process of granting permissions to an authenticated identity. You grant permissions in AWS IoT Core using AWS IoT Core and IAM policies. This topic covers AWS IoT Core policies. For more information about IAM policies, see [Identity and access management for AWS IoT \(p. 364\)](#) and [IAM policies \(p. 368\)](#).

AWS IoT Core policies determine what an authenticated identity can do. An authenticated identity is used by devices, mobile applications, web applications, and desktop applications. An authenticated

identity can even be a user typing AWS IoT Core CLI commands. An identity can execute AWS IoT Core operations only if it has a policy that grants it permission for those operations.

Both AWS IoT Core policies and IAM policies are used with AWS IoT Core to control the operations an identity (also called a *principal*) can perform. The policy type you use depends on the type of identity you are using to authenticate with AWS IoT Core.

AWS IoT Core operations are divided into two groups:

- Control plane API allows you to perform administrative tasks like creating or updating certificates, things, rules, and so on.
- Data plane API allows you send data to and receive data from AWS IoT Core.

The type of policy you use depends on whether you are using control plane or data plane API.

The following table shows the identity types, the protocols they use, and the policy types that can be used for authorization.

AWS IoT Core data plane API and policy types

Protocol and authentication mechanism	SDK	Identity type	Policy type		
MQTT over TLS/TCP, TLS mutual authentication (port 8883 or 443) ^{† (p. 78)}	AWS IoT Device SDK	X.509 certificates	AWS IoT Core policy		
MQTT over HTTPS/ WebSocket, AWS SigV4 authentication (port 443)	AWS Mobile SDK	Authenticated Amazon Cognito identity	IAM and AWS IoT Core policies		
		Unauthenticated Amazon Cognito identity	IAM policy		
		IAM, or federated identity	IAM policy		
HTTPS, AWS Signature Version 4 authentication (port 443)	AWS CLI	Amazon Cognito, IAM, or federated identity	IAM policy		
HTTPS, TLS mutual authentication (port 8443)	No SDK support	X.509 certificates	AWS IoT Core policy		
HTTPS over custom	AWS IoT Device SDK	Custom authorizer	Custom authorizer policy		

Protocol and authentication mechanism	SDK	Identity type	Policy type		
authentication (Port 443)					

AWS IoT Core control plane API and policy types

Protocol and authentication mechanism	SDK	Identity type	Policy type		
HTTPS AWS Signature Version 4 authentication (port 443)	AWS CLI	Amazon Cognito identity	IAM policy		
		IAM, or federated identity	IAM policy		

AWS IoT Core policies are attached to X.509 certificates or Amazon Cognito identities. IAM policies are attached to an IAM user, group, or role. If you use the AWS IoT console or the AWS IoT Core CLI to attach the policy (to a certificate or Amazon Cognito Identity), you use an AWS IoT Core policy. Otherwise, you use an IAM policy.

Policy-based authorization is a powerful tool. It gives you complete control over what a device, user, or application can do in AWS IoT Core. For example, consider a device connecting to AWS IoT Core with a certificate. You can allow the device to access all MQTT topics, or you can restrict its access to a single topic. In another example, consider a user typing CLI commands at the command line. By using a policy, you can allow or deny access to any command or AWS IoT Core resource for the user. You can also control an application's access to AWS IoT Core resources.

Changes made to a policy can take a few minutes to become effective because of how AWS IoT caches the policy documents. That is, it may take a few minutes to access a resource that has recently been granted access, and a resource may be accessible for several minutes after its access has been revoked.

AWS training and certification

For information about authorization in AWS IoT Core, take the [Deep Dive into AWS IoT Core Authentication and Authorization](#) course on the AWS Training and Certification website.

AWS IoT Core policies

AWS IoT Core policies are JSON documents. They follow the same conventions as IAM policies. AWS IoT Core supports named policies so many identities can reference the same policy document. Named policies are versioned so they can be easily rolled back.

AWS IoT Core policies allow you to control access to the AWS IoT Core data plane. The AWS IoT Core data plane consists of operations that allow you to connect to the AWS IoT Core message broker, send and receive MQTT messages, and get or update a thing's Device Shadow.

An AWS IoT Core policy is a JSON document that contains one or more policy statements. Each statement contains:

- **Effect**, which specifies whether the action is allowed or denied.

- Action, which specifies the action the policy is allowing or denying.
- Resource, which specifies the resource or resources on which the action is allowed or denied.

Changes made to a policy can take a few minutes to become effective because of how AWS IoT caches the policy documents. That is, it may take a few minutes to access a resource that has recently been granted access, and a resource may be accessible for several minutes after its access has been revoked.

Topics

- [AWS IoT Core policy actions \(p. 315\)](#)
- [AWS IoT Core action resources \(p. 317\)](#)
- [AWS IoT Core policy variables \(p. 318\)](#)
- [Cross-service confused deputy prevention \(p. 323\)](#)
- [AWS IoT Core policy examples \(p. 324\)](#)
- [Authorization with Amazon Cognito identities \(p. 353\)](#)

AWS IoT Core policy actions

The following policy actions are defined by AWS IoT Core:

MQTT Policy Actions

`iot:Connect`

Represents the permission to connect to the AWS IoT Core message broker. The `iot:Connect` permission is checked every time a CONNECT request is sent to the broker. The message broker doesn't allow two clients with the same client ID to stay connected at the same time. After the second client connects, the broker closes the existing connection. Use the `iot:Connect` permission to ensure only authorized clients using a specific client ID can connect.

`iot:GetRetainedMessage`

Represents the permission to get the contents of a single retained message. Retained messages are the messages that were published with the RETAIN flag set and stored by AWS IoT Core. For permission to get a list of all the account's retained messages, see [iot>ListRetainedMessages \(p. 315\)](#).

`iot>ListRetainedMessages`

Represents the permission to retrieve summary information about the account's retained messages, but not the contents of the messages. Retained messages are the messages that were published with the RETAIN flag set and stored by AWS IoT Core. The resource ARN specified for this action must be *. For permission to get the contents of a single retained message, see [iot:GetRetainedMessage \(p. 315\)](#).

`iot:Publish`

Represents the permission to publish an MQTT topic. This permission is checked every time a PUBLISH request is sent to the broker. You can use this to allow clients to publish to specific topic patterns.

Note

To grant `iot:Publish` permission, you must also grant `iot:Connect` permission.

`iot:Receive`

Represents the permission to receive a message from AWS IoT Core. The `iot:Receive` permission is confirmed every time a message is delivered to a client. Because this permission is checked on every delivery, you can use it to revoke permissions to clients that are currently subscribed to a topic.

iot:RetainPublish

Represents the permission to publish an MQTT message with the RETAIN flag set.

Note

To grant iot:RetainPublish permission, you must also grant iot:Publish permission.

iot:Subscribe

Represents the permission to subscribe to a topic filter. This permission is checked every time a SUBSCRIBE request is sent to the broker. Use it to allow clients to subscribe to topics that match specific topic patterns.

Note

To grant iot:Subscribe permission, you must also grant iot:Connect permission.

Device Shadow Policy Actions

iot:DeleteThingShadow

Represents the permission to delete a thing's Device Shadow. The iot:DeleteThingShadow permission is checked every time a request is made to delete a thing's Device Shadow contents.

iot:GetThingShadow

Represents the permission to retrieve a thing's Device Shadow. The iot:GetThingShadow permission is checked every time a request is made to retrieve a thing's Device Shadow contents.

iot>ListNamedShadowsForThing

Represents the permission to list a thing's named Shadows. The iot>ListNamedShadowsForThing permission is checked every time a request is made to list a thing's named Shadows.

iot:UpdateThingShadow

Represents the permission to update a device's shadow. The iot:UpdateThingShadow permission is checked every time a request is made to update a thing's Device Shadow contents.

Note

The job execution policy actions apply only for the HTTP TLS endpoint. If you use the MQTT endpoint, you must use MQTT policy actions defined in this topic.

For an example of a job execution policy that demonstrates this, see [the section called "Basic job policy example" \(p. 352\)](#) that works with the MQTT protocol.

Job Executions AWS IoT Core Policy Actions

iot:DescribeJobExecution

Represents the permission to retrieve a job execution for a given thing. The iot:DescribeJobExecution permission is checked every time a request is made to get a job execution.

iot:GetPendingJobExecutions

Represents the permission to retrieve the list of jobs that are not in a terminal status for a thing. The iot:GetPendingJobExecutions permission is checked every time a request is made to retrieve the list.

iot:UpdateJobExecution

Represents the permission to update a job execution. The iot:UpdateJobExecution permission is checked every time a request is made to update the state of a job execution.

`iot:StartNextPendingJobExecution`

Represents the permission to get and start the next pending job execution for a thing. (That is, to update a job execution with status QUEUED to IN_PROGRESS.) The `iot:StartNextPendingJobExecution` permission is checked every time a request is made to start the next pending job execution.

AWS IoT Core action resources

To specify a resource for an AWS IoT Core policy action, you must use the ARN of the resource. All resource ARNs are of the following form:

`arn:aws:iot:region:AWS-account-ID:Resource-type/Resource-name`

The following table shows the resource to specify for each action type:

Action	Resource type	Resource name	ARN example
<code>iot:Connect</code>	client	The client's client ID	<code>arn:aws:iot:us-east-1:123456789012:client/myClientId</code>
<code>iot:DeleteThingShadow</code>	thing	The thing's name	<code>arn:aws:iot:us-east-1:123456789012:thing/thingOne</code>
<code>iot:DescribeJobExecution</code>	thing	The thing's name	<code>arn:aws:iot:us-east-1:123456789012:thing/thingOne`</code>
<code>iot:GetPendingJobExecutions</code>	thing	The thing's name	<code>arn:aws:iot:us-east-1:123456789012:thing/thingOne`</code>
<code>iot:GetRetainedMessage</code>	topic	A retained message topic.	<code>arn:aws:iot:us-east-1:123456789012:topic/myTopicName</code>
<code>iot:GetThingShadow</code>	thing	The thing's name	<code>arn:aws:iot:us-east-1:123456789012:thing/thingOne</code>
<code>iot>ListRetainedMessages</code>	All		*
<code>iot:Publish</code>	topic	A topic string	<code>arn:aws:iot:us-east-1:123456789012:topic/myTopicName</code>
<code>iot:Receive</code>	topic	A topic string	<code>arn:aws:iot:us-east-1:123456789012:topic/myTopicName</code>
<code>iot:RetainPublish</code>	topic	A topic to publish with the RETAIN flag set.	<code>arn:aws:iot:us-east-1:123456789012:topic/myTopicName</code>

Action	Resource type	Resource name	ARN example
iot:StartNextPendingExecution	thing	The thing's name	arn:aws:iot:us-east-1:123456789012:thing/thingOne
iot:Subscribe	topicfilter	A topic filter string	arn:aws:iot:us-east-1:123456789012:topicfilter/myTopicFilter
iot:UpdateJobExecution	thing	The thing's name	arn:aws:iot:us-east-1:123456789012:thing/thingOne
iot:UpdateThingShadow	thing	The thing's name, and the shadow's name, if applicable	arn:aws:iot:us-east-1:123456789012:thing/thingOne arn:aws:iot:us-east-1:123456789012:thing/thingOne/shadowOne

AWS IoT Core policy variables

AWS IoT Core defines policy variables that can be used in AWS IoT Core policies in the Resource or Condition block. When a policy is evaluated, the policy variables are replaced by actual values. For example, if a device is connected to the AWS IoT Core message broker with a client ID of 100-234-3456, the `iot:ClientId` policy variable is replaced in the policy document by 100-234-3456. For more information about policy variables, see [IAM Policy Variables](#) and [Multi-Value Conditions](#).

Basic AWS IoT Core policy variables

AWS IoT Core defines the following basic policy variables:

- `iot:ClientId`: The client ID used to connect to the AWS IoT Core message broker.
- `aws:SourceIp`: The IP address of the client connected to the AWS IoT Core message broker.

The following AWS IoT Core policy shows a policy that uses policy variables:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": ["iot:Connect"],
            "Resource": [
                "arn:aws:iot:us-east-1:123451234510:client/${iot:ClientId}"
            ]
        },
        {
            "Effect": "Allow",
            "Action": ["iot:Publish"],
            "Resource": [
                "arn:aws:iot:us-east-1:123451234510:topic/my/topic/${iot:ClientId}"
            ]
        }
    ]
}
```

In these examples, \${iot:ClientId} is replaced by the ID of the client connected to the AWS IoT Core message broker when the policy is evaluated. When you use policy variables like \${iot:ClientId}, you can inadvertently open access to unintended topics. For example, if you use a policy that uses \${iot:ClientId} to specify a topic filter:

```
{  
    "Effect": "Allow",  
    "Action": ["iot:Subscribe"],  
    "Resource": [  
        "arn:aws:iot:us-east-1:123456789012:topicfilter/my/${iot:ClientId}/topic"  
    ]  
}
```

A client can connect using + as the client ID. This would allow the user to subscribe to any topic that matches the topic filter my/+ topic. To protect against such security gaps, use the iot:Connect policy action to control which client IDs can connect. For example, this policy allows only those clients whose client ID is clientid1 to connect:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": ["iot:Connect"],  
            "Resource": [  
                "arn:aws:iot:us-east-1:123456789012:client/clientid1"  
            ]  
        }  
    ]  
}
```

Thing policy variables

Thing policy variables allow you to write AWS IoT Core policies that grant or deny permissions based on thing properties like thing names, thing types, and thing attribute values. You can use thing policy variables to apply the same policy to control many AWS IoT Core devices. For more information about device provisioning, see [Device Provisioning](#). The thing name is obtained from the client ID in the MQTT Connect message sent when a thing connects to AWS IoT Core.

Keep the following in mind when using thing policy variables in AWS IoT Core policies.

- Use the [AttachThingPrincipal](#) API to attach certificates or principals (authenticated Amazon Cognito identities) to a thing.
- When you're replacing thing names with thing policy variables, the value of clientId in the MQTT connect message or the TLS connection must exactly match the thing name.

The following thing policy variables are available:

- `iot:Connection.Thing.ThingName`

This resolves to the name of the thing in the AWS IoT Core registry for which the policy is being evaluated. AWS IoT Core uses the certificate the device presents when it authenticates to determine which thing to use to verify the connection. This policy variable is only available when a device connects over MQTT or MQTT over the WebSocket protocol.

- `iot:Connection.Thing.ThingTypeName`

This resolves to the thing type associated with the thing for which the policy is being evaluated. The thing name is set to the client ID of the MQTT/WebSocket connection. This policy variable is available only when connecting over MQTT or MQTT over the WebSocket protocol.

- `iot:Connection.Thing.Attributes[attributeName]`

This resolves to the value of the specified attribute associated with the thing for which the policy is being evaluated. A thing can have up to 50 attributes. Each attribute is available as a policy variable: `iot:Connection.Thing.Attributes[attributeName]` where `attributeName` is the name of the attribute. The thing name is set to the client ID of the MQTT/WebSocket connection. This policy variable is only available when connecting over MQTT or MQTT over the WebSocket protocol.

- `iot:Connection.Thing.IsAttached`

`iot:Connection.Thing.IsAttached: ["true"]` enforces that only the devices that are both registered in AWS IoT and attached to principal can access the permissions inside the policy. You can use this variable to prevent a device from connecting to AWS IoT Core if it presents a certificate that is not attached to an IoT thing in the AWS IoT Core registry. This variable has values `true` or `false` indicating that the connecting thing is attached to the certificate or Amazon Cognito identity in the registry using [AttachThingPrincipal API](#). Thing name is taken as client Id.

X.509 Certificate AWS IoT Core policy variables

X.509 certificate policy variables allow you to write AWS IoT Core policies that grant permissions based on X.509 certificate attributes. The following sections describe how you can use these certificate policy variables.

CertificateId

In the [RegisterCertificate API](#), the `certificateId` appears in the response body. To get information about your certificate, you can use the `certificateId` in [DescribeCertificate](#).

Issuer attributes

The following AWS IoT Core policy variables allow you to allow or deny permissions based on certificate attributes set by the certificate issuer.

- `iot:Certificate.Issuer.DistinguishedNameQualifier`
- `iot:Certificate.Issuer.Country`
- `iot:Certificate.Issuer.Organization`
- `iot:Certificate.Issuer.OrganizationalUnit`
- `iot:Certificate.Issuer.State`
- `iot:Certificate.Issuer.CommonName`
- `iot:Certificate.Issuer.SerialNumber`
- `iot:Certificate.Issuer.Title`
- `iot:Certificate.Issuer.Surname`
- `iot:Certificate.Issuer.GivenName`
- `iot:Certificate.Issuer.Initials`
- `iot:Certificate.Issuer.Pseudonym`
- `iot:Certificate.Issuer.GenerationQualifier`

Subject attributes

The following AWS IoT Core policy variables allow you to grant or deny permissions based on certificate subject attributes set by the certificate issuer.

- `iot:Certificate.Subject.DistinguishedNameQualifier`
- `iot:Certificate.Subject.Country`
- `iot:Certificate.Subject.Organization`
- `iot:Certificate.Subject.OrganizationalUnit`
- `iot:Certificate.Subject.State`
- `iot:Certificate.Subject.CommonName`
- `iot:Certificate.Subject.SerialNumber`
- `iot:Certificate.Subject.Title`
- `iot:Certificate.Subject.Surname`
- `iot:Certificate.Subject.GivenName`
- `iot:Certificate.Subject.Initials`
- `iot:Certificate.Subject.Pseudonym`
- `iot:Certificate.Subject.GenerationQualifier`

X.509 certificates allow these attributes to contain one or more values. By default, the policy variables for each multi-value attribute return the first value. For example, the `Certificate.Subject.Country` attribute might contain a list of country names, but when evaluated in a policy, `iot:Certificate.Subject.Country` is replaced by the first country name. You can request a specific attribute value other than the first value by using a one-based index. For example, `iot:Certificate.Subject.Country.1` is replaced by the second country name in the `Certificate.Subject.Country` attribute. If you specify an index value that does not exist (for example, if you ask for a third value when there are only two values assigned to the attribute), no substitution is made and authorization fails. You can use the `.List` suffix on the policy variable name to specify all values of the attribute.

Registered devices (2)

For devices registered as things in the AWS IoT Core registry, the following policy allows clients with a thing name registered in the AWS IoT Core registry to connect, but restricts the right to publish to a thing name specific topic to those clients with certificates whose `Certificate.Subject.Organization` attribute is set to "Example Corp" or "AnyCompany". This restriction is accomplished by using a "Condition" field that specifies a condition that must be met to allow the preceding action. In this case the condition is that the `Certificate.Subject.Organization` attribute associated with the certificate must include one of the values listed:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/${iot:Connection.Thing.ThingName}"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/my/topic/
${iot:Connection.Thing.ThingName}"
            ]
        }
    ]
}
```

```

        ],
        "Condition": {
            "ForAllValues:StringEquals": {
                "iot:Certificate.Subject.Organization.List": [
                    "Example Corp",
                    "AnyCompany"
                ]
            }
        }
    ]
}

```

Unregistered devices (2)

For devices not registered as things in the AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core with client IDs `client1`, `client2`, and `client3`, but restricts the right to publish to a client-id specific topic to those clients with certificates whose `Certificate.Subject.Organization` attribute is set to "`Example Corp`" or "`AnyCompany`". This restriction is accomplished by using a "`Condition`" field that specifies a condition that must be met to allow the preceding action. In this case the condition is that the `Certificate.Subject.Organization` attribute associated with the certificate must include one of the values listed:

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/client1",
                "arn:aws:iot:us-east-1:123456789012:client/client2",
                "arn:aws:iot:us-east-1:123456789012:client/client3"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/my/topic/${iot:ClientId}"
            ],
            "Condition": {
                "ForAllValues:StringEquals": {
                    "iot:Certificate.Subject.Organization.List": [
                        "Example Corp",
                        "AnyCompany"
                    ]
                }
            }
        }
    ]
}

```

Issuer alternate name attributes

The following AWS IoT Core policy variables allow you to grant or deny permissions based on issuer alternate name attributes set by the certificate issuer.

- `iot:Certificate.Issuer.AlternativeName.RFC822Name`
- `iot:Certificate.Issuer.AlternativeName.DNSName`
- `iot:Certificate.Issuer.AlternativeName.DirectoryName`
- `iot:Certificate.Issuer.AlternativeName.UniformResourceIdentifier`
- `iot:Certificate.Issuer.AlternativeName.IPAddress`

Subject alternate name attributes

The following AWS IoT Core policy variables allow you to grant or deny permissions based on subject alternate name attributes set by the certificate issuer.

- `iot:Certificate.Subject.AlternativeName.RFC822Name`
- `iot:Certificate.Subject.AlternativeName.DNSName`
- `iot:Certificate.Subject.AlternativeName.DirectoryName`
- `iot:Certificate.Subject.AlternativeName.UniformResourceIdentifier`
- `iot:Certificate.Subject.AlternativeName.IPAddress`

Other attributes

You can use `iot:Certificate.SerialNumber` to allow or deny access to AWS IoT Core resources based on the serial number of a certificate. The `iot:Certificate.AvailableKeys` policy variable contains the name of all certificate policy variables that contain values.

X.509 Certificate policy variable limitations

The following limitations apply to X.509 certificate policy variables:

Wildcards

If wildcard characters are present in certificate attributes, the policy variable is not replaced by the certificate attribute value, leaving the `${policy-variable}` text in the policy document. This might cause authorization failure. The following wildcard characters can be used: *, \$, +, ?, and #.

Array fields

Certificate attributes that contain arrays are limited to five items. Additional items are ignored.

String length

All string values are limited to 1024 characters. If a certificate attribute contains a string longer than 1024 characters, the policy variable is not replaced by the certificate attribute value, leaving the `${policy-variable}` in the policy document. This might cause authorization failure.

Special Characters

Any special character, such as , ", \, +, =, <, > and ; must be prefixed with a backslash (\) when used in a policy variable. For example, Amazon Web Services O=Amazon.com Inc. L=Seattle ST=Washington C=US becomes Amazon Web Service O\=Amazon.com Inc. L\=Seattle ST\=Washington C\=US.

Cross-service confused deputy prevention

The *confused deputy problem* is a security issue where an entity that doesn't have permission to perform an action can coerce a more-privileged entity to perform the action. In AWS, cross-service impersonation can result in the confused deputy problem. Cross-service impersonation can occur when one service (the *calling service*) calls another service (the *called service*). The calling service can be manipulated to use its

permissions to act on another customer's resources in a way it shouldn't otherwise have permission to access. To prevent this, AWS provides tools that help you protect your data for all services with service principals that have been given access to resources in your account.

To limit the permissions that AWS IoT gives another service to the resource, we recommend using the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in resource policies. If you use both global condition context keys, the `aws:SourceAccount` value and the account in the `aws:SourceArn` value must use the same account ID when used in the same policy statement.

The most effective way to protect against the confused deputy problem is to use the `aws:SourceArn` global condition context key with the full Amazon Resource Name (ARN) of the resource. For AWS IoT, your `aws:SourceArn` must comply with the format: `arn:aws:iot:region:account-id/*`. Make sure that the `region` matches your AWS IoT Region and the `account-id` matches your customer account ID.

The following example shows how to prevent the confused deputy problem by using the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in the AWS IoT role trust policy.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Principal": {  
                "Service": "iot.amazonaws.com"  
            },  
            "Action": "sts:AssumeRole",  
            "Condition": {  
                "StringEquals": {  
                    "aws:SourceAccount": "123456789012"  
                },  
                "ArnLike": {  
                    "aws:SourceArn": "arn:aws:iot:us-east-1:123456789012:/*"  
                }  
            }  
        }  
    ]  
}
```

AWS IoT Core policy examples

The example policies in this section illustrate the policy documents used to complete common tasks in AWS IoT Core. You can use them as examples to start from when creating the policies for your solutions.

The examples in this section use these policy elements:

- the section called “AWS IoT Core policy actions” (p. 315)
- the section called “AWS IoT Core action resources” (p. 317)
- the section called “Identity-based policy examples” (p. 385)
- the section called “Basic AWS IoT Core policy variables” (p. 318)
- the section called “X.509 Certificate AWS IoT Core policy variables” (p. 320)

Policy examples in this section:

- Connect policy examples (p. 325)
- Publish/Subscribe policy examples (p. 332)
- Connect and publish policy examples (p. 345)

- [Retained message policy examples \(p. 346\)](#)
- [Certificate policy examples \(p. 347\)](#)
- [Thing policy examples \(p. 352\)](#)
- [Basic job policy example \(p. 352\)](#)

Connect policy examples

The following policy grants permission to connect to AWS IoT Core with client ID `client1`:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:Connect"  
            ],  
            "Resource": [  
                "arn:aws:iot:us-east-1:123456789012:client/client1"  
            ]  
        }  
    ]  
}
```

The following policy denies permission to client IDs `client1` and `client2` to connect to AWS IoT Core, while allowing devices to connect using a client ID that matches the name of a thing registered in the AWS IoT Core registry:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Deny",  
            "Action": [  
                "iot:Connect"  
            ],  
            "Resource": [  
                "arn:aws:iot:us-east-1:123456789012:client/client1",  
                "arn:aws:iot:us-east-1:123456789012:client/client2"  
            ]  
        },  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:Connect"  
            ],  
            "Resource": [  
                "arn:aws:iot:us-east-1:123456789012:client/${iot:Connection.Thing.ThingName}"  
            ]  
        }  
    ]  
}
```

MQTT persistent sessions policy examples

`connectAttributes` allow you to specify what attributes you want to use in your connect message in your IAM policies such as `PersistentConnect` and `LastWill`. For more information, see [Using connectAttributes \(p. 89\)](#)

The following policy allows connect with `PersistentConnect` feature:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": "*",
            "Condition": {
                "ForAllValues:StringEquals": {
                    "iot:ConnectAttributes": [
                        "PersistentConnect"
                    ]
                }
            }
        }
    ]
}
```

The following policy disallows PersistentConnect, other features are allowed:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": "*",
            "Condition": {
                "ForAllValues:StringNotEquals": {
                    "iot:ConnectAttributes": [
                        "PersistentConnect"
                    ]
                }
            }
        }
    ]
}
```

The above policy can also be expressed using `StringEquals`, any other feature including new feature is allowed:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Deny",
            "Action": [
                "iot:Connect"
            ],
            "Resource": "*",
            "Condition": {
                "ForAnyValue:StringEquals": {
                    "iot:ConnectAttributes": [

```

```

        "PersistentConnect"
    ]
}
}
]
}
```

The following policy allows connect by both `PersistentConnect` and `LastWill`, any other new feature is not allowed:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": "*",
            "Condition": {
                "ForAllValues:StringEquals": {
                    "iot:ConnectAttributes": [
                        "PersistentConnect",
                        "LastWill"
                    ]
                }
            }
        }
    ]
}
```

The following policy allows clean connect by clients with or without `LastWill`, no other features will be allowed:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": "*",
            "Condition": {
                "ForAllValues:StringEquals": {
                    "iot:ConnectAttributes": [
                        "LastWill"
                    ]
                }
            }
        }
    ]
}
```

The following policy only allows connect using default features:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",

```

```

    "Action": [
        "iot:Connect"
    ],
    "Resource": "*",
    "Condition": {
        "ForAllValues:StringEquals": {
            "iot:ConnectAttributes": [
            ]
        }
    }
]
}

```

The following policy allows connect only with `PersistentConnect`, any new feature is allowed as long as the connection uses `PersistentConnect`:

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": "*",
            "Condition": {
                "ForAnyValue:StringEquals": {
                    "iot:ConnectAttributes": [
                        "PersistentConnect"
                    ]
                }
            }
        }
    ]
}

```

The following policy states the connect must have both `PersistentConnect` and `LastWill` usage, no new feature is allowed:

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": "*",
            "Condition": {
                "ForAllValues:StringEquals": {
                    "iot:ConnectAttributes": [
                        "PersistentConnect",
                        "LastWill"
                    ]
                }
            }
        },
        {
            "Effect": "Deny",
            "Action": [
                "iot:Connect"
            ],
            "Resource": "*"
        }
    ]
}

```

```

    "Resource": "*",
    "Condition": {
        "ForAllValues:StringEquals": {
            "iot:ConnectAttributes": [
                "PersistentConnect"
            ]
        }
    }
},
{
    "Effect": "Deny",
    "Action": [
        "iot:Connect"
    ],
    "Resource": "*",
    "Condition": {
        "ForAllValues:StringEquals": {
            "iot:ConnectAttributes": [
                "LastWill"
            ]
        }
    }
},
{
    "Effect": "Deny",
    "Action": [
        "iot:Connect"
    ],
    "Resource": "*",
    "Condition": {
        "ForAllValues:StringEquals": {
            "iot:ConnectAttributes": [
            ]
        }
    }
}
]
}

```

The following policy must not have `PersistentConnect` but can have `LastWill`, any other new feature is not allowed:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "iot:Connect"
            ],
            "Resource": "*",
            "Condition": {
                "ForAnyValue:StringEquals": {
                    "iot:ConnectAttributes": [
                        "PersistentConnect"
                    ]
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],

```

```

    "Resource": "*",
    "Condition": {
        "ForAllValues:StringEquals": {
            "iot:ConnectAttributes": [
                "LastWill"
            ]
        }
    }
}
]
}
}

```

The following policy allows connect only by clients that have a `LastWill` with topic "my/lastwill/topicName", any feature is allowed as long as it uses the `LastWill` topic:

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": "*",
            "Condition": {
                "ArnEquals": {
                    "iot:LastWillTopic": "arn:aws:iot:*region*:account-id*:topic/*my/
lastwill/topicName*"
                }
            }
        }
    ]
}

```

The following policy only allows clean connect using a specific `LastWillTopic`, any feature is allowed as long as it uses the `LastWillTopic`:

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": "*",
            "Condition": {
                "ArnEquals": {
                    "iot:LastWillTopic": "arn:aws:iot:*region*:account-id*:topic/*my/
lastwill/topicName*"
                }
            }
        },
        {
            "Effect": "Deny",
            "Action": [
                "iot:Connect"
            ],
            "Resource": "*",
            "Condition": {
                "ForAnyValue:StringEquals": {
                    "iot:ConnectAttributes": [
                        "PersistentConnect"
                    ]
                }
            }
        }
    ]
}

```

```
        ]
    }
}
]
```

Registered devices (3)

The following policy grants permission for a device to connect using its thing name as the client ID and to subscribe to the topic filter `my/topic/filter`. The device must be registered with AWS IoT Core. When the device connects to AWS IoT Core, it must provide the certificate associated with the IoT thing in the AWS IoT Core registry:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:Connect"
      ],
      "Resource": [
        "arn:aws:iot:us-east-1:123456789012:client/${iot:Connection.Thing.ThingName}"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iot:Subscribe"
      ],
      "Resource": [
        "arn:aws:iot:us-east-1:123456789012:topicfilter/my/topic/filter"
      ]
    }
  ]
}
```

Unregistered devices (3)

For devices not registered as things in the AWS IoT Core registry, the following policy grants permission to connect using client ID `client1` and to subscribe to topic filter `my/topic`:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:Connect"
      ],
      "Resource": [
        "arn:aws:iot:us-east-1:123456789012:client/client1"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iot:Subscribe"
      ],
      "Resource": [
        "arn:aws:iot:us-east-1:123456789012:topicfilter/my/topic"
      ]
    }
  ]
}
```

```
    ]
}
```

Publish/Subscribe policy examples

The policy you use depends on how you're connecting to AWS IoT Core. You can connect to AWS IoT Core by using an MQTT client, HTTP, or WebSocket. When you connect with an MQTT client, you're authenticating with an X.509 certificate. When you connect over HTTP or the WebSocket protocol, you're authenticating with Signature Version 4 and Amazon Cognito.

Policies for MQTT clients

MQTT and AWS IoT Core policies have different wildcard characters and they should be used with careful consideration. In MQTT, the wildcard characters + and # are used in [MQTT topic filters](#) to subscribe to multiple topic names. AWS IoT Core policies follow the same conventions as [IAM policies](#) and use * as a wildcard character, and the MQTT wildcard characters + and # are treated as literal strings. Therefore, to specify wildcard characters in topic names and topic filters in AWS IoT Core policies for MQTT clients, you must use *.

The table below shows the different wildcard characters used in MQTT and AWS IoT Core policies for MQTT clients.

Wildcard character	Is MQTT wildcard character	Example in MQTT	Is AWS IoT Core policy wildcard character	Example in AWS IoT Core policies for MQTT clients
#	Yes	some/#	No	N/A
+	Yes	some/+/topic	No	N/A
*	No	N/A	Yes	topicfilter/some/*/topic

To describe multiple topic names in the `resource` attribute of a policy, use the * wildcard character. The following policy enables a device to publish to all subtopics that start with the same thing name.

Registered devices (5)

For devices registered as things in the AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core using a client ID that matches the thing name and to publish to any topic prefixed by the thing name:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:Connect"
      ],
      "Resource": [
        "arn:aws:iot:us-east-1:123456789012:client/${iot:Connection.Thing.ThingName}"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iot:Publish"
      ],
      "TopicPrefix": [
        "${iot:Connection.Thing.ThingName}/"
      ]
    }
  ]
}
```

```

        "Effect": "Allow",
        "Action": [
            "iot:Publish"
        ],
        "Resource": [
            "arn:aws:iot:us-east-1:123456789012:topic/
${iot:Connection.Thing.ThingName}/*"
        ]
    }
}

```

Unregistered devices (5)

For devices not registered as things in the AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core using client ID `client1`, `client2`, or `client3` and to publish to any topic prefixed by the client ID:

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/client1",
                "arn:aws:iot:us-east-1:123456789012:client/client2",
                "arn:aws:iot:us-east-1:123456789012:client/client3"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/${iot:ClientId}/*"
            ]
        }
    ]
}

```

You can also use the `*` wildcard character at the end of a topic filter. Using wildcard characters might lead to granting unintended privileges, so they should only be used after careful consideration. One situation in which they might be useful is when devices must subscribe to messages with many different topics (for example, if a device must subscribe to reports from temperature sensors in multiple locations).

Registered devices (6)

For devices registered as things in the AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core using the device's thing name as the client ID, and to subscribe to a topic prefixed by the thing name, followed by `room`, followed by any string. (It is expected that these topics are, for example, `thing1/room1`, `thing1/room2`, and so on):

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",

```

```

        "Action": [
            "iot:Connect"
        ],
        "Resource": [
            "arn:aws:iot:us-east-1:123456789012:client/
${iot:Connection.Thing.ThingName}"
        ]
    },
    {
        "Effect": "Allow",
        "Action": [
            "iot:Subscribe"
        ],
        "Resource": [
            "arn:aws:iot:us-east-1:123456789012:topicfilter/
${iot:Connection.Thing.ThingName}/room*"
        ]
    },
    {
        "Effect": "Allow",
        "Action": [
            "iot:Receive"
        ],
        "Resource": [
            "arn:aws:iot:us-east-1:123456789012:topic/
${iot:Connection.Thing.ThingName}/room*"
        ]
    }
]
}

```

Unregistered devices (6)

For devices not registered as things in the AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core using client IDs `client1`, `client2`, `client3`, and to subscribe to a topic prefixed by the client ID, followed by `room`, followed by any string. (It is expected that these topics are, for example, `client1/room1`, `client1/room2`, and so on):

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/client1",
                "arn:aws:iot:us-east-1:123456789012:client/client2",
                "arn:aws:iot:us-east-1:123456789012:client/client3"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topicfilter/${iot:ClientId}/room*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [

```

```

        "iot:Receive"
    ],
    "Resource": [
        "arn:aws:iot:us-east-1:123456789012:topic/${iot:ClientId}/room*"
    ]
}
]
}
}

```

When you specify topic filters in AWS IoT Core policies for MQTT clients, MQTT wildcard characters + and # are treated as literal strings. Their use might result in unexpected behavior.

Registered devices (4)

For devices registered as things in the AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core with the client ID that matches the thing name, and to subscribe to the topic filter some/+/*topic only. Note in topicfilter/some/+/*topic of the resource ARN, + is treated as a literal string in AWS IoT Core policies for MQTT clients, meaning that only the string some/+/*topic matches the topic filter.

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/
${iot:Connection.Thing.ThingName}"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topicfilter/some/+/*topic"
            ]
        }
    ]
}

```

For devices registered as things in the AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core with the client ID that matches the thing name, and to subscribe to the topic filter some/*/*topic. Note in topicfilter/some/*/*topic of the resource ARN, * is treated as a wildcard character in AWS IoT Core policies for MQTT clients, meaning that any string in the level that contains the character matches the topic filter. (It is expected that these topics are, for example, some/string1/topic, some/string2/topic, and so on)

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [

```

```

        "arn:aws:iot:us-east-1:123456789012:client/
${iot:Connection.Thing.ThingName}"
    ]
},
{
    "Effect": "Allow",
    "Action": [
        "iot:Subscribe"
    ],
    "Resource": [
        "arn:aws:iot:us-east-1:123456789012:topicfilter/some/*/*topic"
    ]
}
]
}

```

Unregistered devices (4)

For devices not registered as things in the AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core with client ID `client1` and subscribe to the topic filter `some/+/*topic` only. Note in `topicfilter/some/+/*topic` of the resource ARN, `+` is treated as a literal string in AWS IoT Core policies for MQTT clients, meaning that only the string `some/+/*topic` matches the topic filter.

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/client1"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topicfilter/some/+/*topic"
            ]
        }
    ]
}

```

For devices not registered as things in the AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core with client ID `client1` and subscribe to the topic filter `some/+/*topic`. Note in `topicfilter/some/*/*topic` of the resource ARN, `*` is treated as a wildcard character in AWS IoT Core policies for MQTT clients, meaning that any string in the level that contains the character matches the topic filter. (It is expected that these topics are, for example, `some/string1/topic`, `some/string2/topic`, and so on)

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/client1"
            ]
        }
    ]
}

```

```

        ],
        "Resource": [
            "arn:aws:iot:us-east-1:123456789012:client/client1"
        ]
    },
    {
        "Effect": "Allow",
        "Action": [
            "iot:Subscribe"
        ],
        "Resource": [
            "arn:aws:iot:us-east-1:123456789012:topicfilter/some/*/*topic"
        ]
    }
]
}

```

Note

The MQTT wildcards + and # are treated as literal strings in an AWS IoT Core policy for MQTT clients. To specify wildcard characters in topic names and topic filters in AWS IoT Core policies for MQTT clients, you must use *.

Registered devices (7)

For devices registered as things in the AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core using the device's thing name as the client ID, and to subscribe to the topics my/topic and my/othertopic:

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/
${iot:Connection.Thing.ThingName}"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topicfilter/my/topic",
                "arn:aws:iot:us-east-1:123456789012:topicfilter/my/othertopic"
            ]
        }
    ]
}

```

Unregistered devices (7)

For devices not registered as things in the AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core using client ID client1, and to subscribe to the topics my/topic and my/othertopic:

```
{

```

```

"Version": "2012-10-17",
"Statement": [
    {
        "Effect": "Allow",
        "Action": [
            "iot:Connect"
        ],
        "Resource": [
            "arn:aws:iot:us-east-1:123456789012:client/client1"
        ]
    },
    {
        "Effect": "Allow",
        "Action": [
            "iot:Subscribe"
        ],
        "Resource": [
            "arn:aws:iot:us-east-1:123456789012:topicfilter/my/topic",
            "arn:aws:iot:us-east-1:123456789012:topicfilter/my/othertopic"
        ]
    }
]
}

```

Registered devices (8)

For devices registered as things in the AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core using the device's thing name as the client ID and to subscribe to a topic unique to that thing name/client ID:

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/
${iot:Connection.Thing.ThingName}"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/my/topic/
${iot:Thing.ThingName}"
            ]
        }
    ]
}

```

Unregistered devices (8)

For devices not registered as things in the AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core using client ID `client1`, and to publish to a topic unique to that client ID:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/client1"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/my/topic/${iot:ClientId}"
            ]
        }
    ]
}
```

Registered devices (9)

For devices registered as things in the AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core using the device's thing name as the client ID and to publish to any topic prefixed by that thing name or client except for one topic ending with bar:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/
${iot:Connection.Thing.ThingName}"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/${iot:Thing.ThingName}/*"
            ]
        },
        {
            "Effect": "Deny",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/${iot:Thing.ThingName}/bar"
            ]
        }
    ]
}
```

}

Unregistered devices (9)

For devices not registered as things in the AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core using client IDs `client1` and `client1` and to publish to any topic prefixed by the client ID used to connect, except for one topic ending with `bar`:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/client1",
                "arn:aws:iot:us-east-1:123456789012:client/client2"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/${iot:ClientId}/*"
            ]
        },
        {
            "Effect": "Deny",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/${iot:ClientId}/bar"
            ]
        }
    ]
}
```

Registered devices (10)

For devices registered as things in the AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core using the device's thing name as the client ID. The device can subscribe to the topic `my/topic`, but cannot publish to the `thing-name /bar` where `thing-name` is the name of the IoT thing connecting to AWS IoT Core:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/
${iot:Connection.Thing.ThingName}"
            ]
        },
    ]
}
```

```
{
    "Effect": "Allow",
    "Action": [
        "iot:Subscribe"
    ],
    "Resource": [
        "arn:aws:iot:us-east-1:123456789012:topicfilter/my/topic"
    ]
},
{
    "Effect": "Deny",
    "Action": [
        "iot:Publish"
    ],
    "Resource": [
        "arn:aws:iot:us-east-1:123456789012:topic/${iot:Thing.ThingName}/bar"
    ]
}
]
```

Unregistered devices (10)

For devices not registered as things in the AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core using client ID `client1` and to subscribe to the topic `my/topic`:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/client1"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topicfilter/my/topic"
            ]
        }
    ]
}
```

Thing policy variables are also replaced when a certificate or authenticated Amazon Cognito Identity is attached to a thing. The following policy grants permission to connect to AWS IoT Core with client ID `client1` and to publish and receive topic `iotmonitor/provisioning/987654321098`. It also allows the certificate holder to subscribe to this topic.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [

```

```
        "iot:Connect"
    ],
    "Resource": [
        "arn:aws:iot:us-east-1:123456789012:client/client1"
    ]
},
{
    "Effect": "Allow",
    "Action": [
        "iot:Publish",
        "iot:Receive"
    ],
    "Resource": [
        "arn:aws:iot:us-east-1:123456789012:topic/iotmonitor/
provisioning/987654321098"
    ]
},
{
    "Effect": "Allow",
    "Action": [
        "iot:Subscribe"
    ],
    "Resource": [
        "arn:aws:iot:us-east-1:123456789012:topicfilter/iotmonitor/
provisioning/987654321098"
    ]
}
]
```

Policies for HTTP and WebSocket clients

Amazon Cognito identities can be authenticated or unauthenticated. Authenticated identities belong to users who are authenticated by any supported identity provider. Unauthenticated identities typically belong to guest users who do not authenticate with an identity provider. Amazon Cognito provides a unique identifier and AWS credentials to support unauthenticated identities.

For the following operations, AWS IoT Core uses AWS IoT Core policies attached to Amazon Cognito identities (through the `AttachPolicy` API) to scope down the permissions attached to the Amazon Cognito Identity pool with authenticated identities.

- `iot:Connect`
- `iot:Publish`
- `iot:Subscribe`
- `iot:Receive`
- `iot:GetThingShadow`
- `iot:UpdateThingShadow`
- `iot:DeleteThingShadow`

That means an Amazon Cognito Identity needs permission from the IAM role policy attached to the pool and the AWS IoT Core policy attached to the Amazon Cognito Identity through the AWS IoT Core `AttachPolicy` API.

Authenticated and unauthenticated users are different identity types. If you don't attach an AWS IoT policy to the Amazon Cognito Identity, an authenticated user fails authorization in AWS IoT and doesn't have access to AWS IoT resources and actions.

Note

For other AWS IoT Core operations or for unauthenticated identities, AWS IoT Core does not scope down the permissions attached to the Amazon Cognito identity pool role. For both

authenticated and unauthenticated identities, this is the most permissive policy that we recommend you attach to the Amazon Cognito pool role.

HTTP

To allow unauthenticated Amazon Cognito identities to publish messages over HTTP on a topic specific to the Amazon Cognito Identity, attach the following IAM policy to the Amazon Cognito Identity pool role:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:Publish"  
            ],  
            "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/${cognito-  
identity.amazonaws.com:sub}"]  
        }  
    ]  
}
```

To allow authenticated users, attach the preceding policy to the Amazon Cognito Identity pool role and to the Amazon Cognito Identity using the AWS IoT Core [AttachPolicy](#) API.

Note

When authorizing Amazon Cognito identities, AWS IoT Core considers both policies and grants the least privileges specified. An action is allowed only if both policies allow the requested action. If either policy disallows an action, that action is unauthorized.

MQTT

To allow unauthenticated Amazon Cognito identities to publish MQTT messages over WebSocket on a topic specific to the Amazon Cognito Identity in your account, attach the following IAM policy to the Amazon Cognito Identity pool role:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:Publish"  
            ],  
            "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/${cognito-  
identity.amazonaws.com:sub}"]  
        },  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:Connect"  
            ],  
            "Resource": ["arn:aws:iot:us-east-1:123456789012:client/${cognito-  
identity.amazonaws.com:sub}"]  
        }  
    ]  
}
```

To allow authenticated users, attach the preceding policy to the Amazon Cognito Identity pool role and to the Amazon Cognito Identity using the AWS IoT Core [AttachPolicy](#) API.

Note

When authorizing Amazon Cognito identities, AWS IoT Core considers both and grants the least privileges specified. An action is allowed only if both policies allow the requested action. If either policy disallows an action, that action is unauthorized.

[Receive policy examples](#)

Registered devices (11)

For devices registered in AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core with a client ID that matches the thing name and to subscribe to and receive messages on the `my/topic` topic:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:Connect"  
            ],  
            "Resource": ["arn:aws:iot:us-east-1:123456789012:client/  
${iot:Connection.Thing.ThingName}"]  
        },  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:Subscribe"  
            ],  
            "Resource": [  
                "arn:aws:iot:us-east-1:123456789012:topicfilter/my/topic"  
            ]  
        },  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:Receive"  
            ],  
            "Resource": [  
                "arn:aws:iot:us-east-1:123456789012:topic/my/topic"  
            ]  
        }  
    ]  
}
```

Unregistered devices (11)

For devices not registered in AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core with client ID `client1` and to subscribe to and receive messages on one topic:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:Connect"  
            ],  
            "Resource": ["arn:aws:iot:us-east-1:123456789012:client/client1"]  
        },  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:Subscribe"  
            ],  
            "TopicFilter": "my/topic"  
        }  
    ]  
}
```

```

        "Action": [
            "iot:Subscribe"
        ],
        "Resource": [
            "arn:aws:iot:us-east-1:123456789012:topicfilter/my/topic"
        ]
    },
    {
        "Effect": "Allow",
        "Action": [
            "iot:Receive"
        ],
        "Resource": [
            "arn:aws:iot:us-east-1:123456789012:topic/my/topic"
        ]
    }
]
}

```

Connect and publish policy examples

For devices registered as things in the AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core with a client ID that matches the thing name and restricts the device to publishing on a client-ID or thing name-specific MQTT topic. For a connection to be successful, the thing name must be registered in the AWS IoT Core registry and be authenticated using an identity or principal attached to the thing:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": ["iot:Publish"],
            "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/${iot:Connection.Thing.ThingName}"]
        },
        {
            "Effect": "Allow",
            "Action": ["iot:Connect"],
            "Resource": ["arn:aws:iot:us-east-1:123456789012:client/${iot:Connection.Thing.ThingName}"]
        }
    ]
}
```

For devices not registered as things in the AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core with client ID `client1` and restricts the device to publishing on a clientID-specific MQTT topic:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": ["iot:Publish"],
            "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/${iot:ClientId}"]
        },
        {
            "Effect": "Allow",
            "Action": ["iot:Connect"],
            "Resource": ["arn:aws:iot:us-east-1:123456789012:client/client1"]
        }
    ]
}
```

```
        ]  
    }  
}
```

Retained message policy examples

Using [retained messages \(p. 83\)](#) requires specific policies. Retained messages are MQTT messages published with the RETAIN flag set and stored by AWS IoT Core. This section presents examples of policies that allow common uses of retained messages.

In this section:

- [Policy to connect and publish retained messages \(p. 346\)](#)
- [Policy to connect and publish retained Will messages \(p. 346\)](#)
- [Policy to list and get retained messages \(p. 347\)](#)

Policy to connect and publish retained messages

For a device to publish retained messages, the device must be able to connect, publish (any MQTT message), and publish MQTT retained messages. The following policy grants these permissions for the topic: `device/sample/configuration` to client `device1`. For another example that grants permission to connect, see [the section called “Connect and publish policy examples” \(p. 345\)](#).

```
{  
  "Version": "2012-10-17",  
  "Statement": [  
    {  
      {  
        "Effect": "Allow",  
        "Action": [  
          "iot:Connect"  
        ],  
        "Resource": [  
          "arn:aws:iot:us-east-1:123456789012:client/device1"  
        ]  
      },  
      "Effect": "Allow",  
      "Action": [  
        "iot:Publish",  
        "iot:RetainPublish"  
      ],  
      "Resource": [  
        "arn:aws:iot:us-east-1:123456789012:topic/device/sample/configuration"  
      ]  
    }  
  ]  
}
```

Policy to connect and publish retained Will messages

Clients can configure a message that AWS IoT Core will publish when the client disconnects unexpectedly. MQTT calls such a message a [Will message](#). A client must have an additional condition added to its connect permission to include them.

The following policy document grants all clients permission to connect and publish a Will message, identified by its topic, `will`, that AWS IoT Core will also retain.

```
{  
  "Version": "2012-10-17",  
  "Statement": [  
    {  
      "Condition": "aws:multiFactorAuthenticationEnabled",  
      "Effect": "Allow",  
      "Action": [  
        "iot:Connect",  
        "iot:Publish"  
      ],  
      "Resource": "  
        arn:aws:iot:us-east-1:123456789012:topic/will/  
      "
```

```

    "Effect": "Allow",
    "Action": [
        "iot:Connect"
    ],
    "Resource": [
        "arn:aws:iot:us-east-1:123456789012:client/*"
    ],
    "Condition": {
        "ForAllValues:StringEquals": {
            "iot:ConnectAttributes": [
                "LastWill"
            ]
        }
    },
    {
        "Effect": "Allow",
        "Action": [
            "iot:Publish",
            "iot:RetainPublish"
        ],
        "Resource": [
            "arn:aws:iot:us-east-1:123456789012:topic/will"
        ]
    }
]
}

```

Policy to list and get retained messages

Services and applications can access retained messages without the need to support an MQTT client by calling [ListRetainedMessages](#) and [GetRetainedMessage](#). The services and applications that call these actions must be authorized by using a policy such as the following example.

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot>ListRetainedMessages"
            ],
            "Resource": [
                "*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:GetRetainedMessage"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/foo"
            ]
        }
    ]
}

```

Certificate policy examples

For devices registered in AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core with a client ID that matches a thing name, and to publish to a topic whose name is equal to the `certificateId` of the certificate the device used to authenticate itself:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/${iot:CertificateId}"]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": ["arn:aws:iot:us-east-1:123456789012:client/
${iot:Connection.Thing.ThingName}"]
        }
    ]
}
```

For devices not registered in the AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core with client IDs, `client1`, `client2`, and `client3` and to publish to a topic whose name is equal to the `certificateId` of the certificate the device used to authenticate itself:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/${iot:CertificateId}"]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/client1",
                "arn:aws:iot:us-east-1:123456789012:client/client2",
                "arn:aws:iot:us-east-1:123456789012:client/client3"
            ]
        }
    ]
}
```

For devices registered in AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core with a client ID that matches the thing name, and to publish to a topic whose name is equal to the subject's `CommonName` field of the certificate the device used to authenticate itself:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],

```

```

        "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/
${iot:Certificate.Subject.CommonName}"]
    },
    {
        "Effect": "Allow",
        "Action": [
            "iot:Connect"
        ],
        "Resource": ["arn:aws:iot:us-east-1:123456789012:client/
${iot:Connection.Thing.ThingName}"]
    }
}
}

```

Note

In this example, the certificate's subject common name is used as the topic identifier, with the assumption that the subject common name is unique for each registered certificate. If the certificates are shared across multiple devices, the subject common name is the same for all the devices that share this certificate, thereby allowing publish privileges to the same topic from multiple devices (not recommended).

For devices not registered in AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core with client IDs, `client1`, `client2`, and `client3` and to publish to a topic whose name is equal to the subject's `CommonName` field of the certificate the device used to authenticate itself:

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/
${iot:Certificate.Subject.CommonName}"]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/client1",
                "arn:aws:iot:us-east-1:123456789012:client/client2",
                "arn:aws:iot:us-east-1:123456789012:client/client3"
            ]
        }
    ]
}

```

Note

In this example, the certificate's subject common name is used as the topic identifier, with the assumption that the subject common name is unique for each registered certificate. If the certificates are shared across multiple devices, the subject common name is the same for all the devices that share this certificate, thereby allowing publish privileges to the same topic from multiple devices (not recommended).

For devices registered in the AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core with a client ID that matches the thing name, and to publish to a topic whose name is prefixed with `admin/` when the certificate used to authenticate the device has its `Subject.CommonName.2` field set to `Administrator`:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": ["arn:aws:iot:us-east-1:123456789012:client/${iot:Connection.Thing.ThingName}"]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/admin/*"],
            "Condition": {
                "StringEquals": {
                    "iot:Certificate.Subject.CommonName.2": "Administrator"
                }
            }
        }
    ]
}
```

For devices not registered in AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core with client IDs `client1`, `client2`, and `client3` and to publish to a topic whose name is prefixed with `admin/` when the certificate used to authenticate the device has its `Subject.CommonName.2` field set to `Administrator`:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/client1",
                "arn:aws:iot:us-east-1:123456789012:client/client2",
                "arn:aws:iot:us-east-1:123456789012:client/client3"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/admin/*"],
            "Condition": {
                "StringEquals": {
                    "iot:Certificate.Subject.CommonName.2": "Administrator"
                }
            }
        }
    ]
}
```

For devices registered in AWS IoT Core registry, the following policy allows a device to use its thing name to publish on a specific topic that consists of `admin/` followed by the `ThingName` when the certificate used to authenticate the device has any one of its `Subject.CommonName` fields set to `Administrator`:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": ["arn:aws:iot:us-east-1:123456789012:client/${iot:Connection.Thing.ThingName}"]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/admin/${iot:Connection.Thing.ThingName}"],
            "Condition": {
                "ForAnyValue:StringEquals": {
                    "iot:Certificate.Subject.CommonName.List": "Administrator"
                }
            }
        }
    ]
}
```

For devices not registered in AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core with client IDs `client1`, `client2`, and `client3` and to publish to the topic `admin` when the certificate used to authenticate the device has any one of its `Subject.CommonName` fields set to `Administrator`:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/client1",
                "arn:aws:iot:us-east-1:123456789012:client/client2",
                "arn:aws:iot:us-east-1:123456789012:client/client3"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/admin"],
            "Condition": {
                "ForAnyValue:StringEquals": {
                    "iot:Certificate.Subject.CommonName.List": "Administrator"
                }
            }
        }
    ]
}
```

Thing policy examples

The following policy allows a device to connect if the certificate used to authenticate with AWS IoT Core is attached to the thing for which the policy is being evaluated:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": ["iot:Connect"],  
            "Resource": ["*"],  
            "Condition": {  
                "Bool": {  
                    "iot:Connection.Thing.IsAttached": ["true"]  
                }  
            }  
        }  
    ]  
}
```

Basic job policy example

This sample shows the policy statements required for a job target that's a single device to receive a job request and communicate job execution status with AWS IoT.

Replace `us-west-2:57EXAMPLE833` with your AWS Region, a colon character (:), and your 12-digit AWS account number, and then replace `uniqueThingName` with the name of the thing resource that represents the device in AWS IoT.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:Connect"  
            ],  
            "Resource": [  
                "arn:aws:iot:us-west-2:57EXAMPLE833:client/uniqueThingName"  
            ]  
        },  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:Publish"  
            ],  
            "Resource": [  
                "arn:aws:iot:us-west-2:57EXAMPLE833:topic/test/dc/pubtopic",  
                "arn:aws:iot:us-west-2:57EXAMPLE833:topic/$aws/events/job/*",  
                "arn:aws:iot:us-west-2:57EXAMPLE833:topic/$aws/events/jobExecution/*",  
                "arn:aws:iot:us-west-2:57EXAMPLE833:topic/$aws/things/uniqueThingName/jobs/*"  
            ]  
        },  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:Subscribe"  
            ],  
            "Resource": [  
                "arn:aws:iot:us-west-2:57EXAMPLE833:topicfilter/test/dc/subtopic",  
                "arn:aws:iot:us-west-2:57EXAMPLE833:topic/$aws/events/jobExecution/*",  
            ]  
        }  
    ]  
}
```

```
        "arn:aws:iot:us-west-2:57EXAMPLE833:topicfilter/$aws/things/uniqueThingName/jobs/*"
    ],
},
{
    "Effect": "Allow",
    "Action": [
        "iot:Receive"
    ],
    "Resource": [
        "arn:aws:iot:us-west-2:57EXAMPLE833:topic/test/dc/subtopic",
        "arn:aws:iot:us-west-2:57EXAMPLE833:topic/$aws/things/uniqueThingName/jobs/*"
    ]
},
{
    "Effect": "Allow",
    "Action": [
        "iot:DescribeJobExecution",
        "iot:GetPendingJobExecutions",
        "iot:StartNextPendingJobExecution",
        "iot:UpdateJobExecution"
    ],
    "Resource": [
        "arn:aws:iot:us-west-2:57EXAMPLE833:topic/$aws/things/uniqueThingName"
    ]
}
]
```

Authorization with Amazon Cognito identities

There are two types of Amazon Cognito identities: unauthenticated and authenticated. If your app supports unauthenticated Amazon Cognito identities, no authentication is performed so you don't know who the user is. For these users, you grant permission by attaching an IAM role to an unauthenticated identity pool. We recommend you only grant access to those resources you want available to unknown users.

When your app supports authenticated Amazon Cognito identities, in order to authenticate users, you need to specify a policy in two places. Attach an IAM policy to the authenticated Amazon Cognito Identity pool and attach an AWS IoT Core policy to the Amazon Cognito Identity.

Authenticated and unauthenticated users are different identity types. If you don't attach an AWS IoT policy to the Amazon Cognito Identity, an authenticated user fails authorization in AWS IoT and doesn't have access to AWS IoT resources and actions. For more information about creating policies for Amazon Cognito identities, see [Publish/Subscribe policy examples \(p. 332\)](#).

The following example web applications on GitHub show how to incorporate policy attachment to authenticated users into the user signup and authentication process.

- [MQTT publish/subscribe React web application using AWS Amplify and the AWS IoT Device SDK for JavaScript](#)
- [MQTT publish/subscribe React web application using AWS Amplify, the AWS IoT Device SDK for JavaScript, and a Lambda function](#)

Amplify is a set of tools and services that help you build web and mobile applications that integrate with AWS services. For more information about Amplify, see [Amplify Framework Documentation](#).

Both examples perform the following steps.

1. When a user signs up for an account, the application creates an Amazon Cognito user pool and identity.

- When a user authenticates, the application creates and attaches a policy to the identity. This gives the user publish and subscribe permissions.

Note

The application gives the authorized user permission to perform all AWS IoT Core operations on all AWS IoT Core resources. In your own applications, make sure to give only the permissions that authorized users need.

- The user can use the application to publish and subscribe to MQTT topics.

The first example uses the `AttachPolicy` API directly inside the authentication operation. The following example demonstrates how to implement this API inside a React web application that uses Amplify and the AWS IoT Device SDK for JavaScript.

```
function attachPolicy(id, policyName) {
    var Iot = new AWS.Iot({region: AWSConfiguration.region, apiVersion: AWSConfiguration.apiVersion, endpoint: AWSConfiguration.endpoint});
    var params = {policyName: policyName, target: id};

    console.log("Attach IoT Policy: " + policyName + " with cognito identity id: " + id);
    Iot.attachPolicy(params, function(err, data) {
        if (err) {
            if (err.code !== 'ResourceAlreadyExistsException') {
                console.log(err);
            }
        } else {
            console.log("Successfully attached policy with the identity", data);
        }
    });
}
```

This code appears in the [AuthDisplay.js](#) file.

The second example implements the `AttachPolicy` API in a Lambda function. The following example shows how the Lambda uses this API.

```
iot.attachPolicy(params, function(err, data) {
    if (err) {
        if (err.code !== 'ResourceAlreadyExistsException') {
            console.log(err);
            res.json({error: err, url: req.url, body: req.body});
        }
    } else {
        console.log(data);
        res.json({success: 'Create and attach policy call succeed!', url: req.url, body: req.body});
    }
});
```

This code appears inside the `iot.GetPolicy` function in the [app.js](#) file.

Note

When you call the function with AWS credentials that you obtain through Amazon Cognito Identity pools, the context object in your Lambda function contains a value for `context.cognito_identity_id`. For more information, see the following.

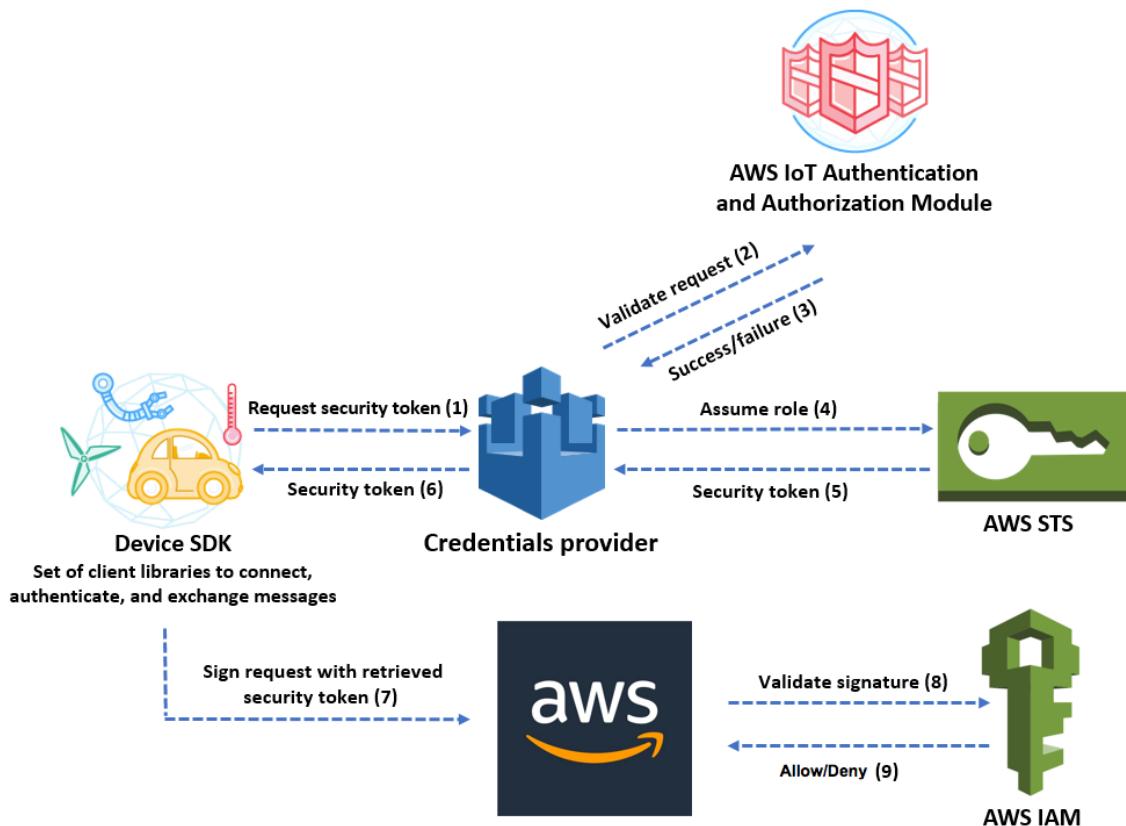
- [AWS Lambda context object in Node.js](#)
- [AWS Lambda context object in Python](#)
- [AWS Lambda context object in Ruby](#)
- [AWS Lambda context object in Java](#)
- [AWS Lambda context object in Go](#)
- [AWS Lambda context object in C#](#)
- [AWS Lambda context object in PowerShell](#)

Authorizing direct calls to AWS services using AWS IoT Core credential provider

Devices can use X.509 certificates to connect to AWS IoT Core using TLS mutual authentication protocols. Other AWS services do not support certificate-based authentication, but they can be called using AWS credentials in [AWS Signature Version 4 format](#). The [Signature Version 4 algorithm](#) normally requires the caller to have an access key ID and a secret access key. AWS IoT Core has a credentials provider that allows you to use the built-in [X.509 certificate](#) as the unique device identity to authenticate AWS requests. This eliminates the need to store an access key ID and a secret access key on your device.

The credentials provider authenticates a caller using an X.509 certificate and issues a temporary, limited-privilege security token. The token can be used to sign and authenticate any AWS request. This way of authenticating your AWS requests requires you to create and configure an [AWS Identity and Access Management \(IAM\) role](#) and attach appropriate IAM policies to the role so that the credentials provider can assume the role on your behalf. For more information about AWS IoT Core and IAM, see [Identity and access management for AWS IoT \(p. 364\)](#).

The following diagram illustrates the credentials provider workflow.



1. The AWS IoT Core device makes an HTTPS request to the credentials provider for a security token. The request includes the device X.509 certificate for authentication.
2. The credentials provider forwards the request to the AWS IoT Core authentication and authorization module to validate the certificate and verify that the device has permission to request the security token.
3. If the certificate is valid and has permission to request a security token, the AWS IoT Core authentication and authorization module returns success. Otherwise, it sends an exception to the device.
4. After successfully validating the certificate, the credentials provider invokes the [AWS Security Token Service \(AWS STS\)](#) to assume the IAM role that you created for it.
5. AWS STS returns a temporary, limited-privilege security token to the credentials provider.
6. The credentials provider returns the security token to the device.
7. The device uses the security token to sign an AWS request with AWS Signature Version 4.
8. The requested service invokes IAM to validate the signature and authorize the request against access policies attached to the IAM role that you created for the credentials provider.
9. If IAM validates the signature successfully and authorizes the request, the request is successful. Otherwise, IAM sends an exception.

The following section describes how to use a certificate to get a security token. It is written with the assumption that you have already [registered a device](#) and [created and activated your own certificate](#) for it.

How to use a certificate to get a security token

1. Configure the IAM role that the credentials provider assumes on behalf of your device. Attach the following trust policy to the role.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {"Effect": "Allow",  
         "Principal": {"Service": "credentials.iot.amazonaws.com"},  
         "Action": "sts:AssumeRole"}  
    ]  
}
```

For each AWS service that you want to call, attach an access policy to the role. The credentials provider supports the following policy variables:

- `credentials-iot:ThingName`
- `credentials-iot:ThingTypeName`
- `credentials-iot:AwsCertificateId`

When the device provides the thing name in its request to an AWS service, the credentials provider adds `credentials-iot:ThingName` and `credentials-iot:ThingTypeName` as context variables to the security token. The credentials provider provides `credentials-iot:AwsCertificateId` as a context variable even if the device doesn't provide the thing name in the request. You pass the thing name as the value of the `x-amzn-iot-thingname` HTTP request header.

These three variables work for IAM policies only, not AWS IoT Core policies.

2. Make sure that the user who performs the next step (creating a role alias) has permission to pass the newly created role to AWS IoT Core. The following policy gives both `iam:GetRole` and `iam:PassRole` permissions to an AWS user. The `iam:GetRole` permission allows the user to get information about the role that you've just created. The `iam:PassRole` permission allows the user to pass the role to another AWS service.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {"Effect": "Allow",  
         "Action": [  
             "iam:GetRole",  
             "iam:PassRole"  
         ],  
         "Resource": "arn:aws:iam::your AWS account id:role/your role name"  
    ]  
}
```

3. Create an AWS IoT Core role alias. The device that is going to make direct calls to AWS services must know which role ARN to use when connecting to AWS IoT Core. Hard-coding the role ARN is not a good solution because it requires you to update the device whenever the role ARN changes. A better solution is to use the `CreateRoleAlias` API to create a role alias that points to the role ARN. If the role ARN changes, you simply update the role alias. No change is required on the device. This API takes the following parameters:

roleAlias

Required. An arbitrary string that identifies the role alias. It serves as the primary key in the role alias data model. It contains 1-128 characters and must include only alphanumeric characters and the =, @, and - symbols. Uppercase and lowercase alphabetic characters are allowed.

roleArn

Required. The ARN of the role to which the role alias refers.

credentialDurationInSeconds

Optional. How long (in seconds) the credential is valid. The minimum value is 900 seconds (15 minutes). The maximum value is 43,200 seconds (12 hours). The default value is 3,600 seconds (1 hour).

Note

The AWS IoT Core Credential Provider can issue a credential with a maximum lifetime is 43,200 seconds (12 hours). Having the credential be valid for up to 12 hours can help reduce the number of calls to the credential provider by caching the credential longer.

For more information about this API, see [CreateRoleAlias](#).

4. Attach a policy to the device certificate. The policy attached to the device certificate must grant the device permission to assume the role. You do this by granting permission for the `iot:AssumeRoleWithCertificate` action to the role alias, as in the following example.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": "iot:AssumeRoleWithCertificate",  
            "Resource": "arn:aws:iot:your_region:your_aws_account_id:rolealias/your_role  
alias"  
        }  
    ]  
}
```

5. Make an HTTPS request to the credentials provider to get a security token. Supply the following information:

- *Certificate*: Because this is an HTTP request over TLS mutual authentication, you must provide the certificate and the private key to your client while making the request. Use the same certificate and private key you used when you registered your certificate with AWS IoT Core.

To make sure your device is communicating with AWS IoT Core (and not a service impersonating it), see [Server Authentication](#), follow the links to download the appropriate CA certificates, and then copy them to your device.

- *RoleAlias*: The name of the role alias that you created for the credentials provider.
- *ThingName*: The thing name that you created when you registered your AWS IoT Core thing. This is passed as the value of the `x-amzn-iot-thingname` HTTP header. This value is required only if you are using thing attributes as policy variables in AWS IoT Core or IAM policies.

Note

The *ThingName* that you provide in `x-amzn-iot-thingname` must match the name of the AWS IoT Thing resource assigned to a cert. If it doesn't match, a 403 error is returned.

Run the following command in the AWS CLI to obtain the credentials provider endpoint for your AWS account. For more information about this API, see [DescribeEndpoint](#).

```
aws iot describe-endpoint --endpoint-type iot:CredentialProvider
```

The following JSON object is sample output of the **describe-endpoint** command. It contains the `endpointAddress` that you use to request a security token.

```
{  
    "endpointAddress": "your_aws_account_specific_prefix.credentials.iot.your  
    region.amazonaws.com"  
}
```

Use the endpoint to make an HTTPS request to the credentials provider to return a security token. The following example command uses `curl`, but you can use any HTTP client.

```
curl -v -cert your_certificate --key your_device_certificate_key_pair  
-H "x-amzn-iot-thingname: your_thing_name" --cacert AmazonRootCA1.pem  
https://your_endpoint /role-aliases/your_role_alias/credentials
```

This command returns a security token object that contains an `accessKeyId`, a `secretAccessKey`, a `sessionToken`, and an expiration. The following JSON object is sample output of the `curl` command.

```
{"credentials": {"accessKeyId": "access key", "secretAccessKey": "secret access  
key", "sessionToken": "session token", "expiration": "2018-01-18T09:18:06Z"}}
```

You can then use the `accessKeyId`, `secretAccessKey`, and `sessionToken` values to sign requests to AWS services. For an end-to-end demonstration, see [How to Eliminate the Need for Hard-Coded AWS Credentials in Devices by Using the AWS IoT Credential Provider](#) blog post on the [AWS Security Blog](#).

Cross account access with IAM

AWS IoT Core allows you to enable a principal to publish or subscribe to a topic that is defined in an AWS account not owned by the principal. You configure cross account access by creating an IAM policy and IAM role and then attaching the policy to the role.

First, create a customer managed IAM policy as described in [Creating IAM Policies](#), just like you would for other users and certificates in your AWS account.

For devices registered in AWS IoT Core registry, the following policy grants permission to devices connect to AWS IoT Core using a client ID that matches the device's thing name and to publish to the `my/topic/thing-name` where `thing-name` is the device's thing name:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:Connect"  
            ],  
            "Resource": ["arn:aws:iot:us-east-1:123456789012:client/  
            ${iot:Connection.Thing.ThingName}"]  
        },  
    ]  
}
```

```
{
    "Effect": "Allow",
    "Action": [
        "iot:Publish"
    ],
    "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/my/topic/
${iot:Connection.Thing.ThingName}"],
}
]
```

For devices not registered in AWS IoT Core registry, the following policy grants permission to a device to use the thing name `client1` registered in your account's (123456789012) AWS IoT Core registry to connect to AWS IoT Core and to publish to a client ID-specific topic whose name is prefixed with `my/topic/`:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/client1"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/my/topic/${iot:ClientId}"
            ]
        }
    ]
}
```

Next, follow the steps in [Creating a Role to Delegate Permissions to an IAM User](#). Enter the account ID of the AWS account with which you want to share access. Then, in the final step, attach the policy you just created to the role. If, at a later time, you need to modify the AWS account ID to which you are granting access, you can use the following trust policy format to do so:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam:us-east-1:567890123456:user:MyUser"
            },
            "Action": "sts:AssumeRole",
        }
    ]
}
```

Data protection in AWS IoT Core

The AWS [shared responsibility model](#) applies to data protection in AWS IoT Core. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. This content includes the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see the [Data Privacy FAQ](#). For information about data protection in Europe, see the [AWS Shared Responsibility Model and GDPR](#) blog post on the [AWS Security Blog](#).

For data protection purposes, we recommend that you protect AWS account credentials and set up individual user accounts with AWS Identity and Access Management (IAM). That way each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We recommend TLS 1.2 or later.
- Set up API and user activity logging with AWS CloudTrail.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing personal data that is stored in Amazon S3.
- If you require FIPS 140-2 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see [Federal Information Processing Standard \(FIPS\) 140-2](#).

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form fields such as a **Name** field. This includes when you work with AWS IoT or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

For more information about data protection, see the [AWS Shared Responsibility Model and GDPR](#) blog post on the [AWS Security Blog](#).

AWS IoT devices gather data, perform some manipulation on that data, and then send that data to another web service. You might choose to store some data on your device for a short period of time. You're responsible for providing any data protection on that data at rest. When your device sends data to AWS IoT, it does so over a TLS connection as discussed later in this section. AWS IoT devices can send data to any AWS service. For more information about each service's data security, see the documentation for that service. AWS IoT can be configured to write logs to CloudWatch Logs and log AWS IoT API calls to AWS CloudTrail. For more information about data security for these services, see [Authentication and Access Control for Amazon CloudWatch](#) and [Encrypting CloudTrail Log Files with AWS KMS-Managed Keys](#).

Data encryption in AWS IoT

By default, all AWS IoT data in transit and at rest is encrypted. [Data in transit is encrypted using TLS \(p. 362\)](#), and data at rest is encrypted using AWS owned keys. AWS IoT does not currently support customer-managed AWS KMS keys (KMS keys) from AWS Key Management Service (AWS KMS); however, Device Advisor and AWS IoT Wireless use only an AWS owned key to encrypt customer data.

Transport security in AWS IoT

The AWS IoT message broker and Device Shadow service encrypt all communication while in-transit by using [TLS version 1.2](#). TLS is used to ensure the confidentiality of the application protocols (MQTT, HTTP, and WebSocket) supported by AWS IoT. TLS support is available in a number of programming languages and operating systems. Data within AWS is encrypted by the specific AWS service. For more information about data encryption on other AWS services, see the security documentation for that service.

For MQTT, TLS encrypts the connection between the device and the broker. TLS client authentication is used by AWS IoT to identify devices. For HTTP, TLS encrypts the connection between the device and the broker. Authentication is delegated to AWS Signature Version 4.

AWS IoT requires devices to send the [Server Name Indication \(SNI\) extension](#) to the Transport Layer Security (TLS) protocol and provide the complete endpoint address in the `host_name` field. The `host_name` field must contain the endpoint you are calling, and it must be:

- The `endpointAddress` returned by `aws iot describe-endpoint --endpoint-type iot:Data-ATS`
or
- The `domainName` returned by `aws iot describe-domain-configuration --domain-configuration-name "domain_configuration_name"`

Connections attempted by devices without the correct `host_name` value will be refused and logged in CloudWatch.

AWS IoT does not support the SessionTicket TLS extension.

Transport security for LoRaWAN wireless devices

LoRaWAN devices follow the security practices described in [LoRaWAN™ SECURITY: A White Paper Prepared for the LoRa Alliance™ by Gemalto, Actility, and Semtech](#).

For more information about transport security with LoRaWAN devices, see [Data security with AWS IoT Core for LoRaWAN \(p. 1097\)](#).

TLS cipher suite support

AWS IoT supports the following cipher suites:

- ECDHE-ECDSA-AES128-GCM-SHA256 (recommended)
- ECDHE-RSA-AES128-GCM-SHA256 (recommended)
- ECDHE-ECDSA-AES128-SHA256
- ECDHE-RSA-AES128-SHA256
- ECDHE-ECDSA-AES128-SHA
- ECDHE-RSA-AES128-SHA
- ECDHE-ECDSA-AES256-GCM-SHA384
- ECDHE-RSA-AES256-GCM-SHA384
- ECDHE-ECDSA-AES256-SHA384
- ECDHE-RSA-AES256-SHA384
- ECDHE-RSA-AES256-SHA

- ECDHE-ECDSA-AES256-SHA
- AES128-GCM-SHA256
- AES128-SHA256
- AES128-SHA
- AES256-GCM-SHA384
- AES256-SHA256
- AES256-SHA

Data encryption in AWS IoT

Data protection refers to protecting data while in-transit (as it travels to and from AWS IoT) and at rest (while it is stored on devices or by other AWS services). All data sent to AWS IoT is sent over an TLS connection using MQTT, HTTPS, and WebSocket protocols, making it secure by default while in transit. AWS IoT devices collect data and then send it to other AWS services for further processing. For more information about data encryption on other AWS services, see the security documentation for that service.

FreeRTOS provides a PKCS#11 library that abstracts key storage, accessing cryptographic objects and managing sessions. It is your responsibility to use this library to encrypt data at rest on your devices. For more information, see [FreeRTOS Public Key Cryptography Standard \(PKCS\) #11 Library](#).

Device Advisor

Encryption in transit

Data sent to and from Device Advisor is encrypted in transit. All data sent to and from the service when using the Device Advisor APIs is encrypted using Signature Version 4. For more information about how AWS API requests are signed, see [Signing AWS API requests](#). All data sent from your test devices to your Device Advisor test endpoint is sent over a TLS connection so it is secure by default in transit.

Key management in AWS IoT

All connections to AWS IoT are done using TLS, so no client-side encryption keys are necessary for the initial TLS connection.

Devices must authenticate using an X.509 certificate or an Amazon Cognito Identity. You can have AWS IoT generate a certificate for you, in which case it will generate a public/private key pair. If you are using the AWS IoT console you will be prompted to download the certificate and keys. If you are using the `create-keys-and-certificate` CLI command, the certificate and keys are returned by the CLI command. You are responsible for copying the certificate and private key onto your device and keeping it safe.

AWS IoT does not currently support customer-managed AWS KMS keys (KMS keys) from AWS Key Management Service (AWS KMS); however, Device Advisor and AWS IoT Wireless use only an AWS owned key to encrypt customer data.

Device Advisor

All data sent to Device Advisor when using the AWS APIs is encrypted at rest. Device Advisor encrypts all of your data at rest using KMS keys stored and managed in [AWS Key Management Service](#). Device Advisor encrypts your data using AWS owned keys. For more information about AWS owned keys, see [AWS owned keys](#).

Identity and access management for AWS IoT

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use AWS IoT resources. IAM is an AWS service that you can use with no additional charge.

Topics

- [Audience \(p. 364\)](#)
- [Authenticating with IAM identities \(p. 364\)](#)
- [Managing access using policies \(p. 366\)](#)
- [How AWS IoT works with IAM \(p. 368\)](#)
- [AWS IoT identity-based policy examples \(p. 385\)](#)
- [Troubleshooting AWS IoT identity and access \(p. 388\)](#)

Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work that you do in AWS IoT.

Service user – If you use the AWS IoT service to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more AWS IoT features to do your work, you might need additional permissions. Understanding how access is managed can help you request the right permissions from your administrator. If you cannot access a feature in AWS IoT, see [Troubleshooting AWS IoT identity and access \(p. 388\)](#).

Service administrator – If you're in charge of AWS IoT resources at your company, you probably have full access to AWS IoT. It's your job to determine which AWS IoT features and resources your employees should access. You must then submit requests to your IAM administrator to change the permissions of your service users. Review the information on this page to understand the basic concepts of IAM. To learn more about how your company can use IAM with AWS IoT, see [How AWS IoT works with IAM \(p. 368\)](#).

IAM administrator – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to AWS IoT. To view example AWS IoT identity-based policies that you can use in IAM, see [AWS IoT identity-based policy examples \(p. 385\)](#).

Authenticating with IAM identities

In AWS IoT identities can be device (X.509) certificates, Amazon Cognito identities, or IAM users or groups. This topic discusses IAM identities only. For more information about the other identities that AWS IoT supports, see [Client authentication \(p. 282\)](#).

Authentication is how you sign in to AWS using your identity credentials. For more information about signing in using the AWS Management Console, see [Signing in to the AWS Management Console as an IAM user or root user](#) in the *IAM User Guide*.

You must be *authenticated* (signed in to AWS) as the AWS account root user, an IAM user, or by assuming an IAM role. You can also use your company's single sign-on authentication or even sign in using Google or Facebook. In these cases, your administrator previously set up identity federation using IAM roles. When you access AWS using credentials from another company, you are assuming a role indirectly.

To sign in directly to the [AWS Management Console](#), use your password with your root user email address or your IAM user name. You can access AWS programmatically using your root user or IAM

users access keys. AWS provides SDK and command line tools to cryptographically sign your request using your credentials. If you don't use AWS tools, you must sign the request yourself. Do this using *Signature Version 4*, a protocol for authenticating inbound API requests. For more information about authenticating requests, see [Signature Version 4 signing process](#) in the *AWS General Reference*.

Regardless of the authentication method that you use, you might also be required to provide additional security information. For example, AWS recommends that you use multi-factor authentication (MFA) to increase the security of your account. To learn more, see [Using multi-factor authentication \(MFA\) in AWS](#) in the *IAM User Guide*.

AWS account root user

When you first create an AWS account, you begin with a single sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the *AWS account root user* and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you do not use the root user for your everyday tasks, even the administrative ones. Instead, adhere to the [best practice of using the root user only to create your first IAM user](#). Then securely lock away the root user credentials and use them to perform only a few account and service management tasks.

IAM users and groups

An *IAM user* is an identity within your AWS account that has specific permissions for a single person or application. An IAM user can have long-term credentials such as a user name and password or a set of access keys. To learn how to generate access keys, see [Managing access keys for IAM users](#) in the *IAM User Guide*. When you generate access keys for an IAM user, make sure you view and securely save the key pair. You cannot recover the secret access key in the future. Instead, you must generate a new access key pair.

An *IAM group* is an identity that specifies a collection of IAM users. You can't sign in as a group. You can use groups to specify permissions for multiple users at a time. Groups make permissions easier to manage for large sets of users. For example, you could have a group named *IAMAdmins* and give that group permissions to administer IAM resources.

Users are different from roles. A user is uniquely associated with one person or application, but a role is intended to be assumable by anyone who needs it. Users have permanent long-term credentials, but roles provide temporary credentials. To learn more, see [When to create an IAM user \(instead of a role\)](#) in the *IAM User Guide*.

IAM roles

An *IAM role* is an identity within your AWS account that has specific permissions. It is similar to an IAM user, but is not associated with a specific person. You can temporarily assume an IAM role in the AWS Management Console by [switching roles](#). You can assume a role by calling an AWS CLI or AWS API operation or by using a custom URL. For more information about methods for using roles, see [Using IAM roles](#) in the *IAM User Guide*.

IAM roles with temporary credentials are useful in the following situations:

- **Temporary IAM user permissions** – An IAM user can assume an IAM role to temporarily take on different permissions for a specific task.
- **Federated user access** – Instead of creating an IAM user, you can use existing identities from AWS Directory Service, your enterprise user directory, or a web identity provider. These are known as *federated users*. AWS assigns a role to a federated user when access is requested through an [identity provider](#). For more information about federated users, see [Federated users and roles](#) in the *IAM User Guide*.

- **Cross-account access** – You can use an IAM role to allow someone (a trusted principal) in a different account to access resources in your account. Roles are the primary way to grant cross-account access. However, with some AWS services, you can attach a policy directly to a resource (instead of using a role as a proxy). To learn the difference between roles and resource-based policies for cross-account access, see [How IAM roles differ from resource-based policies](#) in the *IAM User Guide*.
- **Cross-service access** – Some AWS services use features in other AWS services. For example, when you make a call in a service, it's common for that service to run applications in Amazon EC2 or store objects in Amazon S3. A service might do this using the calling principal's permissions, using a service role, or using a service-linked role.
 - **Principal permissions** – When you use an IAM user or role to perform actions in AWS, you are considered a principal. Policies grant permissions to a principal. When you use some services, you might perform an action that then triggers another action in a different service. In this case, you must have permissions to perform both actions. To see whether an action requires additional dependent actions in a policy, see [Actions, Resources, and Condition Keys for AWS IoT](#) in the *Service Authorization Reference*.
 - **Service role** – A service role is an [IAM role](#) that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see [Creating a role to delegate permissions to an AWS service](#) in the *IAM User Guide*.
 - **Service-linked role** – A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your IAM account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.
- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see [Using an IAM role to grant permissions to applications running on Amazon EC2 instances](#) in the *IAM User Guide*.

To learn whether to use IAM roles or IAM users, see [When to create an IAM role \(instead of a user\)](#) in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to IAM identities or AWS resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. You can sign in as the root user or an IAM user, or you can assume an IAM role. When you then make a request, AWS evaluates the related identity-based or resource-based policies. Permissions in the policies determine whether the request is allowed or denied. Most policies are stored in AWS as JSON documents. For more information about the structure and contents of JSON policy documents, see [Overview of JSON policies](#) in the *IAM User Guide*.

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

Every IAM entity (user or role) starts with no permissions. In other words, by default, users can do nothing, not even change their own password. To give a user permission to do something, an administrator must attach a permissions policy to a user. Or the administrator can add the user to a group that has the intended permissions. When an administrator gives permissions to a group, all users in that group are granted those permissions.

IAM policies define permissions for an action regardless of the method that you use to perform the operation. For example, suppose that you have a policy that allows the `iam:GetRole` action. A user with that policy can get role information from the AWS Management Console, the AWS CLI, or the AWS API.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see [Creating IAM policies](#) in the *IAM User Guide*.

Identity-based policies can be further categorized as *inline policies* or *managed policies*. Inline policies are embedded directly into a single user, group, or role. Managed policies are standalone policies that you can attach to multiple users, groups, and roles in your AWS account. Managed policies include AWS managed policies and customer managed policies. To learn how to choose between a managed policy or an inline policy, see [Choosing between managed policies and inline policies](#) in the *IAM User Guide*.

Resource-based policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must [specify a principal](#) in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Access control lists (ACLs)

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Amazon S3, AWS WAF, and Amazon VPC are examples of services that support ACLs. To learn more about ACLs, see [Access control list \(ACL\) overview](#) in the *Amazon Simple Storage Service Developer Guide*.

Other policy types

AWS supports additional, less-common policy types. These policy types can set the maximum permissions granted to you by the more common policy types.

- **Permissions boundaries** – A permissions boundary is an advanced feature in which you set the maximum permissions that an identity-based policy can grant to an IAM entity (IAM user or role). You can set a permissions boundary for an entity. The resulting permissions are the intersection of entity's identity-based policies and its permissions boundaries. Resource-based policies that specify the user or role in the `Principal` field are not limited by the permissions boundary. An explicit deny in any of these policies overrides the allow. For more information about permissions boundaries, see [Permissions boundaries for IAM entities](#) in the *IAM User Guide*.
- **Service control policies (SCPs)** – SCPs are JSON policies that specify the maximum permissions for an organization or organizational unit (OU) in AWS Organizations. AWS Organizations is a service for grouping and centrally managing multiple AWS accounts that your business owns. If you enable all features in an organization, then you can apply service control policies (SCPs) to any or all of your accounts. The SCP limits permissions for entities in member accounts, including each AWS account root user. For more information about Organizations and SCPs, see [How SCPs work](#) in the *AWS Organizations User Guide*.
- **Session policies** – Session policies are advanced policies that you pass as a parameter when you programmatically create a temporary session for a role or federated user. The resulting session's permissions are the intersection of the user or role's identity-based policies and the session policies.

Permissions can also come from a resource-based policy. An explicit deny in any of these policies overrides the allow. For more information, see [Session policies](#) in the *IAM User Guide*.

Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see [Policy evaluation logic](#) in the *IAM User Guide*.

How AWS IoT works with IAM

Before you use IAM to manage access to AWS IoT, you should understand which IAM features are available to use with AWS IoT. To get a high-level view of how AWS IoT and other AWS services work with IAM, see [AWS Services That Work with IAM](#) in the *IAM User Guide*.

Topics

- [IAM policies \(p. 368\)](#)
- [AWS IoT identity-based policies \(p. 369\)](#)
- [AWS IoT resource-based policies \(p. 384\)](#)
- [Authorization based on AWS IoT tags \(p. 384\)](#)
- [AWS IoT IAM roles \(p. 385\)](#)

IAM policies

AWS IoT works with AWS IoT and IAM policies. This topic discusses IAM policies only. For more information, see [AWS IoT Core policies \(p. 314\)](#). AWS Identity and Access Management defines a policy action for each operation defined by AWS IoT, including control plane and data plane APIs.

IAM managed policies

AWS IoT provides a set of IAM managed policies you can either use as-is or as a starting point for creating custom IAM policies. These policies allow access to configuration and data operations. Configuration operations allow you to create things, certificates, policies, and rules. Data operations send data over MQTT or HTTP protocols. The following table describes these templates.

Policy template	Description
AWSIoTConfigAccess	Allows the associated identity access to all AWS IoT configuration operations. This policy can affect data processing and storage.
AWSIoTConfigReadOnlyAccess	Allows the associated identity to access read-only configuration operations.
AWSIoTDataAccess	Allows the associated identity full access to all AWS IoT data operations. Data operations send data over MQTT or HTTP protocols.
AWSIoTEventsFullAccess	Allows the associated identity full access to AWS IoT events.
AWSIoTEventsReadOnlyAccess	Allows the associated identity read only access to AWS IoT events.

Policy template	Description
AWSIoTFullAccess	Allows the associated identity full access to all AWS IoT configuration and messaging operations.
AWSIoTLogging	Allows the associated identity to create Amazon CloudWatch Logs groups and stream logs to the groups. This policy is attached to your CloudWatch logging role.
AWSIoTOTUpdate	Allows the associated identity to create AWS IoT jobs, AWS IoT code signing jobs, and to describe AWS code signer jobs.
AWSIoTRuleActions	Allows the associated identity access to all AWS services supported in AWS IoT rule actions.
AWSIoTThingsRegistration	Allows the associated identity to register things in bulk using the StartThingRegistrationTask API. This policy can affect data processing and storage.
AWSIoTWirelessDataAccess	Allows the associated identity to send data to AWS IoT wireless devices.
AWSIoTWirelessFullAccess	Allows the associated identity full access to AWS IoT Wireless.
AWSIoTWirelessFullPublishAccess	Grants AWS IoT Wireless limited access to publish to AWS IoT rules on your behalf.
AWSIoTWirelessLogging	Allows the associated identity to create Amazon CloudWatch log groups and stream logs to the groups. This policy is attached to your CloudWatch logging role.
AWSIoTWirelessReadOnlyAccess	Allows the associated identity read only access to AWS IoT Wireless.
AWSIoTWirelessGatewayCertManager	Allows the associated identity access to create, list, and describe AWS IoT certificates.

AWS IoT identity-based policies

With IAM identity-based policies, you can specify allowed or denied actions and resources as well as the conditions under which actions are allowed or denied. AWS IoT supports specific actions, resources, and condition keys. To learn about all of the elements that you use in a JSON policy, see [IAM JSON Policy Elements Reference](#) in the [IAM User Guide](#).

Actions

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The **Action** element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Policy actions usually have the same name as the associated AWS API operation. There are some exceptions, such as *permission-only actions* that don't have a matching API operation. There are also some operations that require multiple actions in a policy. These additional actions are called *dependent actions*.

Include actions in a policy to grant permissions to perform the associated operation.

The following table lists the IAM IoT actions, the associated AWS IoT API, and the resource the action manipulates.

Policy actions	AWS IoT API	Resources
iot:AcceptCertificateTransfer	TransferCertificateTransfer	arn:aws:iot: <i>region:account-id:cert/cert-id</i>
		Note The AWS account specified in the ARN must be the account to which the certificate is being transferred.
iot:AddThingToThingGroup	AddThingToThingGroup	arn:aws:iot: <i>region:account-id:thinggroup/thing-group-name</i> arn:aws:iot: <i>region:account-id:thing/thing-name</i>
iot:AssociateTargetsWithJob	AssociateTargetsWithJob	
iot:AttachPolicy	AttachPolicy	arn:aws:iot: <i>region:account-id:thinggroup/thing-group-name</i> or arn:aws:iot: <i>region:account-id:cert/cert-id</i>
iot:AttachPrincipalPolicy	AttachPrincipalPolicy	arn:aws:iot: <i>region:account-id:cert/cert-id</i>
iot:AttachSecurityProfile	AttachSecurityProfile	arn:aws:iot: <i>region:account-id:securityprofile/security-profile-name</i> arn:aws:iot: <i>region:account-id:dimension/dimension-name</i>
iot:AttachThingPrincipal	AttachThingPrincipal	arn:aws:iot: <i>region:account-id:cert/cert-id</i>
iot:CancelCertificateTransfer	CancelCertificateTransfer	arn:aws:iot: <i>region:account-id:cert/cert-id</i>
		Note The AWS account specified in the ARN must be the account to which the certificate is being transferred.
iot:CancelJob	CancelJob	arn:aws:iot: <i>region:account-id:job/job-id</i>
iot:CancelJobExecution	CancelJobExecution	arn:aws:iot: <i>region:account-id:job/job-id</i> arn:aws:iot: <i>region:account-id:thing/thing-name</i>
iot:ClearDefaultAuthorizer	ClearDefaultAuthorizer	None
iot>CreateAuthorizer	CreateAuthorizer	arn:aws:iot: <i>region:account-id:authorizer/authorizer-function-name</i>
iot>CreateCertificateFromCsr	CreateCertificateFromCsr	
iot>CreateDimension	CreateDimension	arn:aws:iot: <i>region:account-id:dimension/dimension-name</i>
iot>CreateJob	CreateJob	arn:aws:iot: <i>region:account-id:job/job-id</i>

Policy actions	AWS IoT API	Resources
		arn:aws:iot: <i>region:account-id:thinggroup/thing-group-name</i> arn:aws:iot: <i>region:account-id:thing/thing-name</i> arn:aws:iot: <i>region:account-id:jobtemplate/job-template-id</i>
iot:CreateJobTemplate	CreateJobTemplate	arn:aws:iot: <i>region:account-id:job/job-id</i> arn:aws:iot: <i>region:account-id:jobtemplate/job-template-id</i>
iot:CreateKeysAndCertificate	CreateKeysAndCertificate	
iot:CreatePolicy	CreatePolicy	*
iot:CreatePolicyVersion	CreatePolicyVersion	arn:aws:iot: <i>region:account-id:policy/policy-name</i> Note This must be an AWS IoT policy, not an IAM policy.
iot:CreateRoleAlias	CreateRoleAlias	(parameter: roleAlias) arn:aws:iot: <i>region:account-id:rolealias/rolealias-name</i>
iot:CreateSecurityProfile	CreateSecurityProfile	arn:aws:iot: <i>region:account-id:securityprofile/security-profile-name</i> arn:aws:iot: <i>region:account-id:dimension/dimension-name</i>
iot:CreateThing	CreateThing	arn:aws:iot: <i>region:account-id:thing/thing-name</i>
iot:CreateThingGroup	CreateThingGroup	arn:aws:iot: <i>region:account-id:thinggroup/thing-group-name</i> for group being created and for parent group, if used
iot:CreateThingType	CreateThingType	arn:aws:iot: <i>region:account-id:thingtype/thing-type-name</i>
iot:CreateTopicRule	CreateTopicRule	arn:aws:iot: <i>region:account-id:rule/rule-name</i>
iot:DeleteAuthorizer	DeleteAuthorizer	arn:aws:iot: <i>region:account-id:authorizer/authorizer-name</i>
iot:DeleteCACertificate	DeleteCACertificate	arn:aws:iot: <i>region:account-id:cacert/cert-id</i>
iot:DeleteCertificate	DeleteCertificate	arn:aws:iot: <i>region:account-id:cert/cert-id</i>
iot:DeleteDimension	DeleteDimension	arn:aws:iot: <i>region:account-id:dimension/dimension-name</i>
iot:DeleteJob	DeleteJob	arn:aws:iot: <i>region:account-id:job/job-id</i>
iot:DeleteJobTemplate	DeleteJobTemplate	arn:aws:iot: <i>region:account-id:job/job-template-id</i>

Policy actions	AWS IoT API	Resources
iot:DeleteJobExecution	DeleteJobExecution	arn:aws:iot: <i>region:account-id:job/job-id</i> arn:aws:iot: <i>region:account-id:thing/thing-name</i>
iot:DeletePolicy	DeletePolicy	arn:aws:iot: <i>region:account-id:policy/policy-name</i>
iot:DeletePolicyVersion	DeletePolicyVersion	arn:aws:iot: <i>region:account-id:policy/policy-name</i>
iot:DeleteRegistrationCode	DeleteRegistrationCode	
iot:DeleteRoleAlias	DeleteRoleAlias	arn:aws:iot: <i>region:account-id:rolealias/rolealias-name</i>
iot:DeleteSecurityProfile	DeleteSecurityProfile	arn:aws:iot: <i>region:account-id:securityprofile/security-profile-name</i> arn:aws:iot: <i>region:account-id:dimension/dimension-name</i>
iot:DeleteThing	DeleteThing	arn:aws:iot: <i>region:account-id:thing/thing-name</i>
iot:DeleteThingGroup	DeleteThingGroup	arn:aws:iot: <i>region:account-id:thinggroup/thing-group-name</i>
iot:DeleteThingType	DeleteThingType	arn:aws:iot: <i>region:account-id:thingtype/thing-type-name</i>
iot:DeleteTopicRule	DeleteTopicRule	arn:aws:iot: <i>region:account-id:rule/rule-name</i>
iot:DeleteV2LoggingLevel	DeleteV2LoggingLevel	arn:aws:iot: <i>region:account-id:thinggroup/thing-group-name</i>
iot:DeprecateThingType	DeprecateThingType	arn:aws:iot: <i>region:account-id:thingtype/thing-type-name</i>
iot:DescribeAuthorizedFunction	DescribeAuthorizer	arn:aws:iot: <i>region:account-id:authorizer/authorizer-function-name</i> (parameter: authorizerName) none
iot:DescribeCACertificate	DescribeCACertificate	arn:aws:iot: <i>region:account-id:cacert/cert-id</i>
iot:DescribeCertificate	DescribeCertificate	arn:aws:iot: <i>region:account-id:cert/cert-id</i>
iot:DescribeDefaultAuthThing	DescribeDefaultAuthThing	None
iot:DescribeEndpoint	DescribeEndpoint	*
iot:DescribeEventConfiguration	DescribeEventConfiguration	None
iot:DescribeIndex	DescribeIndex	arn:aws:iot: <i>region:account-id:index/index-name</i>
iot:DescribeJob	DescribeJob	arn:aws:iot: <i>region:account-id:job/job-id</i>
iot:DescribeJobExecution	DescribeJobExecution	None

Policy actions	AWS IoT API	Resources
iot:DescribeJobTemplate	DescribeJobTemplate	arn:aws:iot: <i>region:account-id:job/job-template-id</i>
iot:DescribeRoleAlias	DescribeRoleAlias	arn:aws:iot: <i>region:account-id:rolealias/role-alias-name</i>
iot:DescribeThing	DescribeThing	arn:aws:iot: <i>region:account-id:thing/thing-name</i>
iot:DescribeThingGroup	DescribeThingGroup	arn:aws:iot: <i>region:account-id:thinggroup/thing-group-name</i>
iot:DescribeThingRegistrationTask	DescribeThingRegistrationTask	None
iot:DescribeThingType	DescribeThingType	arn:aws:iot: <i>region:account-id:thingtype/thing-type-name</i>
iot:DetachPolicy	DetachPolicy	arn:aws:iot: <i>region:account-id:cert/cert-id</i> or arn:aws:iot: <i>region:account-id:thinggroup/thing-group-name</i>
iot:DetachPrincipalPolicy	DetachPrincipalPolicy	arn:aws:iot: <i>region:account-id:cert/cert-id</i>
iot:DetachSecurityProfile	DetachSecurityProfile	arn:aws:iot: <i>region:account-id:securityprofile/security-profile-name</i> arn:aws:iot: <i>region:account-id:dimension/dimension-name</i>
iot:DetachThingPrincipal	DetachThingPrincipal	arn:aws:iot: <i>region:account-id:cert/cert-id</i>
iot:DisableTopicRule	DisableTopicRule	arn:aws:iot: <i>region:account-id:rule/rule-name</i>
iot:EnableTopicRule	EnableTopicRule	arn:aws:iot: <i>region:account-id:rule/rule-name</i>
iot:GetEffectivePolicies	GetEffectivePolicies	arn:aws:iot: <i>region:account-id:cert/cert-id</i>
iot:GetIndexingConfiguration	GetIndexingConfiguration	None
iot:GetJobDocument	GetJobDocument	arn:aws:iot: <i>region:account-id:job/job-id</i>
iot:GetLoggingOptions	GetLoggingOptions	*
iot:GetPolicy	GetPolicy	arn:aws:iot: <i>region:account-id:policy/policy-name</i>
iot:GetPolicyVersion	GetPolicyVersion	arn:aws:iot: <i>region:account-id:policy/policy-name</i>
iot:GetRegistrationCode	GetRegistrationCode	*
iot:GetTopicRule	GetTopicRule	arn:aws:iot: <i>region:account-id:rule/rule-name</i>

Policy actions	AWS IoT API	Resources
iot:ListAttachedPolicies	ListAttachedPolicies	arn:aws:iot: <i>region:account-id:thinggroup/thing-group-name</i> or arn:aws:iot: <i>region:account-id:cert/cert-id</i>
iot:ListAuthorizers	ListAuthorizers	None
iot:ListCACertificates	ListCACertificates	*
iot:ListCertificates	ListCertificates	*
iot:ListCertificatesByCA	ListCertificatesByCA	*
iot:ListIndices	ListIndices	None
iot:ListJobExecutions	ListJobExecutions	None
iot:ListJobExecutions	ListJobExecutions	None
iot:ListJobs	ListJobs	arn:aws:iot: <i>region:account-id:thinggroup/thing-group-name</i> if thingGroupName parameter used
iot:ListJobTemplates	ListJobs	None
iot:ListOutgoingCertificates	ListOutgoingCertificates	*
iot:ListPolicies	ListPolicies	*
iot:ListPolicyPrincipals	ListPolicyPrincipals	arn:aws:iot: <i>region:account-id:policy/policy-name</i>
iot:ListPolicyVersions	ListPolicyVersions	arn:aws:iot: <i>region:account-id:policy/policy-name</i>
iot:ListPrincipalPolicies	ListPrincipalPolicies	arn:aws:iot: <i>region:account-id:cert/cert-id</i>
iot:ListPrincipalThings	ListPrincipalThings	arn:aws:iot: <i>region:account-id:cert/cert-id</i>
iot:ListRoleAliases	ListRoleAliases	None
iot:ListTargetsForPolicy	ListTargetsForPolicy	arn:aws:iot: <i>region:account-id:policy/policy-name</i>
iot:ListThingGroups	ListThingGroups	None
iot:ListThingGroupsForThing	ListThingGroupsForThing	arn:aws:iot: <i>region:account-id:thing/thing-name</i>
iot:ListThingPrincipals	ListThingPrincipals	arn:aws:iot: <i>region:account-id:thing/thing-name</i>
iot:ListThingRegistrations	ListThingRegistrations	arn:aws:iot: <i>region:account-id:task/task-report</i>
iot:ListThingRegistrations	ListThingRegistrations	arn:aws:iot: <i>region:account-id:task/task</i>
iot:ListThingTypes	ListThingTypes	*
iot:ListThings	ListThings	*

Policy actions	AWS IoT API	Resources
iot:ListThingsInThingGroup	ListThingsInThingGroup	arn:aws:iot: region:account-id:thinggroup/thing-group-name
iot:ListTopicRules	ListTopicRules	*
iot:ListV2LoggingLevel	ListV2LoggingLevels	None
iot:RegisterCACertificate	RegisterCACertificate*	
iot:RegisterCertificate	RegisterCertificate	*
iot:RegisterThing	RegisterThing	None
iot:RejectCertificateTransfer	RejectCertificateTransfer	arn:aws:iot: region:account-id:cert/cert-id
iot:RemoveThingFromThingGroup	RemoveThingFromThingGroup	arn:aws:iot: region:account-id:thinggroup/thing-group-name arn:aws:iot: region:account-id:thing/thing-name
iot:ReplaceTopicRule	ReplaceTopicRule	arn:aws:iot: region:account-id:rule/rule-name
iot:SearchIndex	SearchIndex	arn:aws:iot: region:account-id:index/index-id
iot:SetDefaultAuthorizer	SetDefaultAuthorizer	arn:aws:iot: region:account-id:authorizer/authorizer-function-name
iot:SetDefaultPolicyVersion	SetDefaultPolicyVersion	arn:aws:iot: region:account-id:policy/policy-name
iot:SetLoggingOptions	SetLoggingOptions	arn:aws:iot: region:account-id:role/role-name
iot:SetV2LoggingLevel	SetV2LoggingLevel	arn:aws:iot: region:account-id:thinggroup/thing-group-name
iot:SetV2LoggingOptions	SetV2LoggingOptions	arn:aws:iot: region:account-id:role/role-name
iot:StartThingRegistrationTask	StartThingRegistrationTask	None
iot:StopThingRegistrationTask	StopThingRegistrationTask	None
iot:TestAuthorization	TestAuthorization	arn:aws:iot: region:account-id:cert/cert-id
iot:TestInvokeAuthorizer	TestInvokeAuthorizer	None
iot:TransferCertificate	TransferCertificate	arn:aws:iot: region:account-id:cert/cert-id
iot:UpdateAuthorizer	UpdateAuthorizer	arn:aws:iot: region:account-id:authorizerfunction/authorizer-function-name
iot:UpdateCACertificate	UpdateCACertificate	arn:aws:iot: region:account-id:cacert/cert-id
iot:UpdateCertificate	UpdateCertificate	arn:aws:iot: region:account-id:cert/cert-id
iot:UpdateDimension	UpdateDimension	arn:aws:iot: region:account-id:dimension/dimension-name
iot:UpdateEventConfiguration	UpdateEventConfiguration	None
iot:UpdateIndexingConfiguration	UpdateIndexingConfiguration	None

Policy actions	AWS IoT API	Resources
iot:UpdateRoleAlias	UpdateRoleAlias	arn:aws:iot: <i>region:account-id</i> :rolealias/ <i>role-alias-name</i>
iot:UpdateSecurityProfile	UpdateSecurityProfile	arn:aws:iot: <i>region:account-id</i> :securityprofile/ <i>security-profile-name</i> arn:aws:iot: <i>region:account-id</i> :dimension/ <i>dimension-name</i>
iot:UpdateThing	UpdateThing	arn:aws:iot: <i>region:account-id</i> :thing/ <i>thing-name</i>
iot:UpdateThingGroup	UpdateThingGroup	arn:aws:iot: <i>region:account-id</i> :thinggroup/ <i>thing-group-name</i>
iot:UpdateThingGroups	UpdateThingGroups	arn:aws:iot: <i>region:account-id</i> :thing/ <i>thing-name</i>

Policy actions in AWS IoT use the following prefix before the action: `iot:.` For example, to grant someone permission to list all IoT things registered in their AWS account with the `ListThings` API, you include the `iot>ListThings` action in their policy. Policy statements must include either an `Action` or `NotAction` element. AWS IoT defines its own set of actions that describe tasks that you can perform with this service.

To specify multiple actions in a single statement, separate them with commas as follows:

```
"Action": [
    "ec2:action1",
    "ec2:action2"
```

You can specify multiple actions using wildcards (*). For example, to specify all actions that begin with the word `Describe`, include the following action:

```
"Action": "iot:Describe*"
```

To see a list of AWS IoT actions, see [Actions Defined by AWS IoT](#) in the *IAM User Guide*.

Device Advisor actions

The following table lists the IAM IoT Device Advisor actions, the associated AWS IoT Device Advisor API, and the resource the action manipulates.

Policy actions	AWS IoT API	Resources
iotdeviceadvisor:CreateSuiteDefinition	CreateSuiteDefinition	None
iotdeviceadvisor:DeleteSuiteDefinition	DeleteSuiteDefinition	arn:aws:iotdeviceadvisor: <i>region:account-id</i> :suitedefinition/ <i>suite-definition-id</i>
iotdeviceadvisor:GetSuiteDefinition	GetSuiteDefinition	arn:aws:iotdeviceadvisor: <i>region:account-id</i> :suitedefinition/ <i>suite-definition-id</i>
iotdeviceadvisor:GetSuiteRun	GetSuiteRun	arn:aws:iotdeviceadvisor: <i>region:account-id</i> :suitedefinition/ <i>suite-run-id</i>
iotdeviceadvisor:GetSuiteReport	GetSuiteReport	arn:aws:iotdeviceadvisor: <i>region:account-id</i> :suiterun/ <i>suite-definition-id</i> / <i>suite-run-id</i>

Policy actions	AWS IoT API	Resources
iotdeviceadvisor>ListSuiteDefinitions	ListSuiteDefinitions	None
iotdeviceadvisor>ListSuiteRuns	ListSuiteRuns	arn:aws:iotdeviceadvisor: <i>region:account-id:suitedefinition/suite-definition-id</i>
iotdeviceadvisor>ListTagsForResource	ListTagsForResource	arn:aws:iotdeviceadvisor: <i>region:account-id:suitedefinition/suite-definition-id</i> arn:aws:iotdeviceadvisor: <i>region:account-id:suiterun/suite-definition-id/suite-run-id</i>
iotdeviceadvisor>StartSuiteRun	StartSuiteRun	arn:aws:iotdeviceadvisor: <i>region:account-id:suitedefinition/suite-definition-id</i>
iotdeviceadvisor>TagResource	TagResource	arn:aws:iotdeviceadvisor: <i>region:account-id:suitedefinition/suite-definition-id</i> arn:aws:iotdeviceadvisor: <i>region:account-id:suiterun/suite-definition-id/suite-run-id</i>
iotdeviceadvisor>UntagResource	UntagResource	arn:aws:iotdeviceadvisor: <i>region:account-id:suitedefinition/suite-definition-id</i> arn:aws:iotdeviceadvisor: <i>region:account-id:suiterun/suite-definition-id/suite-run-id</i>
iotdeviceadvisor>UpdateSuiteDefinition	UpdateSuiteDefinition	arn:aws:iotdeviceadvisor: <i>region:account-id:suitedefinition/suite-definition-id</i>
iotdeviceadvisor>StopSuiteRun	StopSuiteRun	arn:aws:iotdeviceadvisor: <i>region:account-id:suiterun/suite-definition-id/suite-run-id</i>

Policy actions in AWS IoT Device Advisor use the following prefix before the action: `iotdeviceadvisor:`. For example, to grant someone permission to list all suite definitions registered in their AWS account with the `ListSuiteDefinitions` API, you include the `iotdeviceadvisor>ListSuiteDefinitions` action in their policy.

Resources

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The Resource JSON policy element specifies the object or objects to which the action applies. Statements must include either a Resource or a NotResource element. As a best practice, specify a resource using its [Amazon Resource Name \(ARN\)](#). You can do this for actions that support a specific resource type, known as *resource-level permissions*.

For actions that don't support resource-level permissions, such as listing operations, use a wildcard (*) to indicate that the statement applies to all resources.

```
"Resource": "*"
```

AWS IoT resources

Policy actions	AWS IoT API	Resources
iot:AcceptCertificateTransfer	AcceptCertificateTransfer	arn:aws:iot: <i>region:account-id:cert/cert-id</i>

Policy actions	AWS IoT API	Resources
		Note The AWS account specified in the ARN must be the account to which the certificate is being transferred.
iot:AddThingToThingGroup	AddThingToThingGroup	<code>arn:aws:iot:<i>region:account-id:thinggroup/thing-group-name</i></code> <code>arn:aws:iot:<i>region:account-id:thing/thing-name</i></code>
iot:AssociateTargetsWithThingGroup	AssociateTargetsWithThingGroup	None
iot:AttachPolicy	AttachPolicy	<code>arn:aws:iot:<i>region:account-id:thinggroup/thing-group-name</i></code> or <code>arn:aws:iot:<i>region:account-id:cert/cert-id</i></code>
iot:AttachPrincipalPolicy	AttachPrincipalPolicy	<code>arn:aws:iot:<i>region:account-id:cert/cert-id</i></code>
iot:AttachThingPrincipal	AttachThingPrincipal	<code>arn:aws:iot:<i>region:account-id:cert/cert-id</i></code>
iot:CancelCertificateTransfer	CancelCertificateTransfer	<code>arn:aws:iot:<i>region:account-id:cert/cert-id</i></code>
		Note The AWS account specified in the ARN must be the account to which the certificate is being transferred.
iot:CancelJob	CancelJob	<code>arn:aws:iot:<i>region:account-id:job/job-id</i></code>
iot:CancelJobExecution	CancelJobExecution	<code>arn:aws:iot:<i>region:account-id:job/job-id</i></code> <code>arn:aws:iot:<i>region:account-id:thing/thing-name</i></code>
iot:ClearDefaultAuthorizer	ClearDefaultAuthorizer	None
iot>CreateAuthorizer	CreateAuthorizer	<code>arn:aws:iot:<i>region:account-id:authorizer/authorizer-function-name</i></code>
iot>CreateCertificateFromCsr	CreateCertificateFromCsr	
iot>CreateJob	CreateJob	<code>arn:aws:iot:<i>region:account-id:job/job-id</i></code> <code>arn:aws:iot:<i>region:account-id:thinggroup/thing-group-name</i></code> <code>arn:aws:iot:<i>region:account-id:thing/thing-name</i></code> <code>arn:aws:iot:<i>region:account-id:jobtemplate/job-template-id</i></code>
iot>CreateJobTemplate	CreateJobTemplate	<code>arn:aws:iot:<i>region:account-id:job/job-id</i></code> <code>arn:aws:iot:<i>region:account-id:jobtemplate/job-template-id</i></code>
iot>CreateKeysAndCertificate	CreateKeysAndCertificate	
iot>CreatePolicy	CreatePolicy	*

Policy actions	AWS IoT API	Resources
CreatePolicyVersion	iot:CreatePolicyVersion	arn:aws:iot: <i>region:account-id:policy/policy-name</i> Note This must be an AWS IoT policy, not an IAM policy.
iot:CreateRoleAlias	CreateRoleAlias	(parameter: roleAlias) arn:aws:iot: <i>region:account-id:rolealias/role-alias-name</i>
iot:CreateThing	CreateThing	arn:aws:iot: <i>region:account-id:thing/thing-name</i>
iot:CreateThingGroup	CreateThingGroup	arn:aws:iot: <i>region:account-id:thinggroup/thing-group-name</i> for group being created and for parent group, if used
iot:CreateThingType	CreateThingType	arn:aws:iot: <i>region:account-id:thingtype/thing-type-name</i>
iot:CreateTopicRule	CreateTopicRule	arn:aws:iot: <i>region:account-id:rule/rule-name</i>
iot:DeleteAuthorizer	DeleteAuthorizer	arn:aws:iot: <i>region:account-id:authorizer/authorizer-name</i>
iot:DeleteCACertificate	DeleteCACertificate	arn:aws:iot: <i>region:account-id:cacert/cert-id</i>
iot:DeleteCertificate	DeleteCertificate	arn:aws:iot: <i>region:account-id:cert/cert-id</i>
iot:DeleteJob	DeleteJob	arn:aws:iot: <i>region:account-id:job/job-id</i>
iot:DeleteJobExecution	DeleteJobExecution	arn:aws:iot: <i>region:account-id:job/job-id</i> arn:aws:iot: <i>region:account-id:thing/thing-name</i>
iot:DeleteJobTemplate	DeleteJobTemplate	arn:aws:iot: <i>region:account-id:jobtemplate/job-template-id</i>
iot:DeletePolicy	DeletePolicy	arn:aws:iot: <i>region:account-id:policy/policy-name</i>
iot:DeletePolicyVersion	DeletePolicyVersion	arn:aws:iot: <i>region:account-id:policy/policy-name</i>
iot:DeleteRegistrationCode	DeleteRegistrationCode	
iot:DeleteRoleAlias	DeleteRoleAlias	arn:aws:iot: <i>region:account-id:rolealias/role-alias-name</i>
iot:DeleteThing	DeleteThing	arn:aws:iot: <i>region:account-id:thing/thing-name</i>
iot:DeleteThingGroup	DeleteThingGroup	arn:aws:iot: <i>region:account-id:thinggroup/thing-group-name</i>
iot:DeleteThingType	DeleteThingType	arn:aws:iot: <i>region:account-id:thingtype/thing-type-name</i>
iot:DeleteTopicRule	DeleteTopicRule	arn:aws:iot: <i>region:account-id:rule/rule-name</i>

Policy actions	AWS IoT API	Resources
iot:DeleteV2LoggingLevel	DeleteV2LoggingLevel	arn:aws:iot: region:account-id:thinggroup/thing-group-name
iot:DeprecateThingType	DeprecateThingType	arn:aws:iot: region:account-id:thingtype/thing-type-name
iot:DescribeAuthorizedFunction	DescribeAuthorizer	arn:aws:iot: region:account-id:authorizer/authorizer-function-name (parameter: authorizerName) none
iot:DescribeCACertificate	DescribeCACertificate	arn:aws:iot: region:account-id:cacert/cert-id
iot:DescribeCertificate	DescribeCertificate	arn:aws:iot: region:account-id:cert/cert-id
iot:DescribeDefaultAuthorizer	DescribeDefaultAuthorizer	
iot:DescribeEndpoint	DescribeEndpoint	*
iot:DescribeEventConfigurations	DescribeEventConfigurations	
iot:DescribeIndex	DescribeIndex	arn:aws:iot: region:account-id:index/index-name
iot:DescribeJob	DescribeJob	arn:aws:iot: region:account-id:job/job-id
iot:DescribeJobExecution	DescribeJobExecution	None
iot:DescribeJobTemplate	DescribeJobTemplate	arn:aws:iot: region:account-id:jobtemplate/job-template-id
iot:DescribeRoleAlias	DescribeRoleAlias	arn:aws:iot: region:account-id:rolealias/role-alias-name
iot:DescribeThing	DescribeThing	arn:aws:iot: region:account-id:thing/thing-name
iot:DescribeThingGroup	DescribeThingGroup	arn:aws:iot: region:account-id:thinggroup/thing-group-name
iot:DescribeThingRegistrationTask	DescribeThingRegistrationTask	None
iot:DescribeThingType	DescribeThingType	arn:aws:iot: region:account-id:thingtype/thing-type-name
iot:DetachPolicy	DetachPolicy	arn:aws:iot: region:account-id:cert/cert-id or arn:aws:iot: region:account-id:thinggroup/thing-group-name
iot:DetachPrincipalPolicy	DetachPrincipalPolicy	arn:aws:iot: region:account-id:cert/cert-id
iot:DetachThingPrincipal	DetachThingPrincipal	arn:aws:iot: region:account-id:cert/cert-id
iot:DisableTopicRule	DisableTopicRule	arn:aws:iot: region:account-id:rule/rule-name
iot:EnableTopicRule	EnableTopicRule	arn:aws:iot: region:account-id:rule/rule-name
iot:GetEffectivePolicies	GetEffectivePolicies	arn:aws:iot: region:account-id:cert/cert-id

Policy actions	AWS IoT API	Resources
iot:GetIndexingConfiguration	GetIndexingConfiguration	None
iot:GetJobDocument	GetJobDocument	arn:aws:iot: <i>region:account-id</i> :job/ <i>job-id</i>
iot:GetLoggingOptions	GetLoggingOptions	*
iot:GetPolicy	GetPolicy	arn:aws:iot: <i>region:account-id</i> :policy/ <i>policy-name</i>
iot:GetPolicyVersion	GetPolicyVersion	arn:aws:iot: <i>region:account-id</i> :policy/ <i>policy-name</i>
iot:GetRegistrationCode	GetRegistrationCode	*
iot:GetTopicRule	GetTopicRule	arn:aws:iot: <i>region:account-id</i> :rule/ <i>rule-name</i>
iot>ListAttachedPolicies	ListAttachedPolicies	arn:aws:iot: <i>region:account-id</i> :thinggroup/ <i>thing-group-name</i> or arn:aws:iot: <i>region:account-id</i> :cert/ <i>cert-id</i>
iot>ListAuthorizers	ListAuthorizers	None
iot>ListCACertificates	ListCACertificates	*
iot>ListCertificates	ListCertificates	*
iot>ListCertificatesByCA	ListCertificatesByCA	*
iot>ListIndices	ListIndices	None
iot>ListJobExecutions	ListJobExecutions	None
iot>ListJobExecutions	ListJobExecutions	None
iot>ListJobs	ListJobs	arn:aws:iot: <i>region:account-id</i> :thinggroup/ <i>thing-group-name</i> if thingGroupName parameter used
iot>ListJobTemplates	ListJobTemplates	None
iot>ListOutgoingCertificates	ListOutgoingCertificates	*
iot>ListPolicies	ListPolicies	*
iot>ListPolicyPrincipals	ListPolicyPrincipals	arn:aws:iot: <i>region:account-id</i> :policy/ <i>policy-name</i>
iot>ListPolicyVersions	ListPolicyVersions	arn:aws:iot: <i>region:account-id</i> :policy/ <i>policy-name</i>
iot>ListPrincipalPolicies	ListPrincipalPolicies	arn:aws:iot: <i>region:account-id</i> :cert/ <i>cert-id</i>
iot>ListPrincipalThings	ListPrincipalThings	arn:aws:iot: <i>region:account-id</i> :cert/ <i>cert-id</i>
iot>ListRoleAliases	ListRoleAliases	None

Policy actions	AWS IoT API	Resources
iot:ListTargetsForPolicy	listTargetsForPolicy	arn:aws:iot: <i>region:account-id:policy/policy-name</i>
iot>ListThingGroups	ListThingGroups	None
iot>ListThingGroupsForThing	listThingGroupsForThing	arn:aws:iot: <i>region:account-id:thing/thing-name</i>
iot>ListThingPrincipals	listThingPrincipals	arn:aws:iot: <i>region:account-id:thing/thing-name</i>
iot>ListThingRegistrationTasks	listThingRegistrationTasks	None
iot>ListThingRegistrations	listThingRegistrations	None
iot>ListThingTypes	ListThingTypes	*
iot>ListThings	ListThings	*
iot>ListThingsInThingGroup	listThingsInThingGroup	arn:aws:iot: <i>region:account-id:thinggroup/thing-group-name</i>
iot>ListTopicRules	ListTopicRules	*
iot>ListV2LoggingLevels	listV2LoggingLevels	None
iot:RegisterCACertificate	registerCACertificate	*
iot:RegisterCertificate	registerCertificate	*
iot:RegisterThing	RegisterThing	None
iot:RejectCertificateTransfer	rejectCertificateTransfer	arn:aws:iot: <i>region:account-id:cert/cert-id</i>
iot:RemoveThingFromThingGroup	removeThingFromThingGroup	arn:aws:iot: <i>region:account-id:thinggroup/thing-group-name</i> arn:aws:iot: <i>region:account-id:thing/thing-name</i>
iot:ReplaceTopicRule	ReplaceTopicRule	arn:aws:iot: <i>region:account-id:rule/rule-name</i>
iot:SearchIndex	SearchIndex	arn:aws:iot: <i>region:account-id:index/index-id</i>
iot:SetDefaultAuthorizer	setDefaultAuthorizer	arn:aws:iot: <i>region:account-id:authorizer/authorizer-function-name</i>
iot:SetDefaultPolicyVersion	setDefaultPolicyVersion	arn:aws:iot: <i>region:account-id:policy/policy-name</i>
iot:SetLoggingOptions	setLoggingOptions	arn:aws:iot: <i>region:account-id:role/role-name</i>
iot:SetV2LoggingLevel	setV2LoggingLevel	arn:aws:iot: <i>region:account-id:thinggroup/thing-group-name</i>
iot:SetV2LoggingOptions	setV2LoggingOptions	arn:aws:iot: <i>region:account-id:role/role-name</i>
iot:StartThingRegistrationTask	startThingRegistrationTask	None
iot:StopThingRegistrationTask	stopThingRegistrationTask	None
iot:TestAuthorization	testAuthorization	arn:aws:iot: <i>region:account-id:cert/cert-id</i>
iot:TestInvokeAuthorizer	testInvokeAuthorizer	None

Policy actions	AWS IoT API	Resources
iot:TransferCertificate	TransferCertificate	arn:aws:iot: <i>region:account-id</i> :cert/ <i>cert-id</i>
iot:UpdateAuthorizer	UpdateAuthorizer	arn:aws:iot: <i>region:account-id</i> :authorizerfunction/ <i>authorizer-function-name</i>
iot:UpdateCACertificate	UpdateCACertificate	arn:aws:iot: <i>region:account-id</i> :cacert/ <i>cert-id</i>
iot:UpdateCertificate	UpdateCertificate	arn:aws:iot: <i>region:account-id</i> :cert/ <i>cert-id</i>
iot:UpdateEventConfiguration	UpdateEventConfiguration	No resources
iot:UpdateIndexingConfiguration	UpdateIndexingConfiguration	No resources
iot:UpdateRoleAlias	UpdateRoleAlias	arn:aws:iot: <i>region:account-id</i> :rolealias/ <i>role-alias-name</i>
iot:UpdateThing	UpdateThing	arn:aws:iot: <i>region:account-id</i> :thing/ <i>thing-name</i>
iot:UpdateThingGroup	UpdateThingGroup	arn:aws:iot: <i>region:account-id</i> :thinggroup/ <i>thing-group-name</i>
iot:UpdateThingGroupsForThing	UpdateThingGroupsForThing	arn:aws:iot: <i>region:account-id</i> :thing/ <i>thing-name</i>

For more information about the format of ARNs, see [Amazon Resource Names \(ARNs\) and AWS Service Namespaces](#).

Some AWS IoT actions, such as those for creating resources, cannot be performed on a specific resource. In those cases, you must use the wildcard (*).

```
"Resource": "*"
```

To see a list of AWS IoT resource types and their ARNs, see [Resources Defined by AWS IoT](#) in the *IAM User Guide*. To learn with which actions you can specify the ARN of each resource, see [Actions Defined by AWS IoT](#).

Device Advisor resources

To define resource-level restrictions for AWS IoT Device Advisor IAM policies, use the following resource ARN formats for suite definitions and suite runs.

Suite definition resource ARN format

```
arn:aws:iotdeviceadvisor:region:account-id:suitedefinition/suite-definition-id
```

Suite run resource ARN format

```
arn:aws:iotdeviceadvisor:region:account-id:suiterun/suite-definition-id/suite-run-id
```

Condition keys

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The Condition element (or Condition block) lets you specify conditions in which a statement is in effect. The Condition element is optional. You can create conditional expressions that use [condition operators](#), such as equals or less than, to match the condition in the policy with values in the request.

If you specify multiple Condition elements in a statement, or multiple keys in a single Condition element, AWS evaluates them using a logical AND operation. If you specify multiple values for a single condition key, AWS evaluates the condition using a logical OR operation. All of the conditions must be met before the statement's permissions are granted.

You can also use placeholder variables when you specify conditions. For example, you can grant an IAM user permission to access a resource only if it is tagged with their IAM user name. For more information, see [IAM policy elements: variables and tags](#) in the *IAM User Guide*.

AWS supports global condition keys and service-specific condition keys. To see all AWS global condition keys, see [AWS global condition context keys](#) in the *IAM User Guide*.

AWS IoT defines its own set of condition keys and also supports using some global condition keys. To see all AWS global condition keys, see [AWS Global Condition Context Keys](#) in the *IAM User Guide*.

AWS IoT condition keys

AWS IoT condition keys	Description	Type
<code>aws:RequestTag/\${tag-key}</code>	A tag key that is present in the request that the user makes to AWS IoT.	String
<code>aws:ResourceTag/\${tag-key}</code>	The tag key component of a tag attached to an AWS IoT resource.	String
<code>aws:TagKeys</code>	The list of all the tag key names associated with the resource in the request.	String

To see a list of AWS IoT condition keys, see [Condition Keys for AWS IoT](#) in the *IAM User Guide*. To learn with which actions and resources you can use a condition key, see [Actions Defined by AWS IoT](#).

Examples

To view examples of AWS IoT identity-based policies, see [AWS IoT identity-based policy examples \(p. 385\)](#).

AWS IoT resource-based policies

Resource-based policies are JSON policy documents that specify what actions a specified principal can perform on the AWS IoT resource and under what conditions.

AWS IoT does not support IAM resource-based policies. It does, however, support AWS IoT resource-based policies. For more information, see [AWS IoT Core policies \(p. 314\)](#).

Authorization based on AWS IoT tags

You can attach tags to AWS IoT resources or pass tags in a request to AWS IoT. To control access based on tags, you provide tag information in the [condition element](#) of a policy using the `iot:ResourceTag/key-name`, `aws:RequestTag/key-name`, or `aws:TagKeys` condition keys. For

more information, see [Using tags with IAM policies \(p. 274\)](#). For more information about tagging AWS IoT resources, see [Tagging your AWS IoT resources \(p. 273\)](#).

To view an example identity-based policy for limiting access to a resource based on the tags on that resource, see [Viewing AWS IoT resources based on tags \(p. 387\)](#).

AWS IoT IAM roles

An [IAM role](#) is an entity within your AWS account that has specific permissions.

Using temporary credentials with AWS IoT

You can use temporary credentials to sign in with federation, assume an IAM role, or to assume a cross-account role. You obtain temporary security credentials by calling AWS STS API operations such as [AssumeRole](#) or [GetFederationToken](#).

AWS IoT supports using temporary credentials.

Service-linked roles

[Service-linked roles](#) allow AWS services to access resources in other services to complete an action on your behalf. Service-linked roles appear in your IAM account and are owned by the service. An IAM administrator can view but not edit the permissions for service-linked roles.

AWS IoT does not support service-linked roles.

Service roles

This feature allows a service to assume a [service role](#) on your behalf. This role allows the service to access resources in other services to complete an action on your behalf. Service roles appear in your IAM account and are owned by the account. This means that an IAM administrator can change the permissions for this role. However, doing so might break the functionality of the service.

AWS IoT identity-based policy examples

By default, IAM users and roles don't have permission to create or modify AWS IoT resources. They also can't perform tasks using the AWS Management Console, AWS CLI, or AWS API. An IAM administrator must create IAM policies that grant users and roles permission to perform specific API operations on the specified resources they need. The administrator must then attach those policies to the IAM users or groups that require those permissions.

To learn how to create an IAM identity-based policy using these example JSON policy documents, see [Creating Policies on the JSON Tab](#) in the *IAM User Guide*.

Topics

- [Policy best practices \(p. 385\)](#)
- [Using the AWS IoT console \(p. 386\)](#)
- [Allow users to view their own permissions \(p. 386\)](#)
- [Viewing AWS IoT resources based on tags \(p. 387\)](#)
- [Viewing AWS IoT Device Advisor resources based on tags \(p. 387\)](#)

Policy best practices

Identity-based policies are very powerful. They determine whether someone can create, access, or delete AWS IoT resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started using AWS managed policies** – To start using AWS IoT quickly, use AWS managed policies to give your employees the permissions they need. These policies are already available in your account and are maintained and updated by AWS. For more information, see [Get started using permissions with AWS managed policies](#) in the *IAM User Guide*.
 - **Grant least privilege** – When you create custom policies, grant only the permissions required to perform a task. Start with a minimum set of permissions and grant additional permissions as necessary. Doing so is more secure than starting with permissions that are too lenient and then trying to tighten them later. For more information, see [Grant least privilege](#) in the *IAM User Guide*.
 - **Enable MFA for sensitive operations** – For extra security, require IAM users to use multi-factor authentication (MFA) to access sensitive resources or API operations. For more information, see [Using multi-factor authentication \(MFA\) in AWS](#) in the *IAM User Guide*.
 - **Use policy conditions for extra security** – To the extent that it's practical, define the conditions under which your identity-based policies allow access to a resource. For example, you can write conditions to specify a range of allowable IP addresses that a request must come from. You can also write conditions to allow requests only within a specified date or time range, or to require the use of SSL or MFA. For more information, see [IAM JSON policy elements: Condition](#) in the *IAM User Guide*.

Using the AWS IoT console

To access the AWS IoT console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the AWS IoT resources in your AWS account. If you create an identity-based policy that is more restrictive than the minimum required permissions, the console won't function as intended for entities (IAM users or roles) with that policy.

To ensure that those entities can still use the AWS IoT console, also attach the following AWS managed policy to the entities: `AWSIoTFullAccess`. For more information, see [Adding Permissions to a User](#) in the *IAM User Guide*.

You don't need to allow minimum console permissions for users that are making calls only to the AWS CLI or the AWS API. Instead, allow access to only the actions that match the API operation that you're trying to perform.

Allow users to view their own permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam>ListGroupsForUser",
                "iam>ListAttachedUserPolicies",
                "iam>ListUserPolicies",
                "iam GetUser"
            ],
            "Resource": [ "arn:aws:iam::*:user/${aws:username}" ]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetRolePolicy",
                "iam:GetUserPolicy",
                "iam:GetPolicyVersion"
            ],
            "Resource": [ "arn:aws:iam::aws:policy/*" ]
        }
    ]
}
```

```

        "iam:GetPolicyVersion",
        "iam:GetPolicy",
        "iam>ListAttachedGroupPolicies",
        "iam>ListGroupPolicies",
        "iam>ListPolicyVersions",
        "iam>ListPolicies",
        "iam>ListUsers"
    ],
    "Resource": "*"
}
]
}

```

Viewing AWS IoT resources based on tags

You can use conditions in your identity-based policy to control access to AWS IoT resources based on tags. This example shows how you might create a policy that allows viewing a thing. However, permission is granted only if the thing tag `Owner` has the value of that user's user name. This policy also grants the permissions necessary to complete this action on the console.

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ListBillingGroupsInConsole",
            "Effect": "Allow",
            "Action": "iot>ListBillingGroups",
            "Resource": "*"
        },
        {
            "Sid": "ViewBillingGroupsIfOwner",
            "Effect": "Allow",
            "Action": "iot>DescribeBillingGroup",
            "Resource": "arn:aws:iot:*::billinggroup/*",
            "Condition": {
                "StringEquals": {"aws:ResourceTag/Owner": "${aws:username}"}
            }
        }
    ]
}

```

You can attach this policy to the IAM users in your account. If a user named `richard-roe` attempts to view an AWS IoT billing group, the billing group must be tagged `Owner=richard-roe` or `owner=richard-roe`. Otherwise, he is denied access. The condition tag key `Owner` matches both `Owner` and `owner` because condition key names are not case-sensitive. For more information, see [IAM JSON Policy Elements: Condition](#) in the [IAM User Guide](#).

Viewing AWS IoT Device Advisor resources based on tags

You can use conditions in your identity-based policy to control access to AWS IoT Device Advisor resources based on tags. The following example shows how you can create a policy that allows viewing a particular suite definition. However, permission is granted only if the suite definition tag has `SuiteType` set to the value of `MQTT`. This policy also grants the permissions necessary to complete this action on the console.

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewSuiteDefinition",
            "Effect": "Allow",

```

```
        "Action": "iotdeviceadvisor:GetSuiteDefinition",
        "Resource": "arn:aws:iotdeviceadvisor:*:*:suitedefinition/*",
        "Condition": {
            "StringEquals": {"aws:ResourceTag/SuiteType": "MQTT"}
        }
    ]
}
```

Troubleshooting AWS IoT identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with AWS IoT and IAM.

Topics

- [I am not authorized to perform an action in AWS IoT \(p. 388\)](#)
- [I am not authorized to perform iam:PassRole \(p. 388\)](#)
- [I want to view my access keys \(p. 389\)](#)
- [I'm an administrator and want to allow others to access AWS IoT \(p. 389\)](#)
- [I want to allow people outside of my AWS account to access my AWS IoT resources \(p. 389\)](#)

I am not authorized to perform an action in AWS IoT

If the AWS Management Console tells you that you're not authorized to perform an action, then you must contact your administrator for assistance. Your administrator is the person that provided you with your user name and password.

The following example error occurs when the `mateojackson` IAM user tries to use the console to view details about a thing but does not have `iot:DescribeThing` permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
  iot:DescribeThing
      on resource: MyIoTThing
```

In this case, Mateo asks his administrator to update his policies to allow him to access the `MyIoTThing` resource using the `iot:DescribeThing` action.

Using AWS IoT Device Advisor

If you're using AWS IoT Device Advisor, the following example error occurs when the `mateojackson` IAM user tries to use the console to view details about a suite definition but does not have `iotdeviceadvisor:GetSuiteDefinition` permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
  iotdeviceadvisor:GetSuiteDefinition
      on resource: MySuiteDefinition
```

In this case, Mateo asks his administrator to update his policies to allow him to access the `MySuiteDefinition` resource using the `iotdeviceadvisor:GetSuiteDefinition` action.

I am not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the `iam:PassRole` action, then you must contact your administrator for assistance. Your administrator is the person that provided you with your user name and password. Ask that person to update your policies to allow you to pass a role to AWS IoT.

Some AWS services allow you to pass an existing role to that service, instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named `marymajor` tries to use the console to perform an action in AWS IoT. However, the action requires the service to have permissions granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform: iam:PassRole
```

In this case, Mary asks her administrator to update her policies to allow her to perform the `iam:PassRole` action.

I want to view my access keys

After you create your IAM user access keys, you can view your access key ID at any time. However, you can't view your secret access key again. If you lose your secret key, you must create a new access key pair.

Access keys consist of two parts: an access key ID (for example, `AKIAIOSFODNN7EXAMPLE`) and a secret access key (for example, `wJAlrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY`). Like a user name and password, you must use both the access key ID and secret access key together to authenticate your requests. Manage your access keys as securely as you do your user name and password.

Important

Do not provide your access keys to a third party, even to help [find your canonical user ID](#). By doing this, you might give someone permanent access to your account.

When you create an access key pair, you are prompted to save the access key ID and secret access key in a secure location. The secret access key is available only at the time you create it. If you lose your secret access key, you must add new access keys to your IAM user. You can have a maximum of two access keys. If you already have two, you must delete one key pair before creating a new one. To view instructions, see [Managing access keys](#) in the *IAM User Guide*.

I'm an administrator and want to allow others to access AWS IoT

To allow others to access AWS IoT, you must create an IAM entity (user or role) for the person or application that needs access. They will use the credentials for that entity to access AWS. You must then attach a policy to the entity that grants them the correct permissions in AWS IoT.

To get started right away, see [Creating your first IAM delegated user and group](#) in the *IAM User Guide*.

I want to allow people outside of my AWS account to access my AWS IoT resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- To learn whether AWS IoT supports these features, see [How AWS IoT works with IAM \(p. 368\)](#).
- To learn how to provide access to your resources across AWS accounts that you own, see [Providing access to an IAM user in another AWS account that you own](#) in the *IAM User Guide*.
- To learn how to provide access to your resources to third-party AWS accounts, see [Providing access to AWS accounts owned by third parties](#) in the *IAM User Guide*.
- To learn how to provide access through identity federation, see [Providing access to externally authenticated users \(identity federation\)](#) in the *IAM User Guide*.

- To learn the difference between using roles and resource-based policies for cross-account access, see [How IAM roles differ from resource-based policies](#) in the *IAM User Guide*.

Logging and Monitoring

Monitoring is an important part of maintaining the reliability, availability, and performance of AWS IoT and your AWS solutions. You should collect monitoring data from all parts of your AWS solution so that you can more easily debug a multi-point failure, if one occurs. For information on logging and monitoring procedures, see [Monitoring AWS IoT \(p. 400\)](#)

Monitoring Tools

AWS provides tools that you can use to monitor AWS IoT. You can configure some of these tools to do the monitoring for you. Some of the tools require manual intervention. We recommend that you automate monitoring tasks as much as possible.

Automated Monitoring Tools

You can use the following automated monitoring tools to watch AWS IoT and report when something is wrong:

- **Amazon CloudWatch Alarms** – Watch a single metric over a time period that you specify, and perform one or more actions based on the value of the metric relative to a given threshold over a number of time periods. The action is a notification sent to an Amazon Simple Notification Service (Amazon SNS) topic or Amazon EC2 Auto Scaling policy. CloudWatch alarms do not invoke actions simply because they are in a particular state. The state must have changed and been maintained for a specified number of periods. For more information, see [Monitor AWS IoT alarms and metrics using Amazon CloudWatch \(p. 406\)](#).
- **Amazon CloudWatch Logs** – Monitor, store, and access your log files from AWS CloudTrail or other sources. Amazon CloudWatch Logs also allows you to see critical steps AWS IoT Device Advisor test cases take, generated events and MQTT messages sent from your devices or AWS IoT Core during test execution. These logs make it possible to debug and take corrective actions on your devices. For more information, see [Monitor AWS IoT using CloudWatch Logs \(p. 421\)](#). For more information about using Amazon CloudWatch, see [Monitoring Log Files](#) in the *Amazon CloudWatch User Guide*.
- **Amazon CloudWatch Events** – Match events and route them to one or more target functions or streams to make changes, capture state information, and take corrective action. For more information, see [What Is Amazon CloudWatch Events](#) in the *Amazon CloudWatch User Guide*.
- **AWS CloudTrail Log Monitoring** – Share log files between accounts, monitor CloudTrail log files in real time by sending them to CloudWatch Logs, write log processing applications in Java, and validate that your log files have not changed after delivery by CloudTrail. For more information, see [Log AWS IoT API calls using AWS CloudTrail \(p. 440\)](#) and also [Working with CloudTrail Log Files](#) in the *AWS CloudTrail User Guide*.

Manual Monitoring Tools

Another important part of monitoring AWS IoT involves manually monitoring those items that the CloudWatch alarms don't cover. The AWS IoT, CloudWatch, and other AWS service console dashboards provide an at-a-glance view of the state of your AWS environment. We recommend that you also check the log files on AWS IoT.

- AWS IoT dashboard shows:
 - CA certificates
 - Certificates

- Policies
- Rules
- Things
- CloudWatch home page shows:
 - Current alarms and status.
 - Graphs of alarms and resources.
 - Service health status.

You can use CloudWatch to do the following:

- Create [customized dashboards](#) to monitor the services you care about.
- Graph metric data to troubleshoot issues and discover trends.
- Search and browse all your AWS resource metrics.
- Create and edit alarms to be notified of problems.

Compliance validation for AWS IoT Core

Third-party auditors assess the security and compliance of AWS services as part of multiple AWS compliance programs, such as SOC, PCI, FedRAMP, and HIPAA.

To learn whether AWS IoT or other AWS services are in scope of specific compliance programs, see [AWS Services in Scope by Compliance Program](#). For general information, see [AWS Compliance Programs](#).

You can download third-party audit reports using AWS Artifact. For more information, see [Downloading Reports in AWS Artifact](#).

Your compliance responsibility when using AWS services is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. AWS provides the following resources to help with compliance:

- [Security and Compliance Quick Start Guides](#) – These deployment guides discuss architectural considerations and provide steps for deploying baseline environments on AWS that are security and compliance focused.
- [Architecting for HIPAA Security and Compliance Whitepaper](#) – This whitepaper describes how companies can use AWS to create HIPAA-compliant applications.

Note

Not all services are compliant with HIPAA.

- [AWS Compliance Resources](#) – This collection of workbooks and guides might apply to your industry and location.
- [Evaluating Resources with Rules](#) in the *AWS Config Developer Guide* – The AWS Config service assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- [AWS Security Hub](#) – This AWS service provides a comprehensive view of your security state within AWS that helps you check your compliance with security industry standards and best practices.
- [AWS Audit Manager](#) – This AWS service helps you continuously audit your AWS usage to simplify how you manage risk and compliance with regulations and industry standards.

Resilience in AWS IoT Core

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency,

high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between Availability Zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see [AWS Global Infrastructure](#).

AWS IoT Core stores information about your devices in the device registry. It also stores CA certificates, device certificates, and device shadow data. In the event of hardware or network failures, this data is automatically replicated across Availability Zones but not across Regions.

AWS IoT Core publishes MQTT events when the device registry is updated. You can use these messages to back up your registry data and save it somewhere, like a DynamoDB table. You are responsible for saving certificates that AWS IoT Core creates for you or those you create yourself. Device shadow stores state data about your devices and can be resent when a device comes back online. AWS IoT Device Advisor stores information about your test suite configuration. This data is automatically replicated in the event of hardware or network failures.

AWS IoT Core resources are Region-specific and aren't replicated across AWS Regions unless you specifically do so.

For information about Security best practices, see [Security best practices in AWS IoT Core \(p. 395\)](#).

Using AWS IoT Core with interface VPC endpoints

With AWS IoT Core, you can create [IoT data endpoints](#) within your VPC by using [interface VPC endpoints](#). Interface VPC endpoints are powered by AWS PrivateLink, an AWS technology that you can use to access services running on AWS by using private IP addresses. For more information, see [Amazon Virtual Private Cloud](#).

In order to connect devices in the field on remote networks, such as a corporate network to your AWS VPC, refer to the various options listed in the [Network-to-Amazon VPC connectivity matrix](#).

Note

VPC endpoints for IoT Core are currently not supported in AWS China Regions.

Chapter Topics:

- [Creating VPC endpoints for AWS IoT Core \(p. 392\)](#)
- [Controlling Access to AWS IoT Core over VPC endpoints \(p. 393\)](#)
- [Limitations of VPC endpoints \(p. 394\)](#)
- [Scaling VPC endpoints with IoT Core \(p. 394\)](#)
- [Using custom domains with VPC endpoints \(p. 394\)](#)
- [Availability of VPC endpoints for AWS IoT Core \(p. 394\)](#)

Creating VPC endpoints for AWS IoT Core

To get started with VPC endpoints, simply [create an interface VPC endpoint](#), and select AWS IoT Core as the AWS service. If you are using the CLI, first call [describe-vpc-endpoint-services](#) to ensure that you are choosing an Availability Zone where AWS IoT Core is present in your particular AWS Region. For example, in us-east-1, this command would look like:

```
aws ec2 describe-vpc-endpoint-services --service-name com.amazonaws.us-east-1.iot.data
```

Note

The VPC feature for automatically creating a DNS record is disabled because the IoT control and data endpoints are split. To join these endpoints, you must manually create a Private DNS record. For more information about Private VPC DNS records, see [Private DNS for interface endpoints](#). For more information about AWS IoT Core VPC limitations, see [Limitations of VPC endpoints \(p. 394\)](#).

To correctly route DNS queries from your devices to the VPC endpoint interfaces, you must manually create DNS records in a Private Hosted Zone that is attached to your VPC. To get started, see [Creating A Private Hosted Zone](#). Within your Private Hosted Zone, create an alias record for each elastic network interface IP for the VPC endpoint. If you have multiple network interface IPs for multiple VPC endpoints, create weighted DNS records with equal weights across all the weighted records. These IP addresses are available from the [DescribeNetworkInterfaces](#) API call when filtered by the VPC endpoint ID in the description field.

Controlling Access to AWS IoT Core over VPC endpoints

You can restrict device access to AWS IoT Core to be allowed only through VPC endpoint by using [VPC condition context keys](#). AWS IoT Core supports the following VPC related context keys:

- [SourceVpc](#)
- [SourceVpce](#)
- [VPCSourceIp](#)

Note

AWS IoT Core does not support <https://docs.aws.amazon.com/vpc/latest/privatelink/vpc-endpoints-access.html#vpc-endpoint-policies> VPC endpoint policies at this time.

For example, the following policy grants permission to connect to AWS IoT Core using a client ID that matches the thing name and to publish to any topic prefixed by the thing name, conditional on the device connecting to a VPC endpoint with a particular VPC Endpoint ID. This policy would deny connection attempts to your public IoT data endpoint.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:Connect"  
            ],  
            "Resource": [  
                "arn:aws:iot:us-east-1:123456789012:client/  
${iot:Connection.Thing.ThingName}"  
            ],  
            "Condition": {  
                "StringEquals": {  
                    "aws:SourceVpce": "vpce-1a2b3c4d"  
                }  
            }  
        },  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:Publish"  
            ],  
            "Condition": {  
                "StringEquals": {  
                    "aws:SourceVpce": "vpce-1a2b3c4d"  
                }  
            }  
        }  
    ]  
}
```

```
        "Resource": [
            "arn:aws:iot:us-east-1:123456789012:topic/
${iot:Connection.Thing.ThingName}/*"
        ]
    }
}
```

Limitations of VPC endpoints

This section covers the limitations of VPC endpoints compared to public endpoints.

- VPC endpoints are currently supported for [IoT data endpoints](#) only
- MQTT keep alive periods are limited to 230 seconds. Keep alives longer than that period will be automatically reduced to 230 seconds
- Each VPC endpoint supports 100,000 total concurrent connected devices. If you require more connections see [Scaling VPC endpoints with IoT Core \(p. 394\)](#).
- VPC endpoints support IPv4 traffic only.
- VPC endpoints will serve [ATS certificates](#) only, except for custom domains.
- [VPC endpoint policies](#) are not supported at this time.

Scaling VPC endpoints with IoT Core

AWS IoT Core Interface VPC endpoints are limited to 100,000 connected devices over a single interface endpoint. If your use case calls for more concurrent connections to the broker, then we recommend using multiple VPC endpoints and manually routing your devices across your interface endpoints. When creating private DNS records to route traffic to your VPC endpoints, make sure to create as many weighted records as you have VPC endpoints to distribute traffic across your multiple endpoints.

Using custom domains with VPC endpoints

If you want to use custom domains with VPC endpoints, you must create your custom domain name records in a Private Hosted Zone and create routing records in Route53. For more information, see [Creating A Private Hosted Zone](#).

Availability of VPC endpoints for AWS IoT Core

AWS IoT Core Interface VPC endpoints are available in all [AWS IoT Core supported regions](#), with the exception of AWS China Regions.

Infrastructure security in AWS IoT

As a collection of managed services, AWS IoT is protected by the AWS global network security procedures that are described in the [Amazon Web Services: Overview of Security Processes](#) whitepaper.

You use AWS published API calls to access AWS IoT through the network. Clients must support Transport Layer Security (TLS) 1.2 or later. Clients must also support cipher suites with perfect forward secrecy (PFS) such as Ephemeral Diffie-Hellman (DHE) or Elliptic Curve Ephemeral Diffie-Hellman (ECDHE). Most modern systems, such as Java 7 and later, support these modes. For more information, see [Transport security in AWS IoT \(p. 362\)](#).

Requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the [AWS Security Token Service](#) (AWS STS) to generate temporary security credentials to sign requests.

Security monitoring of production fleets or devices with AWS IoT Core

IoT fleets can consist of large numbers of devices that have diverse capabilities, are long-lived, and are geographically distributed. These characteristics make fleet setup complex and prone to errors. And because devices are often constrained in computational power, memory, and storage capabilities, this limits the use of encryption and other forms of security on the devices themselves. Also, devices often use software with known vulnerabilities. These factors make IoT fleets an attractive target for hackers and make it difficult to secure your device fleet on an ongoing basis.

AWS IoT Device Defender addresses these challenges by providing tools to identify security issues and deviations from best practices. You can use AWS IoT Device Defender to analyze, audit, and monitor connected devices to detect abnormal behavior, and mitigate security risks. AWS IoT Device Defender can audit device fleets to ensure they adhere to security best practices and detect abnormal behavior on devices. This makes it possible to enforce consistent security policies across your AWS IoT device fleet and respond quickly when devices are compromised. For more information, see [AWS IoT Device Defender \(p. 772\)](#).

AWS IoT Device Advisor pushes updates and patches your fleet as needed. AWS IoT Device Advisor updates test cases automatically. The test cases that you select are always with latest version. For more information, see [Device Advisor \(p. 948\)](#).

Security best practices in AWS IoT Core

This section contains information about security best practices for AWS IoT Core. For more information, see [Ten security golden rules for IoT solutions](#).

Protecting MQTT connections in AWS IoT

[AWS IoT Core](#) is a managed cloud service that makes it possible for connected devices to interact with cloud applications and other devices easily and securely. AWS IoT Core supports HTTP, [WebSocket](#), and [MQTT](#), a lightweight communication protocol specifically designed to tolerate intermittent connections. If you are connecting to AWS IoT using MQTT, each of your connections must be associated with an identifier known as a client ID. MQTT client IDs uniquely identify MQTT connections. If a new connection is established using a client ID that is already claimed for another connection, the AWS IoT message broker drops the old connection to allow the new connection. Client IDs must be unique within each AWS account and each AWS Region. This means that you don't need to enforce global uniqueness of client IDs outside of your AWS account or across Regions within your AWS account.

The impact and severity of dropping MQTT connections on your device fleet depends on many factors. These include:

- Your use case (for example, the data your devices send to AWS IoT, how much data, and the frequency that the data is sent).
- Your MQTT client configuration (for example, auto reconnect settings, associated back-off timings, and use of [MQTT persistent sessions \(p. 82\)](#)).
- Device resource constraints.

- The root cause of the disconnections, its aggressiveness, and persistence.

To avoid client ID conflicts and their potential negative impacts, make sure that each device or mobile application has an AWS IoT or IAM policy that restricts which client IDs can be used for MQTT connections to the AWS IoT message broker. For example, you can use an IAM policy to prevent a device from unintentionally closing another device's connection by using a client ID that is already in use. For more information, see [Authorization \(p. 312\)](#).

All devices in your fleet must have credentials with privileges that authorize intended actions only, which include (but not limited to) AWS IoT MQTT actions such as publishing messages or subscribing to topics with specific scope and context. The specific permission policies can vary for your use cases. Identify the permission policies that best meet your business and security requirements.

To simplify creation and management of permission policies, you can use [AWS IoT Core policy variables \(p. 318\)](#) and [IAM policy variables](#). Policy variables can be placed in a policy and when the policy is evaluated, the variables are replaced by values that come from the device's request. Using policy variables, you can create a single policy for granting permissions to multiple devices. You can identify the relevant policy variables for your use case based on your AWS IoT account configuration, authentication mechanism, and network protocol used in connecting to AWS IoT message broker. However, to write the best permission policies, consider the specifics of your use case and your [threat model](#).

For example, if you registered your devices in the AWS IoT registry, you can use [thing policy variables \(p. 319\)](#) in AWS IoT policies to grant or deny permissions based on thing properties like thing names, thing types, and thing attribute values. The thing name is obtained from the client ID in the MQTT connect message sent when a thing connects to AWS IoT. The thing policy variables are replaced when a thing connects to AWS IoT over MQTT using TLS mutual authentication or MQTT over the WebSocket protocol using authenticated [Amazon Cognito identities](#). You can use the [AttachThingPrincipal API](#) to attach certificates and authenticated Amazon Cognito identities to a thing. `iot:Connection.Thing.ThingName` is a useful thing policy variable to enforce client ID restrictions. The following example AWS IoT policy requires a registered thing's name to be used as the client ID for MQTT connections to the AWS IoT message broker:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iot:Connect",
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/${iot:Connection.Thing.ThingName}"
            ]
        }
    ]
}
```

If you want to identify ongoing client ID conflicts, you can enable and use [CloudWatch Logs for AWS IoT \(p. 421\)](#). For every MQTT connection that the AWS IoT message broker disconnects due to client ID conflicts, a log record similar to the following is generated:

```
{
    "timestamp": "2019-04-28 22:05:30.105",
    "logLevel": "ERROR",
    "traceId": "02a04a93-0b3a-b608-a27c-1ae8ebdb032a",
    "accountId": "123456789012",
    "status": "Failure",
    "eventType": "Disconnect",
    "protocol": "MQTT",
    "clientId": "clientId01",
    "principalId": "1670fcf6de55adc1930169142405c4a2493d9eb5487127cd0091ca0193a3d3f6",
```

```
    "sourceIp": "203.0.113.1",
    "sourcePort": 21335,
    "reason": "DUPLICATE_CLIENT_ID",
    "details": "A new connection was established with the same client ID"
}
```

You can use a [CloudWatch Logs filter](#) such as `{$.reason= "DUPLICATE_CLIENT_ID" }` to search for instances of client ID conflicts or to set up [CloudWatch metric filters](#) and corresponding CloudWatch alarms for continuous monitoring and reporting.

You can use [AWS IoT Device Defender](#) to identify overly permissive AWS IoT and IAM policies. AWS IoT Device Defender also provides an audit check that notifies you if multiple devices in your fleet are connecting to the AWS IoT message broker using the same client ID.

You can use AWS IoT Device Advisor to validate that your devices can reliably connect to AWS IoT Core and follow security best practices.

See also

- [AWS IoT Core](#)
- [AWS IoT's Security Features \(p. 279\)](#)
- [AWS IoT Core policy variables \(p. 318\)](#)
- [IAM Policy Variables](#)
- [Amazon Cognito Identity](#)
- [AWS IoT Device Defender](#)
- [CloudWatch Logs for AWS IoT \(p. 421\)](#)

Keep your device's clock in sync

It's important to have an accurate time on your device. X.509 certificates have an expiry date and time. The clock on your device is used to verify that a server certificate is still valid. If you're building commercial IoT devices, remember that your products may be stored for extended periods before being sold. Real-time clocks can drift during this time and batteries can get discharged, so setting time in the factory is not sufficient.

For most systems, this means that the device's software must include a network time protocol (NTP) client. The device should wait until it synchronizes with an NTP server before it tries to connect to AWS IoT Core. If this isn't possible, the system should provide a way for a user to set the device's time so that subsequent connections succeed.

After the device synchronizes with an NTP server, it can open a connection with AWS IoT Core. How much clock skew that is allowed depends on what you're trying to do with the connection.

Validate the server certificate

The first thing a device does to interact with AWS IoT is to open a secure connection. When you connect your device to AWS IoT, ensure that you're talking to AWS IoT and not another server impersonating AWS IoT. Each of the AWS IoT servers is provisioned with a certificate issued for the `iot.amazonaws.com` domain. This certificate was issued to AWS IoT by a trusted certificate authority that verified our identity and ownership of the domain.

One of the first things AWS IoT Core does when a device connects is send the device a server certificate. Devices can verify that they were expecting to connect to `iot.amazonaws.com` and that the server on the end of that connection possesses a certificate from a trusted authority for that domain.

TLS certificates are in X.509 format and include a variety of information such as the organization's name, location, domain name, and a validity period. The validity period is specified as a pair of time values called `notBefore` and `notAfter`. Services like AWS IoT Core use limited validity periods (for example, one year) for their server certificates and begin serving new ones before the old ones expire.

Use a single identity per device

Use a single identity per client. Devices generally use X.509 client certificates. Web and mobile applications use Amazon Cognito Identity. This enables you to apply fine-grained permissions to your devices.

For example, you have an application that consists of a mobile phone device that receives status updates from two different smart home objects – a light bulb and a thermostat. The light bulb sends the status of its battery level, and a thermostat sends messages that report the temperature.

AWS IoT authenticates devices individually and treats each connection individually. You can apply fine-grained access controls using authorization policies. You can define a policy for the thermostat that allows it to publish to a topic space. You can define a separate policy for the light bulb that allows it to publish to a different topic space. Finally, you can define a policy for the mobile app that only allows it to connect and subscribe to the topics for the thermostat and the light bulb to receive messages from these devices.

Apply the principle of least privilege and scope down the permissions per device as much as possible. All devices or users should have an AWS IoT policy in AWS IoT that only allows it to connect with a known client ID, and to publish and subscribe to an identified and fixed set of topics.

Use a second AWS Region as backup

Consider storing a copy of your data in a second AWS Region as a backup. For more information, see [Disaster Recovery for AWS IoT](#).

Use just in time provisioning

Manually creating and provisioning each device can be time consuming. AWS IoT provides a way to define a template to provision devices when they first connect to AWS IoT. For more information, see [Just-in-time provisioning \(p. 712\)](#).

Permissions to run AWS IoT Device Advisor tests

The following policy template shows the minimum permissions and IAM entity required to run AWS IoT Device Advisor test cases. You will need to replace `your-device-role-arn` with the device role Amazon Resource Name (ARN) that you created under the [prerequisites](#).

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "VisualEditor0",  
            "Effect": "Allow",  
            "Action": "iam:PassRole",  
            "Resource": "your-device-role-arn",  
            "Condition": {  
                "StringEquals": {  
                    "iam:PassedToService": "iotdeviceadvisor.amazonaws.com"  
                }  
            },  
        },  
    ],  
}
```

```
{  
    "Sid": "VisualEditor1",  
    "Effect": "Allow",  
    "Action": [  
        "iam>ListRoles", // Required to list device roles in the device advisor  
        "console  
            "iot:Connect",  
            "logs:DescribeLogStreams",  
            "iot:DescribeThing",  
            "iot:DescribeCertificate",  
            "logs>CreateLogGroup",  
            "logs:DescribeLogGroups",  
            "logs:PutLogEvents",  
            "iot:DescribeEndpoint",  
            "execute-api:Invoke*",  
            "logs>CreateLogStream",  
            "iot>ListPrincipalPolicies",  
            "iot>ListThingPrincipals",  
            "iot>ListThings",  
            "iot:Publish",  
            "iot>CreateJob",  
            "iot:DescribeJob",  
            "iot>ListCertificates",  
            "iot>ListAttachedPolicies",  
            "iot:UpdateThingShadow",  
            "iot:GetPolicy"  
        ],  
        "Resource": "*"  
    ],  
    {  
        "Sid": "VisualEditor2",  
        "Effect": "Allow",  
        "Action": "iotdeviceadvisor:*",  
        "Resource": "*"  
    }  
}
```

AWS training and certification

Take the following course to learn about key concepts for AWS IoT security: [AWS IoT Security Primer](#).

Monitoring AWS IoT

Monitoring is an important part of maintaining the reliability, availability, and performance of AWS IoT and your AWS solutions.

We strongly encourage you to collect monitoring data from all parts of your AWS solution to make it easier to debug a multi-point failure, if one occurs. Start by creating a monitoring plan that answers the following questions. If you're not sure how to answer these, you can still continue to [enable logging \(p. 400\)](#) and establish your performance baselines.

- What are your monitoring goals?
- Which resources will you monitor?
- How often will you monitor these resources?
- Which monitoring tools will you use?
- Who will perform the monitoring tasks?
- Who should be notified when something goes wrong?

Your next step is to [enable logging \(p. 400\)](#) and establish a baseline of normal AWS IoT performance in your environment by measuring performance at various times and under different load conditions. As you monitor AWS IoT, keep historical monitoring data so that you can compare it with current performance data. This will help you identify normal performance patterns and performance anomalies, and devise methods to address issues.

To establish your baseline performance for AWS IoT, you should monitor these metrics to start. You can always monitor more metrics later.

- [PublishIn.Success \(p. 413\)](#)
- [PublishOut.Success \(p. 413\)](#)
- [Subscribe.Success \(p. 413\)](#)
- [Ping.Success \(p. 413\)](#)
- [Connect.Success \(p. 413\)](#)
- [GetThingShadow.Accepted \(p. 416\)](#)
- [UpdateThingShadow.Accepted \(p. 416\)](#)
- [DeleteThingShadow.Accepted \(p. 416\)](#)
- [RulesExecuted \(p. 411\)](#)

The topics in this section can help you start logging and monitoring AWS IoT.

Topics

- [Configure AWS IoT logging \(p. 400\)](#)
- [Monitor AWS IoT alarms and metrics using Amazon CloudWatch \(p. 406\)](#)
- [Monitor AWS IoT using CloudWatch Logs \(p. 421\)](#)
- [Log AWS IoT API calls using AWS CloudTrail \(p. 440\)](#)

Configure AWS IoT logging

You must enable logging by using the AWS IoT console, CLI, or API before you can monitor and log AWS IoT activity.

You can enable logging for all of AWS IoT or only specific thing groups. You can configure AWS IoT logging by using the AWS IoT console, CLI, or API; however, you must use the CLI or API to configure logging for specific thing groups.

When considering how to configure your AWS IoT logging, the default logging configuration determines how AWS IoT activity will be logged unless specified otherwise. Starting out, you might want to obtain detailed logs with a default [log level \(p. 406\)](#) of `INFO` or `DEBUG`. After reviewing the initial logs, you can change the default log level to a less verbose level such as `WARN` or `ERROR` and set a more verbose resource-specific log level on resources that might need more attention. Log levels can be changed whenever you want.

Configure logging role and policy

Before you can enable logging in AWS IoT, you must create an IAM role and a policy that gives AWS permission to monitor AWS IoT activity on your behalf.

Note

Before you enable AWS IoT logging, make sure you understand the CloudWatch Logs access permissions. Users with access to CloudWatch Logs can see debugging information from your devices. For more information, see [Authentication and Access Control for Amazon CloudWatch Logs](#).

If you expect high traffic patterns in AWS IoT Core due to load testing, consider turning off IoT logging to prevent throttling. If high traffic is detected, our service may disable logging in your account.

Following shows how to create a logging role and policy for AWS IoT Core resources. For information about how you can create an IAM logging role and policy for AWS IoT Core for LoRaWAN, see [Create logging role and policy for AWS IoT Core for LoRaWAN \(p. 1079\)](#).

Create a logging role

To create a logging role, open the [Roles hub of the IAM console](#) and choose **Create role**.

1. Under **Select type of trusted entity**, choose **AWS Service, IoT**.
2. Under **Select your use case**, choose **IoT**, and then choose **Next: Permissions**.
3. On the page that displays the policies that are automatically attached to the service role, choose **Next: Tags**, and then choose **Next: Review**.
4. Enter a **Role name** and **Role description** for the role, and then choose **Create role**.
5. In the list of **Roles**, find the role you created, open it, and copy the **Role ARN (`logging-role-arn`)** to use when you [Configure default logging in the AWS IoT \(console\) \(p. 402\)](#).

Logging role policy

The following policy documents provide the role policy and trust policy that allow AWS IoT to submit log entries to CloudWatch on your behalf. If you also allowed AWS IoT Core for LoRaWAN to submit log entries, you'll see a policy document created for you that logs both activities. For information about how to create an IAM logging role and policy for AWS IoT Core for LoRaWAN, see [Create logging role and policy for AWS IoT Core for LoRaWAN \(p. 1079\)](#).

Note

These documents were created for you when you created the logging role.

Role policy:

```
{
```

```
"Version": "2012-10-17",
"Statement": [
    {
        "Effect": "Allow",
        "Action": [
            "logs:CreateLogGroup",
            "logs:CreateLogStream",
            "logs:PutLogEvents",
            "logs:PutMetricFilter",
            "logs:PutRetentionPolicy"
        ],
        "Resource": [
            "*"
        ]
    }
]
```

Trust policy to log only AWS IoT Core activity:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "",
            "Effect": "Allow",
            "Principal": {
                "Service": "iot.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

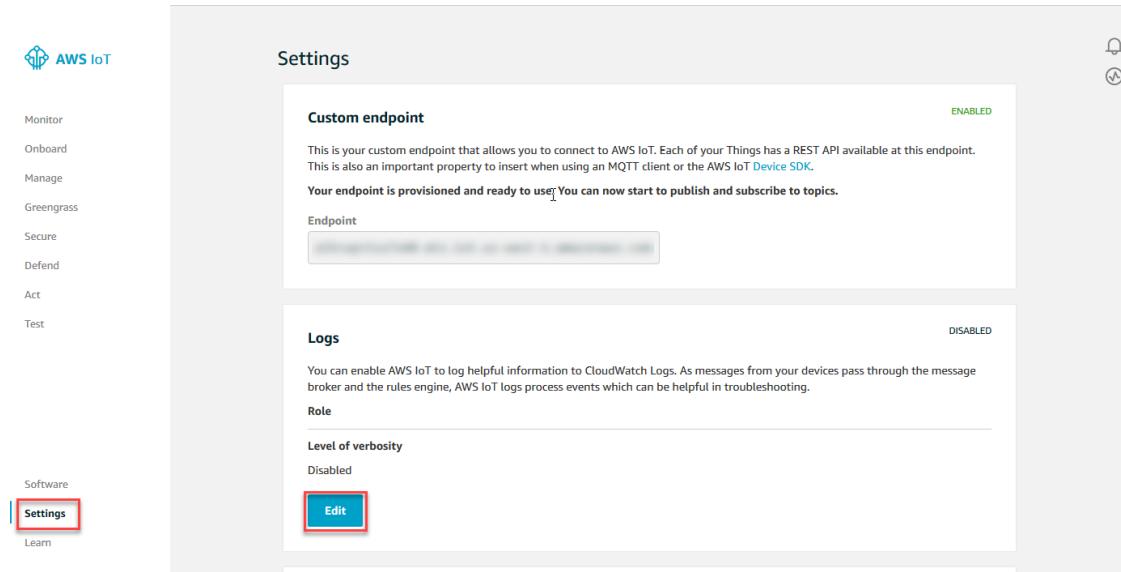
Configure default logging in the AWS IoT (console)

This section describes how to use the AWS IoT console to configure logging for all of AWS IoT. To configure logging for only specific thing groups, you must use the CLI or API. For information about configuring logging for specific thing groups, see [Configure resource-specific logging in AWS IoT \(CLI\) \(p. 404\)](#).

To use the AWS IoT console to configure default logging for all of AWS IoT

1. Sign in to the AWS IoT console. For more information, see [Open the AWS IoT console \(p. 19\)](#).
2. In the left navigation pane, choose **Settings**. In the **Logs** section of the **Settings** page, choose **Edit**.

The **Logs** section displays the logging role and level of verbosity used by all of AWS IoT.



3. On the **Configure role setting** page, choose the **Level of verbosity** that describes the [level of detail \(p. 406\)](#) of the log entries that you want to appear in the CloudWatch logs.

The screenshot shows the 'Configure role setting' page. At the top, it says 'Configure role setting'. Below that, there's a section for 'Level of verbosity' with a dropdown menu containing 'Disable logging' (which is highlighted with a red box). Underneath, there's a 'Set role' section with a note about selecting a role for CloudWatch Logs. It shows a dropdown menu with 'No role selected' and two buttons: 'Create Role' and 'Select' (both highlighted with red boxes). At the bottom, there are 'Cancel' and 'Update' buttons, with 'Update' highlighted with a red box.

4. Choose **Select** to specify a role that you created in [Create a logging role \(p. 401\)](#), or **Create Role** to create a new role to use for logging.
5. Choose **Update** to save your changes.

After you've enabled logging, visit [Viewing AWS IoT logs in the CloudWatch console \(p. 421\)](#) to learn more about viewing the log entries.

Configure default logging in AWS IoT (CLI)

This section describes how to configure global logging for AWS IoT by using the CLI.

Note

You need the Amazon Resource Name (ARN) of the role that you want to use. If you need to create a role to use for logging, see [Create a logging role \(p. 401\)](#) before continuing.

The principal used to call the API must have [Pass role permissions \(p. 445\)](#) for your logging role.

You can also perform this procedure with the API by using the methods in the AWS API that correspond to the CLI commands shown here.

To use the CLI to configure default logging for AWS IoT

1. Use the [set-v2-logging-options](#) command to set the logging options for your account.

```
aws iot set-v2-logging-options \
--role-arn logging-role-arn \
--default-log-level log-level
```

where:

--role-arn

The role ARN that grants AWS IoT permission to write to your logs in CloudWatch Logs.

--default-log-level

The [log level \(p. 406\)](#) to use. Valid values are: ERROR, WARN, INFO, DEBUG, or DISABLED

--no-disable-all-logs

An optional parameter that enables all AWS IoT logging. Use this parameter to enable logging when it is currently disabled.

--disable-all-logs

An optional parameter that disables all AWS IoT logging. Use this parameter to disable logging when it is currently enabled.

2. Use the [get-v2-logging-options](#) command to get your current logging options.

```
aws iot get-v2-logging-options
```

After you've enabled logging, visit [Viewing AWS IoT logs in the CloudWatch console \(p. 421\)](#) to learn more about viewing the log entries.

Note

AWS IoT continues to support older commands ([set-logging-options](#) and [get-logging-options](#)) to set and get global logging on your account. Be aware that when these commands are used, the resulting logs contain plain-text, rather than JSON payloads and logging latency is generally higher. No further improvements will be made to the implementation of these older commands. We recommend that you use the "v2" versions to configure your logging options and, when possible, change legacy applications that use the older versions.

Configure resource-specific logging in AWS IoT (CLI)

This section describes how to configure resource-specific logging for AWS IoT by using the CLI. Resource-specific logging allows you to specify a logging level for a specific [thing group \(p. 258\)](#).

Thing groups can contain other thing groups to create a hierarchical relationship. This procedure describes how to configure the logging of a single thing group. You can apply this procedure to the parent thing group in a hierarchy to configure the logging for all thing groups in the hierarchy. You can also apply this procedure to a child thing group to override the logging configuration of its parent.

Note

You need the Amazon Resource Name (ARN) of the role you want to use. If you need to create a role to use for logging, see [Create a logging role \(p. 401\)](#) before continuing.

The principal used to call the API must have [Pass role permissions \(p. 445\)](#) for your logging role.

You can also perform this procedure with the API by using the methods in the AWS API that correspond to the CLI commands shown here.

To use the CLI to configure resource-specific logging for AWS IoT

1. Use the [set-v2-logging-options](#) command to set the logging options for your account.

```
aws iot set-v2-logging-options \  
  --role-arn logging-role-arn \  
  --default-log-level log-level
```

where:

--role-arn

The role ARN that grants AWS IoT permission to write to your logs in CloudWatch Logs.

--default-log-level

The [log level \(p. 406\)](#) to use. Valid values are: **ERROR**, **WARN**, **INFO**, **DEBUG**, or **DISABLED**

--no-disable-all-logs

An optional parameter that enables all AWS IoT logging. Use this parameter to enable logging when it is currently disabled.

--disable-all-logs

An optional parameter that disables all AWS IoT logging. Use this parameter to disable logging when it is currently enabled.

2. Use the [set-v2-logging-level](#) command to configure resource-specific logging for a thing group.

```
aws iot set-v2-logging-level \  
  --log-target targetType=THING_GROUP,targetName=thing_group_name \  
  --log-level log_level
```

--log-target

The type and name of the resource for which you are configuring logging. The `target_type` value must be `THING_GROUP`. The `log-target` parameter value can be text, as shown in the preceding command example, or a JSON string, such as the following example.

```
aws iot set-v2-logging-level \  
  --log-target '{"targetType": "THING_GROUP", "targetName":  
  "thing_group_name"}' \  
  --log-level log_level
```

--log-level

The logging level used when generating logs for the specified resource. Valid values are: **DEBUG**, **INFO**, **ERROR**, **WARN**, and **DISABLED**

3. Use the [list-v2-logging-levels](#) command to list the currently configured logging levels.

```
aws iot list-v2-logging-levels
```

4. Use the [delete-v2-logging-level](#) command to delete a resource-specific logging level.

```
aws iot delete-v2-logging-level \
    --targetType "THING_GROUP" \
    --targetName "thing_group_name"
```

--targetType

The `target_type` value must be `THING_GROUP`

--targetName

The name of the thing group for which to remove the logging level.

After you've enabled logging, visit [Viewing AWS IoT logs in the CloudWatch console \(p. 421\)](#) to learn more about viewing the log entries.

Log levels

These log levels determine the events that are logged and apply to default and resource-specific log levels.

ERROR

Any error that causes an operation to fail.

Logs include ERROR information only.

WARN

Anything that can potentially cause inconsistencies in the system, but might not cause the operation to fail.

Logs include ERROR and WARN information.

INFO

High-level information about the flow of things.

Logs include INFO, ERROR, and WARN information.

DEBUG

Information that might be helpful when debugging a problem.

Logs include DEBUG, INFO, ERROR, and WARN information.

DISABLED

All logging is disabled.

Monitor AWS IoT alarms and metrics using Amazon CloudWatch

You can monitor AWS IoT using CloudWatch, which collects and processes raw data from AWS IoT into readable, near real-time metrics. These statistics are recorded for a period of two weeks, so that you

can access historical information and gain a better perspective on how your web application or service is performing. By default, AWS IoT metric data is sent automatically to CloudWatch in one minute intervals. For more information, see [What Are Amazon CloudWatch, Amazon CloudWatch Events, and Amazon CloudWatch Logs?](#) in the [Amazon CloudWatch User Guide](#).

Using AWS IoT metrics

The metrics reported by AWS IoT provide information that you can analyze in different ways. The following use cases are based on a scenario where you have ten things that connect to the internet once a day. Each day:

- Ten things connect to AWS IoT at roughly the same time.
- Each thing subscribes to a topic filter, and then waits for an hour before disconnecting. During this period, things communicate with one another and learn more about the state of the world.
- Each thing publishes some perception it has based on its newly found data using `UpdateThingShadow`.
- Each thing disconnects from AWS IoT.

To help you get started, these topics explore some of the questions that you might have.

- [How can I be notified if my things do not connect successfully each day? \(p. 408\)](#)
- [How can I be notified if my things are not publishing data each day? \(p. 408\)](#)
- [How can I be notified if my thing's shadow updates are being rejected each day? \(p. 409\)](#)
- [How can I create a CloudWatch alarm for Jobs? \(p. 409\)](#)

More about CloudWatch alarms and metrics

- [Creating CloudWatch alarms to monitor AWS IoT \(p. 407\)](#)
- [AWS IoT metrics and dimensions \(p. 410\)](#)

Creating CloudWatch alarms to monitor AWS IoT

You can create a CloudWatch alarm that sends an Amazon SNS message when the alarm changes state. An alarm watches a single metric over a time period you specify. When the value of the metric exceeds a given threshold over a number of time periods, one or more actions are performed. The action can be a notification sent to an Amazon SNS topic or Auto Scaling policy. Alarms trigger actions for sustained state changes only. CloudWatch alarms do not trigger actions simply because they are in a particular state; the state must have changed and been maintained for a specified number of periods.

The following topics describe some examples of using CloudWatch alarms.

- [How can I be notified if my things do not connect successfully each day? \(p. 408\)](#)
- [How can I be notified if my things are not publishing data each day? \(p. 408\)](#)
- [How can I be notified if my thing's shadow updates are being rejected each day? \(p. 409\)](#)
- [How can I create a CloudWatch alarm for jobs? \(p. 409\)](#)

You can see all the metrics that CloudWatch alarms can monitor at [AWS IoT metrics and dimensions \(p. 410\)](#).

How can I be notified if my things do not connect successfully each day?

1. Create an Amazon SNS topic named `things-not-connecting-successfully`, and record its Amazon Resource Name (ARN). This procedure will refer to your topic's ARN as `sns-topic-arn`.

For more information on how to create an Amazon SNS notification, see [Getting Started with Amazon SNS](#).

2. Create the alarm.

```
aws cloudwatch put-metric-alarm \
--alarm-name ConnectSuccessAlarm \
--alarm-description "Alarm when my Things don't connect successfully" \
--namespace AWS/IoT \
--metric-name Connect.Success \
--dimensions Name=Protocol,Value=MQTT \
--statistic Sum \
--threshold 10 \
--comparison-operator LessThanThreshold \
--period 86400 \
--evaluation-periods 1 \
--alarm-actions sns-topic-arn
```

3. Test the alarm.

```
aws cloudwatch set-alarm-state --alarm-name ConnectSuccessAlarm --state-reason
"initializing" --state-value OK
```

```
aws cloudwatch set-alarm-state --alarm-name ConnectSuccessAlarm --state-reason
"initializing" --state-value ALARM
```

4. Verify that the alarm appears in your [CloudWatch console](#).

How can I be notified if my things are not publishing data each day?

1. Create an Amazon SNS topic named `things-not-publishing-data`, and record its Amazon Resource Name (ARN). This procedure will refer to your topic's ARN as `sns-topic-arn`.

For more information on how to create an Amazon SNS notification, see [Getting Started with Amazon SNS](#).

2. Create the alarm.

```
aws cloudwatch put-metric-alarm \
--alarm-name PublishInSuccessAlarm \
--alarm-description "Alarm when my Things don't publish their data" \
--namespace AWS/IoT \
--metric-name PublishIn.Success \
--dimensions Name=Protocol,Value=MQTT \
--statistic Sum \
--threshold 10 \
--comparison-operator LessThanThreshold \
--period 86400 \
--evaluation-periods 1 \
--alarm-actions sns-topic-arn
```

3. Test the alarm.

```
aws cloudwatch set-alarm-state --alarm-name PublishInSuccessAlarm --state-reason "initializing" --state-value OK
```

```
aws cloudwatch set-alarm-state --alarm-name PublishInSuccessAlarm --state-reason "initializing" --state-value ALARM
```

4. Verify that the alarm appears in your [CloudWatch console](#).

How can I be notified if my thing's shadow updates are being rejected each day?

1. Create an Amazon SNS topic named `things-shadow-updates-rejected`, and record its Amazon Resource Name (ARN). This procedure will refer to your topic's ARN as `sns-topic-arn`.

For more information on how to create an Amazon SNS notification, see [Getting Started with Amazon SNS](#).

2. Create the alarm.

```
aws cloudwatch put-metric-alarm \
--alarm-name UpdateThingShadowSuccessAlarm \
--alarm-description "Alarm when my Things Shadow updates are getting rejected" \
--namespace AWS/IoT \
--metric-name UpdateThingShadow.Success \
--dimensions Name=Protocol,Value=MQTT \
--statistic Sum \
--threshold 10 \
--comparison-operator LessThanThreshold \
--period 86400 \
--unit Count \
--evaluation-periods 1 \
--alarm-actions sns-topic-arn
```

3. Test the alarm.

```
aws cloudwatch set-alarm-state --alarm-name UpdateThingShadowSuccessAlarm --state-reason "initializing" --state-value OK
```

```
aws cloudwatch set-alarm-state --alarm-name UpdateThingShadowSuccessAlarm --state-reason "initializing" --state-value ALARM
```

4. Verify that the alarm appears in your [CloudWatch console](#).

How can I create a CloudWatch alarm for jobs?

The Jobs service provides CloudWatch metrics for you to monitor your jobs. You can create CloudWatch alarms to monitor any [Jobs metrics \(p. 416\)](#).

The following command creates a CloudWatch alarm to monitor the total number of failed job executions for Job `SampleOTAJob` and notifies you when more than 20 job executions have failed. The alarm monitors the Jobs metric `FailedJobExecutionTotalCount` by checking the reported value every 300 seconds. It is activated when a single reported value is greater than 20, meaning there were more than 20 failed job executions since the job started. When the alarm goes off, it sends a notification to the provided Amazon SNS topic.

```
aws cloudwatch put-metric-alarm \
--alarm-name TotalFailedJobExecution-SampleOTAJob \
--alarm-description "Alarm when total number of failed job execution exceeds the threshold for SampleOTAJob" \
--namespace AWS/IoT \
--metric-name FailedJobExecutionTotalCount \
--dimensions Name=JobId,Value=SampleOTAJob \
--statistic Sum \
--threshold 20 \
--comparison-operator GreaterThanThreshold \
--period 300 \
--unit Count \
--evaluation-periods 1 \
--alarm-actions arn:aws:sns:<AWS_REGION>:<AWS_ACCOUNT_ID>:SampleOTAJob-has-too-many-failed-job-eexecutions
```

The following command creates a CloudWatch alarm to monitor the number of failed job executions for Job *SampleOTAJob* in a given period. It then notifies you when more than five job executions have failed during that period. The alarm monitors the Jobs metric `FailedJobExecutionCount` by checking the reported value every 3600 seconds. It is activated when a single reported value is greater than 5, meaning there were more than 5 failed job executions in the past hour. When the alarm goes off, it sends a notification to the provided Amazon SNS topic.

```
aws cloudwatch put-metric-alarm \
--alarm-name FailedJobExecution-SampleOTAJob \
--alarm-description "Alarm when number of failed job execution per hour exceeds the threshold for SampleOTAJob" \
--namespace AWS/IoT \
--metric-name FailedJobExecutionCount \
--dimensions Name=JobId,Value=SampleOTAJob \
--statistic Sum \
--threshold 5 \
--comparison-operator GreaterThanThreshold \
--period 3600 \
--unit Count \
--evaluation-periods 1 \
--alarm-actions arn:aws:sns:<AWS_REGION>:<AWS_ACCOUNT_ID>:SampleOTAJob-has-too-many-failed-job-eexecutions-per-hour
```

AWS IoT metrics and dimensions

When you interact with AWS IoT, the service sends the following metrics and dimensions to CloudWatch every minute. You can use the following procedures to view the metrics for AWS IoT.

To view metrics (CloudWatch console)

Metrics are grouped first by the service namespace, and then by the various dimension combinations within each namespace.

1. Open the [CloudWatch console](#).
2. In the navigation pane, choose **Metrics** and then choose **All metrics**.
3. In the **Browse** tab, search for AWS IoT to view the list of metrics.

To view metrics (CLI)

- At a command prompt, use the following command:

```
aws cloudwatch list-metrics --namespace "AWS/IoT"
```

CloudWatch displays the following groups of metrics for AWS IoT:

- [AWS IoT metrics \(p. 411\)](#)
- [AWS IoT Core credential provider metrics \(p. 411\)](#)
- [Rule metrics \(p. 411\)](#)
- [Rule action metrics \(p. 412\)](#)
- [HTTP action specific metrics \(p. 412\)](#)
- [Message broker metrics \(p. 413\)](#)
- [Device shadow metrics \(p. 416\)](#)
- [Jobs metrics \(p. 416\)](#)
- [Device Defender audit metrics \(p. 418\)](#)
- [Device Defender detect metrics \(p. 418\)](#)
- [Device provisioning metrics \(p. 418\)](#)
- [Dimensions for metrics \(p. 420\)](#)

AWS IoT metrics

Metric	Description
AddThingToDynamicThingGroupsFailed	The number of failure events associated with adding a thing to a dynamic thing group. The DynamicThingGroupName dimension contains the name of the dynamic groups that failed to add things.
NumLogBatchesFailedToPublishThrottled	The singular batch of log events that has failed to publish due to throttling errors.
NumLogEventsFailedToPublishThrottled	The number of log events within the batch that have failed to publish due to throttling errors.

AWS IoT Core credential provider metrics

Metric	Description
CredentialExchangeSuccess	The number of successful AssumeRoleWithCertificate requests to AWS IoT Core credentials provider.

Rule metrics

Metric	Description
ParseError	The number of JSON parse errors that occurred in messages published on a topic on which a rule is

Metric	Description
	listening. The <code>RuleName</code> dimension contains the name of the rule.
<code>RuleMessageThrottled</code>	The number of messages throttled by the rules engine because of malicious behavior or because the number of messages exceeds the rules engine's throttle limit. The <code>RuleName</code> dimension contains the name of the rule to be triggered.
<code>RuleNotFound</code>	The rule to be triggered could not be found. The <code>RuleName</code> dimension contains the name of the rule.
<code>RulesExecuted</code>	The number of AWS IoT rules executed.
<code>TopicMatch</code>	The number of incoming messages published on a topic on which a rule is listening. The <code>RuleName</code> dimension contains the name of the rule.

Rule action metrics

Metric	Description
<code>Failure</code>	The number of failed rule action invocations. The <code>RuleName</code> dimension contains the name of the rule that specifies the action. The <code>ActionType</code> dimension contains the type of action that was invoked.
<code>Success</code>	The number of successful rule action invocations. The <code>RuleName</code> dimension contains the name of the rule that specifies the action. The <code>ActionType</code> dimension contains the type of action that was invoked.
<code>ErrorActionFailure</code>	The number of failed error actions. The <code>RuleName</code> dimension contains the name of the rule that specifies the action. The <code>ActionType</code> dimension contains the type of action that was invoked.
<code>ErrorActionSuccess</code>	The number of successful error actions. The <code>RuleName</code> dimension contains the name of the rule that specifies the action. The <code>ActionType</code> dimension contains the type of action that was invoked.

HTTP action specific metrics

Metric	Description
<code>HttpCode_Other</code>	Generated if the status code of the response from the downstream web service/application is not 2xx, 4xx or 5xx.
<code>HttpCode_4XX</code>	Generated if the status code of the response from the downstream web service/application is between 400 and 499.

Metric	Description
HttpCode_5XX	Generated if the status code of the response from the downstream web service/application is between 500 and 599.
HttpInvalidUrl	Generated if an endpoint URL, after substitution templates are replaced, does not start with https://.
HttpRequestTimeout	Generated if the downstream web service/application does not return response within request timeout limit. For more information, see Service Quotas .
HttpUnknownHost	Generated if the URL is valid, but the service does not exist or is unreachable.

Message broker metrics

Note

The message broker metrics are displayed in the CloudWatch console under **Protocol Metrics**.

Metric	Description
Connect.AuthError	The number of connection requests that could not be authorized by the message broker. The <code>Protocol</code> dimension contains the protocol used to send the CONNECT message.
Connect.ClientError	The number of connection requests rejected because the MQTT message did not meet the requirements defined in AWS IoT quotas (p. 1144) . The <code>Protocol</code> dimension contains the protocol used to send the CONNECT message.
Connect.ClientIDThrottle	The number of connection requests throttled because the client exceeded the allowed connect request rate for a specific client ID. The <code>Protocol</code> dimension contains the protocol used to send the CONNECT message.
Connect.ServerError	The number of connection requests that failed because an internal error occurred. The <code>Protocol</code> dimension contains the protocol used to send the CONNECT message.
Connect.Success	The number of successful connections to the message broker. The <code>Protocol</code> dimension contains the protocol used to send the CONNECT message.
Connect.Throttle	The number of connection requests that were throttled because the account exceeded the allowed connect request rate. The <code>Protocol</code> dimension contains the protocol used to send the CONNECT message.
Ping.Success	The number of ping messages received by the message broker. The <code>Protocol</code> dimension contains the protocol used to send the ping message.

Metric	Description
PublishIn.AuthError	The number of publish requests the message broker was unable to authorize. The <code>Protocol</code> dimension contains the protocol used to publish the message.
PublishIn.ClientError	The number of publish requests rejected by the message broker because the message did not meet the requirements defined in AWS IoT quotas (p. 1144) . The <code>Protocol</code> dimension contains the protocol used to publish the message.
PublishIn.ServerError	The number of publish requests the message broker failed to process because an internal error occurred. The <code>Protocol</code> dimension contains the protocol used to send the <code>PUBLISH</code> message.
PublishIn.Success	The number of publish requests successfully processed by the message broker. The <code>Protocol</code> dimension contains the protocol used to send the <code>PUBLISH</code> message.
PublishIn.Throttle	The number of publish request that were throttled because the client exceeded the allowed inbound message rate. The <code>Protocol</code> dimension contains the protocol used to send the <code>PUBLISH</code> message.
PublishOut.AuthError	The number of publish requests made by the message broker that could not be authorized by AWS IoT. The <code>Protocol</code> dimension contains the protocol used to send the <code>PUBLISH</code> message.
PublishOut.ClientError	The number of publish requests made by the message broker that were rejected because the message did not meet the requirements defined in AWS IoT quotas (p. 1144) . The <code>Protocol</code> dimension contains the protocol used to send the <code>PUBLISH</code> message.
PublishOut.Success	The number of publish requests successfully made by the message broker. The <code>Protocol</code> dimension contains the protocol used to send the <code>PUBLISH</code> message.
PublishOut.Throttle	The number of publish requests that were throttled because the client exceeded the allowed outbound message rate. The <code>Protocol</code> dimension contains the protocol used to send the <code>PUBLISH</code> message.
PublishRetained.AuthError	The number of publish requests with the <code>REtain</code> flag set that the message broker was unable to authorize. The <code>Protocol</code> dimension contains the protocol used to send the <code>PUBLISH</code> message.
PublishRetained.ServerError	The number of retained publish requests the message broker failed to process because an internal error occurred. The <code>Protocol</code> dimension contains the protocol used to send the <code>PUBLISH</code> message.

Metric	Description
PublishRetained.Success	The number of publish requests with the RETAIN flag set that were successfully processed by the message broker. The Protocol dimension contains the protocol used to send the PUBLISH message.
PublishRetained.Throttle	The number of publish requests with the RETAIN flag set that were throttled because the client exceeded the allowed inbound message rate. The Protocol dimension contains the protocol used to send the PUBLISH message.
Subscribe.AuthError	The number of subscription requests made by a client that could not be authorized. The Protocol dimension contains the protocol used to send the SUBSCRIBE message.
Subscribe.ClientError	The number of subscribe requests that were rejected because the SUBSCRIBE message did not meet the requirements defined in AWS IoT quotas (p. 1144) . The Protocol dimension contains the protocol used to send the SUBSCRIBE message.
Subscribe.ServerError	The number of subscribe requests that were rejected because an internal error occurred. The Protocol dimension contains the protocol used to send the SUBSCRIBE message.
Subscribe.Success	The number of subscribe requests that were successfully processed by the message broker. The Protocol dimension contains the protocol used to send the SUBSCRIBE message.
Subscribe.Throttle	The number of subscribe requests that were throttled because the client exceeded the allowed subscribe request rate. The Protocol dimension contains the protocol used to send the SUBSCRIBE message.
Unsubscribe.ClientError	The number of unsubscribe requests that were rejected because the UNSUBSCRIBE message did not meet the requirements defined in AWS IoT quotas (p. 1144) . The Protocol dimension contains the protocol used to send the UNSUBSCRIBE message.
Unsubscribe.ServerError	The number of unsubscribe requests that were rejected because an internal error occurred. The Protocol dimension contains the protocol used to send the UNSUBSCRIBE message.
Unsubscribe.Success	The number of unsubscribe requests that were successfully processed by the message broker. The Protocol dimension contains the protocol used to send the UNSUBSCRIBE message.
Unsubscribe.Throttle	The number of unsubscribe requests that were rejected because the client exceeded the allowed unsubscribe request rate. The Protocol dimension contains the protocol used to send the UNSUBSCRIBE message.

Device shadow metrics

Note

The device shadow metrics are displayed in the CloudWatch console under **Protocol Metrics**.

Metric	Description
DeleteThingShadow.Accepted	The number of DeleteThingShadow requests processed successfully. The Protocol dimension contains the protocol used to make the request.
GetThingShadow.Accepted	The number of GetThingShadow requests processed successfully. The Protocol dimension contains the protocol used to make the request.
ListThingShadow.Accepted	The number of ListThingShadow requests processed successfully. The Protocol dimension contains the protocol used to make the request.
UpdateThingShadow.Accepted	The number of UpdateThingShadow requests processed successfully. The Protocol dimension contains the protocol used to make the request.

Jobs metrics

Metric	Description
CanceledJobExecutionCount	The number of job executions whose status has changed to CANCELED within a time period that is determined by CloudWatch. (For more information about CloudWatch metrics, see Amazon CloudWatch Metrics .) The JobId dimension contains the ID of the job.
CanceledJobExecutionTotalCount	The total number of job executions whose status is CANCELED for the given job. The JobId dimension contains the ID of the job.
ClientErrorCount	The number of client errors generated while executing the job. The JobId dimension contains the ID of the job.
FailedJobExecutionCount	The number of job executions whose status has changed to FAILED within a time period that is determined by CloudWatch. (For more information about CloudWatch metrics, see Amazon CloudWatch Metrics .) The JobId dimension contains the ID of the job.
FailedJobExecutionTotalCount	The total number of job executions whose status is FAILED for the given job. The JobId dimension contains the ID of the job.
InProgressJobExecutionCount	The number of job executions whose status has changed to IN_PROGRESS within a time period that is determined by CloudWatch. (For more information

Metric	Description
	about CloudWatch metrics, see Amazon CloudWatch Metrics .) The <code>JobId</code> dimension contains the ID of the job.
<code>InProgressJobExecutionTotalCount</code>	The total number of job executions whose status is <code>IN_PROGRESS</code> for the given job. The <code>JobId</code> dimension contains the ID of the job.
<code>RejectedJobExecutionTotalCount</code>	The total number of job executions whose status is <code>REJECTED</code> for the given job. The <code>JobId</code> dimension contains the ID of the job.
<code>RemovedJobExecutionTotalCount</code>	The total number of job executions whose status is <code>REMOVED</code> for the given job. The <code>JobId</code> dimension contains the ID of the job.
<code>QueuedJobExecutionCount</code>	The number of job executions whose status has changed to <code>QUEUED</code> within a time period that is determined by CloudWatch. (For more information about CloudWatch metrics, see Amazon CloudWatch Metrics .) The <code>JobId</code> dimension contains the ID of the job.
<code>QueuedJobExecutionTotalCount</code>	The total number of job executions whose status is <code>QUEUED</code> for the given job. The <code>JobId</code> dimension contains the ID of the job.
<code>RejectedJobExecutionCount</code>	The number of job executions whose status has changed to <code>REJECTED</code> within a time period that is determined by CloudWatch. (For more information about CloudWatch metrics, see Amazon CloudWatch Metrics .) The <code>JobId</code> dimension contains the ID of the job.
<code>RemovedJobExecutionCount</code>	The number of job executions whose status has changed to <code>REMOVED</code> within a time period that is determined by CloudWatch. (For more information about CloudWatch metrics, see Amazon CloudWatch Metrics .) The <code>JobId</code> dimension contains the ID of the job.
<code>ServerErrorCount</code>	The number of server errors generated while executing the job. The <code>JobId</code> dimension contains the ID of the job.
<code>SucceededJobExecutionCount</code>	The number of job executions whose status has changed to <code>SUCCESS</code> within a time period that is determined by CloudWatch. (For more information about CloudWatch metrics, see Amazon CloudWatch Metrics .) The <code>JobId</code> dimension contains the ID of the job.
<code>SucceededJobExecutionTotalCount</code>	The total number of job executions whose status is <code>SUCCESS</code> for the given job. The <code>JobId</code> dimension contains the ID of the job.

Device Defender audit metrics

Metric	Description
NonCompliantResources	The number of resources that were found to be noncompliant with a check. The system reports the number of resources that were out of compliance for each check of each audit performed.
ResourcesEvaluated	The number of resources that were evaluated for compliance. The system reports the number of resources that were evaluated for each check of each audit performed.

Device Defender detect metrics

Metric	Description
Violations	The number of new violations of security profile behaviors that have been found since the last time an evaluation was performed. The system reports the number of new violations for the account, for a specific security profile, and for a specific behavior of a specific security profile.
ViolationsCleared	The number of violations of security profile behaviors that have been resolved since the last time an evaluation was performed. The system reports the number of resolved violations for the account, for a specific security profile, and for a specific behavior of a specific security profile.
ViolationsInvalidated	The number of violations of security profile behaviors for which information is no longer available since the last time an evaluation was performed (because the reporting device stopped reporting, or is no longer being monitored for some reason). The system reports the number of invalidated violations for the entire account, for a specific security profile, and for a specific behavior of a specific security profile.

Device provisioning metrics

AWS IoT Fleet provisioning metrics

Metric	Description
ApproximateNumberOfThingsRegistered	The count of things that have been registered by Fleet Provisioning. While the count is generally accurate, the distributed architecture of AWS IoT Core makes it difficult to maintain a precise count of registered things.

Metric	Description
	<p>The statistic to use for this metric is:</p> <ul style="list-style-type: none"> • Max to report the total number of things that have been registered. For a count of things registered during the CloudWatch aggregation window, see the <code>RegisterThingFailed</code> metric. <p>Dimensions: ClaimCertificateId (p. 420)</p>
<code>CreateKeysAndCertificateFailed</code>	<p>The number of failures that occurred by calls to the <code>CreateKeysAndCertificate</code> MQTT API.</p> <p>The metric is emitted in both Success (value = 0) and Failure (value = 1) cases. This metric can be used to track the number of certificates created and registered during the CloudWatch-supported aggregation windows, such as 5 min. or 1 hour.</p> <p>The statistics available for this metric are:</p> <ul style="list-style-type: none"> • Sum to report the number of failed calls. • SampleCount to report the total number of successful and failed calls.
<code>CreateCertificateFromCsrFailed</code>	<p>The number of failures that occurred by calls to the <code>CreateCertificateFromCsr</code> MQTT API.</p> <p>The metric is emitted in both Success (value = 0) and Failure (value = 1) cases. This metric can be used to track the number of things registered during the CloudWatch-supported aggregation windows, such as 5 min. or 1 hour.</p> <p>The statistics available for this metric are:</p> <ul style="list-style-type: none"> • Sum to report the number of failed calls. • SampleCount to report the total number of successful and failed calls.

Metric	Description
RegisterThingFailed	<p>The number of failures that occurred by calls to the RegisterThing MQTT API.</p> <p>The metric is emitted in both Success (value = 0) and Failure (value = 1) cases. This metric can be used to track the number of things registered during the CloudWatch-supported aggregation windows, such as 5 min. or 1 hour. For the total number of things registered, see the ApproximateNumberOfThingsRegistered metric.</p> <p>The statistics available for this metric are:</p> <ul style="list-style-type: none"> • Sum to report the number of failed calls. • SampleCount to report the total number of successful and failed calls. <p>Dimensions: TemplateName (p. 420)</p>

Just-in-time provisioning metrics

Metric	Description
ProvisionThing.ClientError	The number of times a device failed to provision due to a client error. For example, the policy specified in the template did not exist.
ProvisionThing.ServerError	The number of times a device failed to provision due to a server error. Customers can retry to provision the device after waiting and they can contact AWS IoT if the issue remains the same.
ProvisionThing.Success	The number of times a device was successfully provisioned.

Dimensions for metrics

Metrics use the namespace and provide metrics for the following dimensions

Dimension	Description
ActionType	The action type (p. 450) specified by the rule that triggered the request.
BehaviorName	The name of the Device Defender Detect security profile behavior that is being monitored.
ClaimCertificateId	The certificateId of the claim used to provision the devices.
CheckName	The name of the Device Defender audit check whose results are being monitored.

Dimension	Description
JobId	The ID of the job whose progress or message connection success/failure is being monitored.
Protocol	The protocol used to make the request. Valid values are: MQTT or HTTP
RuleName	The name of the rule triggered by the request.
ScheduledAuditName	The name of the Device Defender scheduled audit whose check results are being monitored. This has the value OnDemand if the results reported are for an audit that was performed on demand.
SecurityProfileName	The name of the Device Defender Detect security profile whose behaviors are being monitored.
TemplateName	The name of the provisioning template.

Monitor AWS IoT using CloudWatch Logs

When [AWS IoT logging is enabled \(p. 400\)](#), AWS IoT sends progress events about each message as it passes from your devices through the message broker and rules engine. In the [CloudWatch console](#), CloudWatch logs appear in a log group named **AWSIoTLogs**.

For more information about CloudWatch Logs, see [CloudWatch Logs](#). For information about supported AWS IoT CloudWatch Logs, see [CloudWatch AWS IoT log entries \(p. 422\)](#).

Viewing AWS IoT logs in the CloudWatch console

Note

The **AWSIoTLogsV2** log group is not visible in the CloudWatch console until:

- You've enabled logging in AWS IoT. For more info on how to enable logging in AWS IoT, see [Configure AWS IoT logging \(p. 400\)](#)
- Some log entries have been written by AWS IoT operations.

To view your AWS IoT logs in the CloudWatch console

1. Browse to <https://console.aws.amazon.com/cloudwatch/>. In the navigation pane, choose **Log groups**.
2. In the **Filter** text box, enter **AWSIoTLogsV2**, and then press Enter.
3. Double-click the **AWSIoTLogsV2** log group.
4. Choose **Search All**. A complete list of the AWS IoT logs generated for your account is displayed.
5. Choose the expand icon to look at an individual stream.

You can also enter a query in the **Filter events** text box. Here are some interesting queries to try:

- { `$.logLevel = "INFO"` }
- Find all logs that have a log level of `INFO`.
- { `$.status = "Success"` }

Find all logs that have a status of Success.

```
• { $.status = "Success" && $.eventType = "GetThingShadow" }
```

Find all logs that have a status of Success and an event type of GetThingShadow.

For more information about creating filter expressions, see [CloudWatch Logs Queries](#).

CloudWatch AWS IoT log entries

Each component of AWS IoT generates its own log entries. Each log entry has an `eventType` that specifies the operation that caused the log entry to be generated. This section describes the log entries generated by the following AWS IoT components. For information about AWS IoT Core for LoRaWAN monitoring, see [View CloudWatch AWS IoT Core for LoRaWAN log entries \(p. 1089\)](#).

Topics

- [Message broker log entries \(p. 422\)](#)
- [Device Shadow log entries \(p. 427\)](#)
- [Rules engine log entries \(p. 429\)](#)
- [Job log entries \(p. 434\)](#)
- [Device provisioning log entries \(p. 437\)](#)
- [Dynamic thing group log entries \(p. 438\)](#)
- [Common CloudWatch Logs attributes \(p. 439\)](#)

Message broker log entries

The AWS IoT message broker generates log entries for the following events:

Topics

- [Connect log entry \(p. 422\)](#)
- [Disconnect log entry \(p. 423\)](#)
- [GetRetainedMessage log entry \(p. 424\)](#)
- [ListRetainedMessage log entry \(p. 424\)](#)
- [Publish-In log entry \(p. 425\)](#)
- [Publish-Out log entry \(p. 426\)](#)
- [Subscribe log entry \(p. 426\)](#)

Connect log entry

The AWS IoT message broker generates a log entry with an `eventType` of `Connect` when an MQTT client connects.

Connect log entry example

```
{  
    "timestamp": "2017-08-10 15:37:23.476",  
    "logLevel": "INFO",  
    "traceId": "20b23f3f-d7f1-feae-169f-82263394fbdb",  
    "accountId": "123456789012",  
    "status": "Success",  
    "eventType": "Connect",  
    "protocol": "MQTT",  
}
```

```
{  
    "clientId": "abf27092886e49a8a5c1922749736453",  
    "principalId": "145179c40e2219e18a909d896a5340b74cf97a39641beec2fc3eeafc5a932167",  
    "sourceIp": "205.251.233.181",  
    "sourcePort": 13490  
}
```

In addition to the [Common CloudWatch Logs attributes \(p. 439\)](#), Connect log entries contain the following attributes:

clientId

The ID of the client making the request.

principalId

The ID of the principal making the request.

protocol

The protocol used when making the request. Valid values are **MQTT** or **HTTP**.

sourceIp

The IP address where the request originated.

sourcePort

The port where the request originated.

Disconnect log entry

The AWS IoT message broker generates a log entry with an **eventType** of **Disconnect** when an MQTT client disconnects.

Disconnect log entry example

```
{  
    "timestamp": "2017-08-10 15:37:23.476",  
    "logLevel": "INFO",  
    "traceId": "20b23f3f-d7f1-feae-169f-82263394fbdb",  
    "accountId": "123456789012",  
    "status": "Success",  
    "eventType": "Disconnect",  
    "protocol": "MQTT",  
    "clientId": "abf27092886e49a8a5c1922749736453",  
    "principalId": "145179c40e2219e18a909d896a5340b74cf97a39641beec2fc3eeafc5a932167",  
    "sourceIp": "205.251.233.181",  
    "sourcePort": 13490,  
    "disconnectReason": "CLIENT_INITIATED_DISCONNECT"  
}
```

In addition to the [Common CloudWatch Logs attributes \(p. 439\)](#), Disconnect log entries contain the following attributes:

clientId

The ID of the client making the request.

principalId

The ID of the principal making the request.

protocol

The protocol used when making the request. Valid values are **MQTT** or **HTTP**.

sourceIp

The IP address where the request originated.

sourcePort

The port where the request originated.

disconnectReason

The reason why the client is disconnecting.

GetRetainedMessage log entry

The AWS IoT message broker generates a log entry with an `eventType` of `GetRetainedMessage` when [GetRetainedMessage](#) is called.

GetRetainedMessage log entry example

```
{  
    "timestamp": "2017-08-07 18:47:56.664",  
    "logLevel": "INFO",  
    "traceId": "1a60d02e-15b9-605b-7096-a9f584a6ad3f",  
    "accountId": "123456789012",  
    "status": "Success",  
    "eventType": "GetRetainedMessage",  
    "protocol": "HTTP",  
    "topicName": "a/b/c",  
    "qos": "1",  
    "lastModifiedDate": "2017-08-07 18:47:56.664"  
}
```

In addition to the [Common CloudWatch Logs attributes \(p. 439\)](#), GetRetainedMessage log entries contain the following attributes:

lastModifiedDate

The Epoch date and time, in milliseconds, when the retained message was stored by AWS IoT.

protocol

The protocol used when making the request. Valid value: `HTTP`.

qos

The Quality of Service (QoS) level used in the publish request. Valid values are 0 or 1.

topicName

The name of the subscribed topic.

ListRetainedMessage log entry

The AWS IoT message broker generates a log entry with an `eventType` of `ListRetainedMessage` when [ListRetainedMessages](#) is called.

ListRetainedMessage log entry example

```
{  
    "timestamp": "2017-08-07 18:47:56.664",  
    "logLevel": "INFO",  
}
```

```
"traceId": "1a60d02e-15b9-605b-7096-a9f584a6ad3f",
"accountId": "123456789012",
"status": "Success",
"eventType": "ListRetainedMessage",
"protocol": "HTTP"
}
```

In addition to the [Common CloudWatch Logs attributes \(p. 439\)](#), ListRetainedMessage log entries contains the following attribute:

protocol

The protocol used when making the request. Valid value: HTTP.

Publish-In log entry

When the AWS IoT message broker receives an MQTT message, it generates a log entry with an eventType of Publish-In.

Publish-In log entry example

```
{
    "timestamp": "2017-08-10 15:39:30.961",
    "logLevel": "INFO",
    "traceId": "672ec480-31ce-fd8b-b5fb-22e3ac420699",
    "accountId": "123456789012",
    "status": "Success",
    "eventType": "Publish-In",
    "protocol": "MQTT",
    "topicName": "$aws/things/MyThing/shadow/get",
    "clientId": "abf27092886e49a8a5c1922749736453",
    "principalId": "145179c40e2219e18a909d896a5340b74cf97a39641beec2fc3eeafc5a932167",
    "sourceIp": "205.251.233.181",
    "sourcePort": 13490
}
```

In addition to the [Common CloudWatch Logs attributes \(p. 439\)](#), Publish-In log entries contain the following attributes:

clientId

The ID of the client making the request.

principalId

The ID of the principal making the request.

protocol

The protocol used when making the request. Valid values are MQTT or HTTP.

sourceIp

The IP address where the request originated.

sourcePort

The port where the request originated.

topicName

The name of the subscribed topic.

Publish-Out log entry

When the message broker publishes an MQTT message, it generates a log entry with an `eventType` of `Publish-Out`

Publish-Out log entry example

```
{  
    "timestamp": "2017-08-10 15:39:30.961",  
    "logLevel": "INFO",  
    "traceId": "672ec480-31ce-fd8b-b5fb-22e3ac420699",  
    "accountId": "123456789012",  
    "status": "Success",  
    "eventType": "Publish-Out",  
    "protocol": "MQTT",  
    "topicName": "$aws/things/MyThing/shadow/get",  
    "clientId": "abf27092886e49a8a5c1922749736453",  
    "principalId": "145179c40e2219e18a909d896a5340b74cf97a39641beec2fc3eeafc5a932167",  
    "sourceIp": "205.251.233.181",  
    "sourcePort": 13490  
}
```

In addition to the [Common CloudWatch Logs attributes \(p. 439\)](#), Publish-Out log entries contain the following attributes:

`clientId`

The ID of the subscribed client that receives messages on that MQTT topic.

`principalId`

The ID of the principal making the request.

`protocol`

The protocol used when making the request. Valid values are `MQTT` or `HTTP`.

`sourceIp`

The IP address where the request originated.

`sourcePort`

The port where the request originated.

`topicName`

The name of the subscribed topic.

Subscribe log entry

The AWS IoT message broker generates a log entry with an `eventType` of `Subscribe` when an MQTT client subscribes to a topic.

Subscribe log entry example

```
{  
    "timestamp": "2017-08-10 15:39:04.413",  
    "logLevel": "INFO",  
    "traceId": "7aa5c38d-1b49-3753-15dc-513ce4ab9fa6",  
    "accountId": "123456789012",  
    "status": "Success",  
    "eventType": "Subscribe",  
    "topic": "testTopic"  
}
```

```
    "protocol": "MQTT",
    "topicName": "$aws/things/MyThing/shadow/#",
    "clientId": "abf27092886e49a8a5c1922749736453",
    "principalId": "145179c40e2219e18a909d896a5340b74cf97a39641beec2fc3eeafc5a932167",
    "sourceIp": "205.251.233.181",
    "sourcePort": 13490
}
```

In addition to the [Common CloudWatch Logs attributes \(p. 439\)](#), Subscribe log entries contain the following attributes:

clientId

The ID of the client making the request.

principalId

The ID of the principal making the request.

protocol

The protocol used when making the request. Valid values are MQTT or HTTP.

sourceIp

The IP address where the request originated.

sourcePort

The port where the request originated.

topicName

The name of the subscribed topic.

Device Shadow log entries

The AWS IoT Device Shadow service generates log entries for the following events:

Topics

- [DeleteThingShadow log entry \(p. 427\)](#)
- [GetThingShadow log entry \(p. 428\)](#)
- [UpdateThingShadow log entry \(p. 428\)](#)

DeleteThingShadow log entry

The Device Shadow service generates a log entry with an `eventType` of DeleteThingShadow when a request to delete a device's shadow is received.

DeleteThingShadow log entry example

```
{
    "timestamp": "2017-08-07 18:47:56.664",
    "logLevel": "INFO",
    "traceId": "1a60d02e-15b9-605b-7096-a9f584a6ad3f",
    "accountId": "123456789012",
    "status": "Success",
    "eventType": "DeleteThingShadow",
    "protocol": "MQTT",
    "deviceShadowName": "Jack",
    "topicName": "$aws/things/Jack/shadow/delete"
```

```
}
```

In addition to the [Common CloudWatch Logs attributes \(p. 439\)](#), `DeleteThingShadow` log entries contain the following attributes:

`deviceShadowName`

The name of the shadow to update.

`protocol`

The protocol used when making the request. Valid values are `MQTT` or `HTTP`.

`topicName`

The name of the topic on which the request was published.

GetThingShadow log entry

The Device Shadow service generates a log entry with an `eventType` of `GetThingShadow` when a get request for a shadow is received.

GetThingShadow log entry example

```
{
  "timestamp": "2017-08-09 17:56:30.941",
  "logLevel": "INFO",
  "traceId": "b575f19a-97a2-cf72-0ed0-c64a783a2504",
  "accountId": "123456789012",
  "status": "Success",
  "eventType": "GetThingShadow",
  "protocol": "MQTT",
  "deviceShadowName": "MyThing",
  "topicName": "$aws/things/MyThing/shadow/get"
}
```

In addition to the [Common CloudWatch Logs attributes \(p. 439\)](#), `GetThingShadow` log entries contain the following attributes:

`deviceShadowName`

The name of the requested shadow.

`protocol`

The protocol used when making the request. Valid values are `MQTT` or `HTTP`.

`topicName`

The name of the topic on which the request was published.

UpdateThingShadow log entry

The Device Shadow service generates a log entry with an `eventType` of `UpdateThingShadow` when a request to update a device's shadow is received.

UpdateThingShadow log entry example

```
{
  "timestamp": "2017-08-07 18:43:59.436",
```

```
    "logLevel": "INFO",
    "traceId": "d0074ba8-0c4b-a400-69df-76326d414c28",
    "accountId": "123456789012",
    "status": "Success",
    "eventType": "UpdateThingShadow",
    "protocol": "MQTT",
    "deviceShadowName": "Jack",
    "topicName": "$aws/things/Jack/shadow/update"
}
```

In addition to the [Common CloudWatch Logs attributes \(p. 439\)](#), `UpdateThingShadow` log entries contain the following attributes:

`deviceShadowName`

The name of the shadow to update.

`protocol`

The protocol used when making the request. Valid values are `MQTT` or `HTTP`.

`topicName`

The name of the topic on which the request was published.

Rules engine log entries

The AWS IoT rules engine generates logs for the following events:

Topics

- [FunctionExecution log entry \(p. 429\)](#)
- [RuleExecution log entry \(p. 430\)](#)
- [RuleMatch log entry \(p. 431\)](#)
- [RuleMessageThrottled log entry \(p. 431\)](#)
- [RuleNotFound log entry \(p. 432\)](#)
- [StartingRuleExecution log entry \(p. 433\)](#)

FunctionExecution log entry

The rules engine generates a log entry with an `eventType` of `FunctionExecution` when a rule's SQL query calls an external function. An external function is called when a rule's action makes an HTTP request to AWS IoT or another web service (for example, calling `get_thing_shadow` or `machinelearning_predict`).

FunctionExecution log entry example

```
{
    "timestamp": "2017-07-13 18:33:51.903",
    "logLevel": "DEBUG",
    "traceId": "180532b7-0cc7-057b-687a-5ca1824838f5",
    "status": "Success",
    "eventType": "FunctionExecution",
    "clientId": "N/A",
    "topicName": "rules/test",
    "ruleName": "ruleTestPredict",
    "ruleAction": "MachinelearningPredict",
    "resources": {
        "ModelId": "predict-model"
    }
}
```

```
    },
    "principalId": "145179c40e2219e18a909d896a5340b74cf97a39641beec2fc3eeafc5a932167"
}
```

In addition to the [Common CloudWatch Logs attributes \(p. 439\)](#), `FunctionExecution` log entries contain the following attributes:

`clientId`

N/A for `FunctionExecution` logs.

`principalId`

The ID of the principal making the request.

`resources`

A collection of resources used by the rule's actions.

`ruleName`

The name of the matching rule.

`topicName`

The name of the subscribed topic.

RuleExecution log entry

When the AWS IoT rules engine triggers a rule's action, it generates a `RuleExecution` log entry.

RuleExecution log entry example

```
{
    "timestamp": "2017-08-10 16:32:46.070",
    "logLevel": "INFO",
    "traceId": "30aa7ccc-1d23-0b97-aa7b-76196d83537e",
    "accountId": "123456789012",
    "status": "Success",
    "eventType": "RuleExecution",
    "clientId": "abf27092886e49a8a5c1922749736453",
    "topicName": "rules/test",
    "ruleName": "JSONLogsRule",
    "ruleAction": "RepublishAction",
    "resources": {
        "RepublishTopic": "rules/republish"
    },
    "principalId": "145179c40e2219e18a909d896a5340b74cf97a39641beec2fc3eeafc5a932167"
}
```

In addition to the [Common CloudWatch Logs attributes \(p. 439\)](#), `RuleExecution` log entries contain the following attributes:

`clientId`

The ID of the client making the request.

`principalId`

The ID of the principal making the request.

`resources`

A collection of resources used by the rule's actions.

ruleAction

The name of the action triggered.

ruleName

The name of the matching rule.

topicName

The name of the subscribed topic.

RuleMatch log entry

The AWS IoT rules engine generates a log entry with an `eventType` of `RuleMatch` when the message broker receives a message that matches a rule.

RuleMatch log entry example

```
{  
    "timestamp": "2017-08-10 16:32:46.002",  
    "logLevel": "INFO",  
    "traceId": "30aa7ccc-1d23-0b97-aa7b-76196d83537e",  
    "accountId": "123456789012",  
    "status": "Success",  
    "eventType": "RuleMatch",  
    "clientId": "abf27092886e49a8a5c1922749736453",  
    "topicName": "rules/test",  
    "ruleName": "JSONLogsRule",  
    "principalId": "145179c40e2219e18a909d896a5340b74cf97a39641beec2fc3eeafc5a932167"  
}
```

In addition to the [Common CloudWatch Logs attributes \(p. 439\)](#), RuleMatch log entries contain the following attributes:

clientId

The ID of the client making the request.

principalId

The ID of the principal making the request.

ruleName

The name of the matching rule.

topicName

The name of the subscribed topic.

RuleMessageThrottled log entry

When a message is throttled, the AWS IoT rules engine generates a log entry with an `eventType` of `RuleMessageThrottled`.

RuleMessageThrottled log entry example

```
{  
    "timestamp": "2017-10-04 19:25:46.070",  
    "logLevel": "ERROR",
```

```

    "traceId": "30aa7ccc-1d23-0b97-aa7b-76196d83537e",
    "accountId": "123456789012",
    "status": "Failure",
    "eventType": "RuleMessageThrottled",
    "clientId": "abf27092886e49a8a5c1922749736453",
    "topicName": "$aws/rules/example_rule",
    "ruleName": "example_rule",
    "principalId": "145179c40e2219e18a909d896a5340b74cf97a39641beec2fc3eeafc5a932167",
    "reason": "RuleExecutionThrottled",
    "details": "Message for Rule example_rule throttled"
}

```

In addition to the [Common CloudWatch Logs attributes \(p. 439\)](#), RuleMessageThrottled log entries contain the following attributes:

clientId

The ID of the client making the request.

details

A brief explanation of the error.

principalId

The ID of the principal making the request.

reason

The string "RuleMessageThrottled".

ruleName

The name of the rule to be triggered.

topicName

The name of the topic that was published.

RuleNotFound log entry

When the AWS IoT rules engine cannot find a rule with a given name, it generates a log entry with an eventType of RuleNotFound.

RuleNotFound log entry example

```

{
    "timestamp": "2017-10-04 19:25:46.070",
    "logLevel": "ERROR",
    "traceId": "30aa7ccc-1d23-0b97-aa7b-76196d83537e",
    "accountId": "123456789012",
    "status": "Failure",
    "eventType": "RuleNotFound",
    "clientId": "abf27092886e49a8a5c1922749736453",
    "topicName": "$aws/rules/example_rule",
    "ruleName": "example_rule",
    "principalId": "145179c40e2219e18a909d896a5340b74cf97a39641beec2fc3eeafc5a932167",
    "reason": "RuleNotFound",
    "details": "Rule example_rule not found"
}

```

In addition to the [Common CloudWatch Logs attributes \(p. 439\)](#), RuleNotFound log entries contain the following attributes:

clientId

The ID of the client making the request.

details

A brief explanation of the error.

principalId

The ID of the principal making the request.

reason

The string "RuleNotFound".

ruleName

The name of the rule that could not be found.

topicName

The name of the topic that was published.

StartingRuleExecution log entry

When the AWS IoT rules engine starts to trigger a rule's action, it generates a log entry with an eventType of StartingRuleExecution.

StartingRuleExecution log entry example

```
{  
    "timestamp": "2017-08-10 16:32:46.002",  
    "logLevel": "DEBUG",  
    "traceId": "30aa7ccc-1d23-0b97-aa7b-76196d83537e",  
    "accountId": "123456789012",  
    "status": "Success",  
    "eventType": "StartingRuleExecution",  
    "clientId": "abf27092886e49a8a5c1922749736453",  
    "topicName": "rules/test",  
    "ruleName": "JSONLogsRule",  
    "ruleAction": "RepublishAction",  
    "principalId": "145179c40e2219e18a909d896a5340b74cf97a39641beec2fc3eeafc5a932167"  
}
```

In addition to the [Common CloudWatch Logs attributes \(p. 439\)](#), rule- log entries contain the following attributes:

clientId

The ID of the client making the request.

principalId

The ID of the principal making the request.

ruleAction

The name of the action triggered.

ruleName

The name of the matching rule.

topicName

The name of the subscribed topic.

Job log entries

The AWS IoT Job service generates log entries for the following events. Log entries are generated when an MQTT or HTTP request is received from the device.

Topics

- [DescribeJobExecution log entry \(p. 434\)](#)
- [GetPendingJobExecution log entry \(p. 435\)](#)
- [ReportFinalJobExecutionCount log entry \(p. 435\)](#)
- [StartNextPendingJobExecution log entry \(p. 436\)](#)
- [UpdateJobExecution log entry \(p. 436\)](#)

DescribeJobExecution log entry

The AWS IoT Jobs service generates a log entry with an `eventType` of `DescribeJobExecution` when the service receives a request to describe a job execution.

DescribeJobExecution log entry example

```
{  
    "timestamp": "2017-08-10 19:13:22.841",  
    "logLevel": "DEBUG",  
    "accountId": "123456789012",  
    "status": "Success",  
    "eventType": "DescribeJobExecution",  
    "protocol": "MQTT",  
    "clientId": "thingOne",  
    "jobId": "002",  
    "topicName": "$aws/things/thingOne/jobs/002/get",  
    "clientToken": "myToken",  
    "details": "The request status is SUCCESS."  
}
```

In addition to the [Common CloudWatch Logs attributes \(p. 439\)](#), `GetJobExecution` log entries contain the following attributes:

`clientId`

The ID of the client making the request.

`clientToken`

A unique, case-sensitive identifier to ensure the idempotency of the request. For more information, see [How to Ensure Idempotency](#).

`details`

Other information from the Jobs service.

`jobId`

The job ID for the job execution.

`protocol`

The protocol used when making the request. Valid values are `MQTT` or `HTTP`.

`topicName`

The topic used to make the request.

GetPendingJobExecution log entry

The AWS IoT Jobs service generates a log entry with an `eventType` of `GetPendingJobExecution` when the service receives a job execution request.

GetPendingJobExecution log entry example

```
{  
    "timestamp": "2018-06-13 17:45:17.197",  
    "logLevel": "DEBUG",  
    "accountId": "123456789012",  
    "status": "Success",  
    "eventType": "GetPendingJobExecution",  
    "protocol": "MQTT",  
    "clientId": "299966ad-54de-40b4-99d3-4fc8b52da0c5",  
    "topicName": "$aws/things/299966ad-54de-40b4-99d3-4fc8b52da0c5/jobs/get",  
    "clientToken": "24b9a741-15a7-44fc-bd3c-1ff2e34e5e82",  
    "details": "The request status is SUCCESS."  
}
```

In addition to the [Common CloudWatch Logs attributes \(p. 439\)](#), `GetPendingJobExecution` log entries contain the following attributes:

`clientId`

The ID of the client making the request.

`clientToken`

A unique, case sensitive identifier to ensure the idempotency of the request. For more information, see [How to Ensure Idempotency](#).

`details`

Other information from the Jobs service.

`protocol`

The protocol used when making the request. Valid values are `MQTT` or `HTTP`.

`topicName`

The name of the subscribed topic.

ReportFinalJobExecutionCount log entry

The AWS IoT Jobs service generates a log entry with an `entryType` of `ReportFinalJobExecutionCount` when a job is completed.

ReportFinalJobExecutionCount log entry example

```
{  
    "timestamp": "2017-08-10 19:44:16.776",  
    "logLevel": "INFO",  
    "accountId": "123456789012",  
    "status": "Success",  
    "eventType": "ReportFinalJobExecutionCount",  
    "jobId": "002",  
    "details": "Job 002 completed. QUEUED job execution count: 0 IN_PROGRESS job execution count: 0 FAILED job execution count: 0 SUCCEEDED job execution count: 1 CANCELED job execution count: 0 REJECTED job execution count: 0 REMOVED job execution count: 0"  
}
```

In addition to the [Common CloudWatch Logs attributes \(p. 439\)](#), `ReportFinalJobExecutionCount` log entries contain the following attributes:

`details`

Other information from the Jobs service.

`jobId`

The job ID for the job execution.

StartNextPendingJobExecution log entry

When it receives a request to start the next pending job execution, the AWS IoT Jobs service generates a log entry with an `eventType` of `StartNextPendingJobExecution`.

StartNextPendingJobExecution log entry example

```
{  
    "timestamp": "2018-06-13 17:49:51.036",  
    "logLevel": "DEBUG",  
    "accountId": "123456789012",  
    "status": "Success",  
    "eventType": "StartNextPendingJobExecution",  
    "protocol": "MQTT",  
    "clientId": "95c47808-b1ca-4794-bc68-a588d6d9216c",  
    "topicName": "$aws/things/95c47808-b1ca-4794-bc68-a588d6d9216c/jobs/start-next",  
    "clientToken": "bd7447c4-3a05-49f4-8517-dd89b2c68d94",  
    "details": "The request status is SUCCESS."  
}
```

In addition to the [Common CloudWatch Logs attributes \(p. 439\)](#), `StartNextPendingJobExecution` log entries contain the following attributes:

`clientId`

The ID of the client making the request.

`clientToken`

A unique, case sensitive identifier to ensure the idempotency of the request. For more information, see [How to Ensure Idempotency](#).

`details`

Other information from the Jobs service.

`protocol`

The protocol used when making the request. Valid values are `MQTT` or `HTTP`.

`topicName`

The topic used to make the request.

UpdateJobExecution log entry

The AWS IoT Jobs service generates a log entry with an `eventType` of `UpdateJobExecution` when the service receives a request to update a job execution.

UpdateJobExecution log entry example

```
{
```

```
"timestamp": "2017-08-10 19:25:14.758",
"LogLevel": "DEBUG",
"accountId": "123456789012",
"status": "Success",
"eventType": "UpdateJobExecution",
"protocol": "MQTT",
"clientId": "thingOne",
"jobId": "002",
"topicName": "$aws/things/thingOne/jobs/002/update",
"clientToken": "myClientToken",
"versionNumber": "1",
"details": "The destination status is IN_PROGRESS. The request status is SUCCESS."
}
```

In addition to the [Common CloudWatch Logs attributes \(p. 439\)](#), `UpdateJobExecution` log entries contain the following attributes:

clientId

The ID of the client making the request.

clientToken

A unique, case sensitive identifier to ensure the idempotency of the request. For more information, see [How to Ensure Idempotency](#).

details

Other information from the Jobs service.

jobId

The job ID for the job execution.

protocol

The protocol used when making the request. Valid values are `MQTT` or `HTTP`.

topicName

The topic used to make the request.

versionNumber

The version of the job execution.

Device provisioning log entries

The AWS IoT Device Provisioning service generates logs for the following events.

Topics

- [GetDeviceCredentials log entry \(p. 437\)](#)
- [ProvisionDevice log entry \(p. 438\)](#)

GetDeviceCredentials log entry

The AWS IoT Device Provisioning service generates a log entry with an `eventType` of `GetDeviceCredential` when a client calls `GetDeviceCredential`.

GetDeviceCredentials log entry example

```
{
```

```
  "timestamp" : "2019-02-20 20:31:22.932",
  "logLevel" : "INFO",
  "traceId" : "8d9c016f-6cc7-441e-8909-7ee3d5563405",
  "accountId" : "123456789101",
  "status" : "Success",
  "eventType" : "GetDeviceCredentials",
  "deviceCertificateId" :
  "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855",
  "details" : "Additional details about this log."
}
```

In addition to the [Common CloudWatch Logs attributes \(p. 439\)](#), `GetDeviceCredentials` log entries contain the following attributes:

`details`

A brief explanation of the error.

`deviceCertificateId`

The ID of the device certificate.

ProvisionDevice log entry

The AWS IoT Device Provisioning service generates a log entry with an `eventType` of `ProvisionDevice` when a client calls `ProvisionDevice`.

ProvisionDevice log entry example

```
{
  "timestamp" : "2019-02-20 20:31:22.932",
  "logLevel" : "INFO",
  "traceId" : "8d9c016f-6cc7-441e-8909-7ee3d5563405",
  "accountId" : "123456789101",
  "status" : "Success",
  "eventType" : "ProvisionDevice",
  "provisioningTemplateName" : "myTemplate",
  "deviceCertificateId" :
  "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855",
  "details" : "Additional details about this log."
}
```

In addition to the [Common CloudWatch Logs attributes \(p. 439\)](#), `ProvisionDevice` log entries contain the following attributes:

`details`

A brief explanation of the error.

`deviceCertificateId`

The ID of the device certificate.

`provisioningTemplateName`

The name of the provisioning template.

Dynamic thing group log entries

AWS IoT Dynamic Thing Groups generate logs for the following event.

Topics

- [AddThingToDynamicThingGroupsFailed log entry \(p. 439\)](#)

AddThingToDynamicThingGroupsFailed log entry

When AWS IoT was not able to add a thing to the specified dynamic groups, it generates a log entry with an eventType of AddThingToDynamicThingGroupsFailed. This happens when a thing met the criteria to be in the dynamic thing group; however, it could not be added to the dynamic group or it was removed from the dynamic group. This can happen because:

- The thing already belongs to the maximum number of groups.
- The **--override-dynamic-groups** option was used to add the thing to a static thing group. It was removed from a dynamic thing group to make that possible.

For more information, see [Dynamic Thing Group Limitations and Conflicts \(p. 270\)](#).

AddThingToDynamicThingGroupsFailed log entry example

This example shows the log entry of an AddThingToDynamicThingGroupsFailed error. In this example, *TestThing* met the criteria to be in the dynamic thing groups listed in *dynamicThingGroupNames*, but could not be added to those dynamic groups, as described in *reason*.

```
{  
  "timestamp": "2020-03-16 22:24:43.804",  
  "logLevel": "ERROR",  
  "traceId": "70b1f2f5-d95e-f897-9dcc-31e68c3e1a30",  
  "accountId": "57EXAMPLE833",  
  "status": "Failure",  
  "eventType": "AddThingToDynamicThingGroupsFailed",  
  "thingName": "TestThing",  
  "dynamicThingGroupNames": [  
    "DynamicThingGroup11",  
    "DynamicThingGroup12",  
    "DynamicThingGroup13",  
    "DynamicThingGroup14"  
  ],  
  "reason": "The thing failed to be added to the given dynamic thing group(s) because the  
  thing already belongs to the maximum allowed number of groups."  
}
```

In addition to the [Common CloudWatch Logs attributes \(p. 439\)](#), AddThingToDynamicThingGroupsFailed log entries contain the following attributes:

dynamicThingGroupNames

An array of the dynamic thing groups to which the thing could not be added.

reason

The reason why the thing could not be added to the dynamic thing groups.

thingName

The name of the thing that could not be added to a dynamic thing group.

Common CloudWatch Logs attributes

All CloudWatch Logs log entries include these attributes:

accountId

Your AWS account ID.

eventType

The event type for which the log was generated. The value of the event type depends on the event that generated the log entry. Each log entry description includes the value of eventType for that log entry.

logLevel

The log level being used. For more information, see [the section called "Log levels" \(p. 406\)](#).

status

The status of the request.

timestamp

The UNIX timestamp of when the client connected to the AWS IoT message broker.

traceId

A randomly generated identifier that can be used to correlate all logs for a specific request.

Log AWS IoT API calls using AWS CloudTrail

AWS IoT is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in AWS IoT. CloudTrail captures all API calls for AWS IoT as events, including calls from the AWS IoT console and from code calls to the AWS IoT APIs. If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon S3 bucket, including events for AWS IoT. If you don't configure a trail, you can still view the most recent events in the CloudTrail console in **Event history**. Using the information collected by CloudTrail, you can determine the request that was made to AWS IoT, the IP address from which the request was made, who made the request, when it was made, and other details.

To learn more about CloudTrail, see the [AWS CloudTrail User Guide](#).

AWS IoT information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When activity occurs in AWS IoT, that activity is recorded in a CloudTrail event along with other AWS service events in **Event history**. You can view, search, and download recent events in your AWS account. For more information, see [Viewing Events with CloudTrail Event History](#).

For an ongoing record of events in your AWS account, including events for AWS IoT, create a trail. A trail enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all AWS Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. You can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs. For more information, see:

- [Overview for Creating a Trail](#)
- [CloudTrail Supported Services and Integrations](#)
- [Configuring Amazon SNS Notifications for CloudTrail](#)
- [Receiving CloudTrail Log Files from Multiple Regions](#) and [Receiving CloudTrail Log Files from Multiple Accounts](#)

Note

AWS IoT data plane actions (device side) are not logged by CloudTrail. Use CloudWatch to monitor these actions.

Generally speaking, AWS IoT control plane actions that make changes are logged by CloudTrail. Calls such as **CreateThing**, **CreateKeysAndCertificate**, and **UpdateCertificate** leave CloudTrail entries, while calls such as **ListThings** and **ListTopicRules** do not.

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root or IAM user credentials.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

For more information, see the [CloudTrail userIdentity Element](#).

AWS IoT actions are documented in the [AWS IoT API Reference](#). AWS IoT Wireless actions are documented in the [AWS IoT Wireless API Reference](#).

Understanding AWS IoT log file entries

A trail is a configuration that enables delivery of events as log files to an Amazon S3 bucket that you specify. CloudTrail log files contain one or more log entries. An event represents a single request from any source and includes information about the requested action, the date and time of the action, request parameters, and so on. CloudTrail log files are not an ordered stack trace of the public API calls, so they do not appear in any specific order.

The following example shows a CloudTrail log entry that demonstrates the **AttachPolicy** action.

```
{  
    "timestamp": "1460159496",  
    "AdditionalEventData": "",  
    "Annotation": "",  
    "ApiVersion": "",  
    "ErrorCode": "",  
    "ErrorMessage": "",  
    "EventID": "8bfff4fed-c229-4d2d-8264-4ab28a487505",  
    "EventName": "AttachPolicy",  
    "EventTime": "2016-04-08T23:51:36Z",  
    "EventType": "AwsApiCall",  
    "ReadOnly": "",  
    "RecipientAccountList": "",  
    "RequestID": "d4875df2-fde4-11e5-b829-23bf9b56cbcd",  
    "RequestParamters": {  
        "principal": "arn:aws:iot:us-  
east-1:123456789012:cert/528ce36e8047f6a75ee51ab7beddb4eb268ad41d2ea881a10b67e8e76924d894",  
        "policyName": "ExamplePolicyForIoT"  
    },  
    "Resources": "",  
    "ResponseElements": "",  
    "SourceIpAddress": "52.90.213.26",  
    "UserAgent": "aws-internal/3",  
    "UserIdentity": {  
        "type": "AssumedRole",  
        "principalId": "AKIAI44QH8DHBEEXAMPLE",  
        "arn": "arn:aws:sts::12345678912:assumed-role/iotmonitor-us-east-1-beta-  
InstanceRole-1C5T1YCYMHPYT/i-35d0a4b6",  
        "accountId": "222222222222",  
    }  
}
```

```
"accessKeyId":"access-key-id",
"sessionContext":{
    "attributes":{
        "mfaAuthenticated":"false",
        "creationDate":"Fri Apr 08 23:51:10 UTC 2016"
    },
    "sessionIssuer":{
        "type":"Role",
        "principalId":"AKIAI44QH8DHBEXAMPLE",
        "arn":"arn:aws:iam::123456789012:role/executionServiceEC2Role/iotmonitor-
us-east-1-beta-InstanceRole-1C5T1YCYMHPYT",
        "accountId":"222222222222",
        "userName":"iotmonitor-us-east-1-InstanceRole-1C5T1YCYMHPYT"
    }
},
"invokedBy":{
    "serviceAccountId":"111111111111"
}
},
"VpcEndpointId":""}
```

Rules for AWS IoT

Rules give your devices the ability to interact with AWS services. Rules are analyzed and actions are performed based on the MQTT topic stream. You can use rules to support tasks like these:

- Augment or filter data received from a device.
- Write data received from a device to an Amazon DynamoDB database.
- Save a file to Amazon S3.
- Send a push notification to all users using Amazon SNS.
- Publish data to an Amazon SQS queue.
- Invoke a Lambda function to extract data.
- Process messages from a large number of devices using Amazon Kinesis.
- Send data to the Amazon OpenSearch Service.
- Capture a CloudWatch metric.
- Change a CloudWatch alarm.
- Send the data from an MQTT message to Amazon Machine Learning to make predictions based on an Amazon ML model.
- Send a message to a Salesforce IoT Input Stream.
- Send message data to an AWS IoT Analytics channel.
- Start execution of a Step Functions state machine.
- Send message data to an AWS IoT Events input.
- Send message data an asset property in AWS IoT SiteWise.
- Send message data to a web application or service.

Your rules can use MQTT messages that pass through the publish/subscribe protocol supported by the [the section called “Device communication protocols” \(p. 77\)](#) or, using the [Basic Ingest \(p. 528\)](#) feature, you can securely send device data to the AWS services listed above without incurring [messing costs](#). (The [Basic Ingest \(p. 528\)](#) feature optimizes data flow by removing the publish/subscribe message broker from the ingestion path, so it is more cost effective while keeping the security and data processing features of AWS IoT.)

Before AWS IoT can perform these actions, you must grant it permission to access your AWS resources on your behalf. When the actions are performed, you incur the standard charges for the AWS services you use.

Contents

- [Granting AWS IoT the required access \(p. 444\)](#)
- [Pass role permissions \(p. 445\)](#)
- [Creating an AWS IoT rule \(p. 446\)](#)
- [Viewing your rules \(p. 450\)](#)
- [Deleting a rule \(p. 450\)](#)
- [AWS IoT rule actions \(p. 450\)](#)
- [Troubleshooting a rule \(p. 521\)](#)
- [Accessing cross-account resources using AWS IoT rules \(p. 521\)](#)

- [Error handling \(error action\) \(p. 526\)](#)
- [Reducing messaging costs with basic ingest \(p. 528\)](#)
- [AWS IoT SQL reference \(p. 529\)](#)

Granting AWS IoT the required access

You use IAM roles to control the AWS resources to which each rule has access. Before you create a rule, you must create an IAM role with a policy that allows access to the required AWS resources. AWS IoT assumes this role when executing a rule.

To create an IAM role (AWS CLI)

1. Save the following trust policy document, which grants AWS IoT permission to assume the role, to a file named `iot-role-trust.json`.

This example includes a global condition context key to protect against the [confused deputy problem \(p. 323\)](#). For AWS IoT rules, your `aws:SourceArn` must comply with the format: `arn:aws:iot:<region>:<account-id>:<*>`. Make sure that `region` matches your AWS IoT Region and `account-id` matches your customer account ID.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Principal": {  
                "Service": "iot.amazonaws.com"  
            },  
            "Action": "sts:AssumeRole",  
            "Condition": {  
                "StringEquals": {  
                    "aws:SourceAccount": "123456789012"  
                },  
                "ArnLike": {  
                    "aws:SourceArn": "arn:aws:iot:us-east-1:123456789012:role/my-iot-role"  
                }  
            }  
        }  
    ]  
}
```

Use the `create-role` command to create an IAM role specifying the `iot-role-trust.json` file:

```
aws iam create-role --role-name my-iot-role --assume-role-policy-document file://iot-role-trust.json
```

The output of this command looks like the following:

```
{  
    "Role": {  
        "AssumeRolePolicyDocument": "url-encoded-json",  
        "RoleId": "AKIAIOSFODNN7EXAMPLE",  
        "CreateDate": "2015-09-30T18:43:32.821Z",  
        "RoleName": "my-iot-role",  
        "Path": "/",  
        "Arn": "arn:aws:iam::123456789012:role/my-iot-role"  
    }  
}
```

2. Save the following JSON into a file named `my-iot-policy.json`.

```
{  
    "Version": "2012-10-17",  
    "Statement": [{  
        "Effect": "Allow",  
        "Action": "dynamodb:*",  
        "Resource": "*"  
    }]  
}
```

This JSON is an example policy document that grants AWS IoT administrator access to DynamoDB.

Use the [create-policy](#) command to grant AWS IoT access to your AWS resources upon assuming the role, passing in the `my-iot-policy.json` file:

```
aws iam create-policy --policy-name my-iot-policy --policy-document file://my-iot-policy.json
```

For more information about how to grant access to AWS services in policies for AWS IoT, see [Creating an AWS IoT rule \(p. 446\)](#).

The output of the [create-policy](#) command contains the ARN of the policy. You need to attach the policy to a role.

```
{  
    "Policy": {  
        "PolicyName": "my-iot-policy",  
        "CreateDate": "2015-09-30T19:31:18.620Z",  
        "AttachmentCount": 0,  
        "IsAttachable": true,  
        "PolicyId": "ZXR6A36LTYANPAI7NJ5UV",  
        "DefaultVersionId": "v1",  
        "Path": "/",  
        "Arn": "arn:aws:iam::123456789012:policy/my-iot-policy",  
        "UpdateDate": "2015-09-30T19:31:18.620Z"  
    }  
}
```

3. Use the [attach-role-policy](#) command to attach your policy to your role:

```
aws iam attach-role-policy --role-name my-iot-role --policy-arn  
"arn:aws:iam::123456789012:policy/my-iot-policy"
```

Pass role permissions

Part of a rule definition is an IAM role that grants permission to access resources specified in the rule's action. The rules engine assumes that role when the rule's action is triggered. The role must be defined in the same AWS account as the rule.

When creating or replacing a rule you are, in effect, passing a role to the rules engine. The user performing this operation requires the `iam:PassRole` permission. To ensure you have this permission, create a policy that grants the `iam:PassRole` permission and attach it to your IAM user. The following policy shows how to allow `iam:PassRole` permission for a role.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {
```

```
{  
    "Sid": "Stmt1",  
    "Effect": "Allow",  
    "Action": [  
        "iam:PassRole"  
    ],  
    "Resource": [  
        "arn:aws:iam::123456789012:role/myRole"  
    ]  
}  
]  
}
```

In this policy example, the `iam:PassRole` permission is granted for the role `myRole`. The role is specified using the role's ARN. You must attach this policy to your IAM user or role to which your user belongs. For more information, see [Working with Managed Policies](#).

Note

Lambda functions use resource-based policy, where the policy is attached directly to the Lambda function itself. When you create a rule that invokes a Lambda function, you do not pass a role, so the user creating the rule does not need the `iam:PassRole` permission. For more information about Lambda function authorization, see [Granting Permissions Using a Resource Policy](#).

Creating an AWS IoT rule

You configure rules to route data from your connected things. Rules consist of the following:

Rule name

The name of the rule.

Note

We do not recommend the use of personally identifiable information in your rule names.

Optional description

A textual description of the rule.

Note

We do not recommend the use of personally identifiable information in your rule descriptions.

SQL statement

A simplified SQL syntax to filter messages received on an MQTT topic and push the data elsewhere. For more information, see [AWS IoT SQL reference \(p. 529\)](#).

SQL version

The version of the SQL rules engine to use when evaluating the rule. Although this property is optional, we strongly recommend that you specify the SQL version. The AWS IoT Core console sets this property to `2016-03-23` by default. If this property is not set, such as in an AWS CLI command or an AWS CloudFormation template, `2015-10-08` is used. For more information, see [SQL versions \(p. 589\)](#).

One or more actions

The actions AWS IoT performs when executing the rule. For example, you can insert data into a DynamoDB table, write data to an Amazon S3 bucket, publish to an Amazon SNS topic, or invoke a Lambda function.

An error action

The action AWS IoT performs when it is unable to perform a rule's action.

When you create a rule, be aware of how much data you are publishing on topics. If you create rules that include a wildcard topic pattern, they might match a large percentage of your messages, and you might need to increase the capacity of the AWS resources used by the target actions. Also, if you create a republish rule that includes a wildcard topic pattern, you can end up with a circular rule that causes an infinite loop.

Note

Creating and updating rules are administrator-level actions. Any user who has permission to create or update rules is able to access data processed by the rules.

To create a rule (AWS CLI)

Use the [create-topic-rule](#) command to create a rule:

```
aws iot create-topic-rule --rule-name myrule --topic-rule-payload file://myrule.json
```

The following is an example payload file with a rule that inserts all messages sent to the `iot/test` topic into the specified DynamoDB table. The SQL statement filters the messages and the role ARN grants AWS IoT permission to write to the DynamoDB table.

```
{
  "sql": "SELECT * FROM 'iot/test'",
  "ruleDisabled": false,
  "awsIotSqlVersion": "2016-03-23",
  "actions": [
    {
      "dynamodb": {
        "tableName": "my-dynamodb-table",
        "roleArn": "arn:aws:iam::123456789012:role/my-iot-role",
        "hashKeyField": "topic",
        "hashKeyValue": "${topic(2)}",
        "rangeKeyField": "timestamp",
        "rangeKeyValue": "${timestamp()}"
      }
    }
  ]
}
```

The following is an example payload file with a rule that inserts all messages sent to the `iot/test` topic into the specified S3 bucket. The SQL statement filters the messages, and the role ARN grants AWS IoT permission to write to the Amazon S3 bucket.

```
{
  "awsIotSqlVersion": "2016-03-23",
  "sql": "SELECT * FROM 'iot/test'",
  "ruleDisabled": false,
  "actions": [
    {
      "s3": {
        "roleArn": "arn:aws:iam::123456789012:role/aws_iot_s3",
        "bucketName": "my-bucket",
        "key": "myS3Key"
      }
    }
  ]
}
```

The following is an example payload file with a rule that pushes data to Amazon OpenSearch Service:

```
{
    "sql": "SELECT *, timestamp() as timestamp FROM 'iot/test'",
    "ruleDisabled": false,
    "awsIotSqlVersion": "2016-03-23",
    "actions": [
        {
            "elasticsearch": {
                "roleArn": "arn:aws:iam::123456789012:role/aws_iot_es",
                "endpoint": "https://my-endpoint",
                "index": "my-index",
                "type": "my-type",
                "id": "${newuuid()}"
            }
        }
    ]
}
```

The following is an example payload file with a rule that invokes a Lambda function:

```
{
    "sql": "expression",
    "ruleDisabled": false,
    "awsIotSqlVersion": "2016-03-23",
    "actions": [
        {
            "lambda": {
                "functionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-lambda-function"
            }
        }
    ]
}
```

The following is an example payload file with a rule that publishes to an Amazon SNS topic:

```
{
    "sql": "expression",
    "ruleDisabled": false,
    "awsIotSqlVersion": "2016-03-23",
    "actions": [
        {
            "sns": {
                "targetArn": "arn:aws:sns:us-west-2:123456789012:my-sns-topic",
                "roleArn": "arn:aws:iam::123456789012:role/my-iot-role"
            }
        }
    ]
}
```

The following is an example payload file with a rule that republishes on a different MQTT topic:

```
{
    "sql": "expression",
    "ruleDisabled": false,
    "awsIotSqlVersion": "2016-03-23",
    "actions": [
        {
            "republish": {
                "topic": "my-mqtt-topic",
                "roleArn": "arn:aws:iam::123456789012:role/my-iot-role"
            }
        }
    ]
}
```

The following is an example payload file with a rule that pushes data to an Amazon Kinesis Data Firehose stream:

```
{  
    "sql": "SELECT * FROM 'my-topic'",  
    "ruleDisabled": false,  
    "awsIotSqlVersion": "2016-03-23",  
    "actions": [  
        {"  
            "firehose": {  
                "roleArn": "arn:aws:iam::123456789012:role/my-iot-role",  
                "deliveryStreamName": "my-stream-name"  
            }  
        }  
    ]  
}
```

The following is an example payload file with a rule that uses the Amazon Machine Learning `machinelearning_predict` function to republish to a topic if the data in the MQTT payload is classified as a 1.

```
{  
    "sql": "SELECT * FROM 'iot/test' where machinelearning_predict('my-model',  
    'arn:aws:iam::123456789012:role/my-iot-aml-role', *).predictedLabel=1",  
    "ruleDisabled": false,  
    "awsIotSqlVersion": "2016-03-23",  
    "actions": [  
        {"  
            "republish": {  
                "roleArn": "arn:aws:iam::123456789012:role/my-iot-role",  
                "topic": "my-mqtt-topic"  
            }  
        }  
    ]  
}
```

The following is an example payload file with a rule that publishes messages to a Salesforce IoT Cloud input stream.

```
{  
    "sql": "expression",  
    "ruleDisabled": false,  
    "awsIotSqlVersion": "2016-03-23",  
    "actions": [  
        {"  
            "salesforce": {  
                "token": "ABCDEFGHI123456789abcdefghi123456789",  
                "url": "https://ingestion-cluster-id.my-env.sfdcnow.com/streams/stream-id/  
connection-id/my-event"  
            }  
        }  
    ]  
}
```

The following is an example payload file with a rule that starts an execution of a Step Functions state machine.

```
{  
    "sql": "expression",  
    "ruleDisabled": false,  
    "awsIotSqlVersion": "2016-03-23",  
    "actions": [  
        {"  
            "stepFunctions": {  
                "stateMachineName": "myCoolStateMachine",  
                "executionNamePrefix": "coolRunning",  
                "roleArn": "arn:aws:iam::123456789012:role/my-iot-role"  
            }  
        }  
    ]  
}
```

Viewing your rules

Use the [list-topic-rules](#) command to list your rules:

```
aws iot list-topic-rules
```

Use the [get-topic-rule](#) command to get information about a rule:

```
aws iot get-topic-rule --rule-name myrule
```

Deleting a rule

When you are finished with a rule, you can delete it.

To delete a rule (AWS CLI)

Use the [delete-topic-rule](#) command to delete a rule:

```
aws iot delete-topic-rule --rule-name myrule
```

AWS IoT rule actions

AWS IoT rule actions specify what to do when a rule is triggered. You can define actions to send data to an Amazon DynamoDB database, send data to Amazon Kinesis Data Streams, invoke an AWS Lambda function, and so on. AWS IoT supports the following actions in AWS Regions where the action's service is available.

Rule action	Description	Name in API
Apache Kafka (p. 452)	Sends a message to an Apache Kafka cluster.	kafka
CloudWatch alarms (p. 459)	Changes the state of an Amazon CloudWatch alarm.	cloudwatchAlarm
CloudWatch Logs (p. 460)	Sends a message to Amazon CloudWatch Logs.	cloudwatchLogs
CloudWatch metrics (p. 461)	Sends a message to a CloudWatch metric.	cloudwatchMetric
DynamoDB (p. 463)	Sends a message to a DynamoDB table.	dynamoDB
DynamoDBv2 (p. 465)	Sends message data to multiple columns in a DynamoDB table.	dynamoDBv2
Elasticsearch (p. 466)	Sends a message to an Elasticsearch endpoint.	elasticsearch

Rule action	Description	Name in API
HTTP (p. 468)	Posts a message to an HTTPS endpoint.	http
IoT Analytics (p. 495)	Sends a message to an AWS IoT Analytics channel.	iotAnalytics
IoT Events (p. 496)	Sends a message to an AWS IoT Events input.	iotEvents
IoT SiteWise (p. 498)	Sends message data to AWS IoT SiteWise asset properties.	iotSiteWise
Kinesis Data Firehose (p. 502)	Sends a message to a Kinesis Data Firehose delivery stream.	firehose
Kinesis Data Streams (p. 503)	Sends a message to a Kinesis data stream.	kinesis
Lambda (p. 505)	Invokes a Lambda function with message data as input.	lambda
OpenSearch (p. 507)	Sends a message to an Amazon OpenSearch Service endpoint.	OpenSearch
Republish (p. 509)	Republishes a message to another MQTT topic.	republish
S3 (p. 510)	Stores a message in an Amazon Simple Storage Service (Amazon S3) bucket.	s3
Salesforce IoT (p. 511)	Sends a message to a Salesforce IoT input stream.	salesforce
SNS (p. 512)	Publishes a message as an Amazon Simple Notification Service (Amazon SNS) push notification.	sns
SQS (p. 514)	Sends a message to an Amazon Simple Queue Service (Amazon SQS) queue.	sqrs
Step Functions (p. 515)	Starts an AWS Step Functions state machine.	stepFunctions
the section called "Timestream" (p. 516)	Sends a message to an Amazon Timestream database table.	timestream

Notes

- You must define the rule in the same AWS Region as another service's resource, so that the rule action can interact with that resource.
- The AWS IoT rules engine might make multiple attempts to perform an action in case of intermittent errors. If all attempts fail, the message is discarded and the error is available in your CloudWatch logs. You can specify an error action for each rule that is invoked after a failure occurs. For more information, see [Error handling \(error action\) \(p. 526\)](#).

- Some rule actions trigger actions in services that integrate with AWS Key Management Service (AWS KMS) to support data encryption at rest. If you use a customer-managed AWS KMS key (KMS key) to encrypt data at rest, the service must have permission to use the KMS key on the caller's behalf. See the data encryption topics in the appropriate service guide to learn how to manage permissions for your customer-managed KMS key. For more information about customer-managed KMS keys, see [AWS Key Management Service concepts](#) in the [AWS Key Management Service Developer Guide](#).

Apache Kafka

The Apache Kafka (Kafka) action sends messages directly to your Amazon Managed Streaming for Apache Kafka (Amazon MSK) or self-managed Apache Kafka clusters for data analysis and visualization.

Note

This topic assumes familiarity with the Apache Kafka platform and related concepts. For more information about Apache Kafka, see [Apache Kafka](#).

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `ec2:CreateNetworkInterface`, `ec2:DescribeNetworkInterfaces`, `ec2:CreateNetworkInterfacePermission`, `ec2:DeleteNetworkInterface`, `ec2:DescribeSubnets`, `ec2:DescribeVpcs`, `ec2:DescribeVpcAttribute`, and `ec2:DescribeSecurityGroups` operations. This role creates and manages elastic network interfaces to your Amazon Virtual Private Cloud to reach your Kafka broker. For more information, see [Granting AWS IoT the required access \(p. 444\)](#).

In the AWS IoT console, you can choose or create a role to allow AWS IoT Core to perform this rule action.

For more information about network interfaces, see [Elastic network interfaces](#) in the Amazon EC2 User Guide.

The policy attached to the role you specify should look like the following example.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "ec2:CreateNetworkInterface",  
                "ec2:DescribeNetworkInterfaces",  
                "ec2:CreateNetworkInterfacePermission",  
                "ec2:DeleteNetworkInterface",  
                "ec2:DescribeSubnets",  
                "ec2:DescribeVpcs",  
                "ec2:DescribeVpcAttribute",  
                "ec2:DescribeSecurityGroups"  
            ],  
            "Resource": "*"  
        }  
    ]  
}
```

- If you use AWS Secrets Manager to store the credentials required to connect to your Kafka broker, you must create an IAM role that AWS IoT Core can assume to perform the `secretsmanager:GetSecretValue` and `secretsmanager:DescribeSecret` operations.

The policy attached to the role you specify should look like the following example.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "secretsmanager:GetSecretValue",  
                "secretsmanager:DescribeSecret"  
            ],  
            "Resource": [  
  
                "arn:aws:secretsmanager:region:123456789012:secret:kafka_client_truststore-*",  
                "arn:aws:secretsmanager:region:123456789012:secret:kafka_keytab-*"  
            ]  
        }  
    ]  
}
```

- You must create a virtual private cloud (VPC) destination. (You can run your Apache Kafka clusters inside Amazon Virtual Private Cloud.) The AWS IoT rules engine creates a network interface in each of the subnets listed in the VPC destination. This allows the rules engine to route traffic directly to the VPC. When you create a VPC destination, the AWS IoT rules engine automatically creates a VPC rule action. For more information about VPC rule actions, see [Virtual private cloud \(VPC\) destinations \(p. 457\)](#).
- If you use a customer-managed AWS KMS key (KMS key) to encrypt data at rest, the service must have permission to use the KMS key on the caller's behalf. For more information, see [Amazon MSK encryption](#) in the *Amazon Managed Streaming for Apache Kafka Developer Guide*.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

destinationArn

The Amazon Resource Name (ARN) of the VPC destination. For information about creating a VPC destination, see [Virtual private cloud \(VPC\) destinations \(p. 457\)](#).

topic

The Kafka topic for messages to be sent to the Kafka broker.

You can substitute this field using a substitution template. For more information, see [the section called “Substitution templates” \(p. 586\)](#).

key (optional)

The Kafka message key.

You can substitute this field using a substitution template. For more information, see [the section called “Substitution templates” \(p. 586\)](#).

partition (optional)

The Kafka message partition.

You can substitute this field using a substitution template. For more information, see [the section called "Substitution templates" \(p. 586\)](#).

clientProperties

An object that defines the properties of the Apache Kafka producer client.

acks (optional)

The number of acknowledgments the producer requires the server to have received before considering a request complete.

If you specify 0 as the value, the producer will not wait for any acknowledgment from the server. If the server doesn't receive the message, the producer won't retry to send the message.

Valid values: 0, 1. The default value is 1.

bootstrap.servers

A list of host and port pairs (`host1:port1, host2:port2, etc.`) used to establish the initial connection to your Kafka cluster.

compression.type (optional)

The compression type for all data generated by the producer.

Valid values: `none, gzip, snappy, lz4, zstd`. The default value is `none`.

security.protocol

The security protocol used to attach to your Kafka broker.

Valid values: `SSL, SASL_SSL`. The default value is `SSL`.

key.serializer

Specifies how to turn the key objects you provide with the `ProducerRecord` into bytes.

Valid value: `StringSerializer`.

value.serializer

Specifies how to turn value objects you provide with the `ProducerRecord` into bytes.

Valid value: `ByteBufferSerializer`.

ssl.truststore

The truststore file in base64 format or the location of the truststore file in [AWS Secrets Manager](#). This value isn't required if your truststore is trusted by Amazon certificate authorities (CA).

This field supports substitution templates. If you use Secrets Manager to store the credentials required to connect to your Kafka broker, you can use the `get_secret` SQL function to retrieve the value for this field. For more information about substitution templates, see [the section called "Substitution templates" \(p. 586\)](#). For more information about the `get_secret` SQL function, see [the section called "get_secret\(secretId, secretType, key, roleArn\)" \(p. 556\)](#). If the truststore is in the form of a file, use the `SecretBinary` parameter. If the truststore is in the form of a string, use the `SecretString` parameter.

The maximum size of this value is 65 KB.

ssl.truststore.password

The password for the truststore. This value is required only if you've created a password for the truststore.

ssl.keystore

The keystore file. This value is required when you specify `SSL` as the value for `security.protocol`.

This field supports substitution templates. You must use Secrets Manager to store the credentials required to connect to your Kafka broker. Use the `get_secret` SQL function to retrieve the value for this field. For more information about substitution templates, see [the section called "Substitution templates" \(p. 586\)](#). For more information about the `get_secret` SQL function, see [the section called "get_secret\(secretId, secretType, key, roleArn\)" \(p. 556\)](#). Use the `SecretBinary` parameter.

ssl.keystore.password

The store password for the keystore file. This value is required if you specify a value for `ssl.keystore`.

The value of this field can be plain text. This field also supports substitution templates. You must use Secrets Manager to store the credentials required to connect to your Kafka broker. Use the `get_secret` SQL function to retrieve the value for this field. For more information about substitution templates, see [the section called "Substitution templates" \(p. 586\)](#). For more information about the `get_secret` SQL function, see [the section called "get_secret\(secretId, secretType, key, roleArn\)" \(p. 556\)](#). Use the `SecretString` parameter.

ssl.key.password

The password of the private key in your keystore file.

This field supports substitution templates. You must use Secrets Manager to store the credentials required to connect to your Kafka broker. Use the `get_secret` SQL function to retrieve the value for this field. For more information about substitution templates, see [the section called "Substitution templates" \(p. 586\)](#). For more information about the `get_secret` SQL function, see [the section called "get_secret\(secretId, secretType, key, roleArn\)" \(p. 556\)](#). Use the `SecretString` parameter.

sasl.mechanism

The security mechanism used to connect to your Kafka broker. This value is required when you specify `SASL_SSL` for `security.protocol`.

Valid values: `PLAIN`, `SCRAM-SHA-512`, `GSSAPI`.

Note

`SCRAM-SHA-512` is the only supported security mechanism in the `cn-north-1`, `cn-northwest-1`, `us-gov-east-1`, and `us-gov-west-1` Regions.

sasl.plain.username

The user name used to retrieve the secret string from Secrets Manager. This value is required when you specify `SASL_SSL` for `security.protocol` and `PLAIN` for `sasl.mechanism`.

sasl.plain.password

The password used to retrieve the secret string from Secrets Manager. This value is required when you specify `SASL_SSL` for `security.protocol` and `PLAIN` for `sasl.mechanism`.

sasl.scram.username

The user name used to retrieve the secret string from Secrets Manager. This value is required when you specify `SASL_SSL` for `security.protocol` and `SCRAM-SHA-512` for `sasl.mechanism`.

sasl.scram.password

The password used to retrieve the secret string from Secrets Manager. This value is required when you specify `SASL_SSL` for `security.protocol` and `SCRAM-SHA-512` for `sasl.mechanism`.

sasl.kerberos.keytab

The keytab file for Kerberos authentication in Secrets Manager. This value is required when you specify `SASL_SSL` for `security.protocol` and `GSSAPI` for `sasl.mechanism`.

This field supports substitution templates. You must use Secrets Manager to store the credentials required to connect to your Kafka broker. Use the `get_secret` SQL function to retrieve the value for this field. For more information about substitution templates, see [the section called "Substitution templates" \(p. 586\)](#). For more information about the `get_secret` SQL function, see [the section called "get_secret\(secretId, secretType, key, roleArn\)" \(p. 556\)](#). Use the `SecretBinary` parameter.

sasl.kerberos.service.name

The Kerberos principal name under which Apache Kafka runs. This value is required when you specify `SASL_SSL` for `security.protocol` and `GSSAPI` for `sasl.mechanism`.

sasl.kerberos.krb5.kdc

The hostname of the key distribution center (KDC) to which your Apache Kafka producer client connects. This value is required when you specify `SASL_SSL` for `security.protocol` and `GSSAPI` for `sasl.mechanism`.

sasl.kerberos.krb5.realm

The realm to which your Apache Kafka producer client connects. This value is required when you specify `SASL_SSL` for `security.protocol` and `GSSAPI` for `sasl.mechanism`.

sasl.kerberos.principal

The unique Kerberos identity to which Kerberos can assign tickets to access Kerberos-aware services. This value is required when you specify `SASL_SSL` for `security.protocol` and `GSSAPI` for `sasl.mechanism`.

Examples

The following JSON example defines an Apache Kafka action in an AWS IoT rule.

```
{
    "topicRulePayload": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "kafka": {
                    "destinationArn": "arn:aws:iot:region:123456789012:ruledestination/vpc/VPCDestinationARN",
                    "topic": "TopicName",
                    "clientProperties": {
                        "bootstrap.servers": "kafka.com:9092",
                        "security.protocol": "SASL_SSL",
                        "ssl.truststore": "${get_secret('kafka_client_truststore', 'SecretBinary', 'arn:aws:iam::123456789012:role/kafka-get-secret-role-name')}",
                        "ssl.truststore.password": "kafka password",
                        "sasl.mechanism": "GSSAPI",
                        "sasl.kerberos.service.name": "kafka",
                        "sasl.kerberos.krb5.kdc": "kerberosdns.com",
                        "sasl.kerberos.keytab": "${get_secret('kafka_keytab', 'SecretBinary', 'arn:aws:iam::123456789012:role/kafka-get-secret-role-name')}",
                        "sasl.kerberos.krb5.realm": "KERBEROSREALM",
                        "sasl.kerberos.principal": "kafka-keytab/kafka-keytab.com"
                    }
                }
            }
        ]
    }
}
```

```
        }
    }
]
```

Important notes about your Kerberos setup

- Your key distribution center (KDC) must be resolvable through private Domain Name System (DNS) within your target VPC. One possible approach is to add the KDC DNS entry to a private hosted zone. For more information about this approach, see [Working with private hosted zones](#).
- Each VPC must have DNS resolution enabled. For more information, see [Using DNS with your VPC](#).
- Network interface security groups and instance-level security groups in the VPC destination must allow traffic from within your VPC on the following ports.
 - TCP traffic on the bootstrap broker listener port (often 9092, but must be within the 9000 - 9100 range)
 - TCP and UDP traffic on port 88 for the KDC
- SCRAM-SHA-512 is the only supported security mechanism in the cn-north-1, cn-northwest-1, us-gov-east-1, and us-gov-west-1 Regions.

Virtual private cloud (VPC) destinations

The Apache Kafka rule action routes data to an Apache Kafka cluster in an Amazon Virtual Private Cloud (Amazon VPC). The VPC configuration used by the Apache Kafka rule action is automatically enabled when you specify the VPC destination for your rule action.

Note

If a VPC topic rule destination doesn't receive any traffic for 30 days in a row, it will be disabled. If any resources used by the VPC destination change, the destination will be disabled and unable to be used.

Some changes that can disable a VPC destination include: deleting the VPC, subnets, security groups, or the role used; modifying the role to no longer have the necessary permissions; and disabling the destination.

A VPC destination contains a list of subnets inside the VPC. The rules engine creates an elastic network interface in each subnet that you specify in this list. For more information about network interfaces, see [Elastic network interfaces](#) in the Amazon EC2 User Guide.

For pricing purposes, a VPC rule action is metered in addition to the action that sends a message to a resource when the resource is in your VPC. For pricing information, see [AWS IoT Core pricing](#).

Creating virtual private cloud (VPC) topic rule destinations

You create a virtual private cloud (VPC) destination by using the [CreateTopicRuleDestination](#) API or the AWS IoT Core console.

When you create a VPC destination, you must specify the following information.

`vpclId`

The unique ID of the VPC destination.

`subnetIds`

A list of subnets in which the rules engine creates elastic network interfaces. The rules engine allocates a single network interface for each subnet in the list.

securityGroups (optional)

A list of security groups to apply to the network interfaces.

roleArn

The Amazon Resource Name (ARN) of a role that has permission to create network interfaces on your behalf.

This ARN should have a policy attached to it that looks like the following example.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "ec2:CreateNetworkInterface",  
                "ec2:DescribeNetworkInterfaces",  
                "ec2:CreateNetworkInterfacePermission",  
                "ec2:DeleteNetworkInterface",  
                "ec2:DescribeSubnets",  
                "ec2:DescribeVpcs",  
                "ec2:DescribeVpcAttribute"  
            ],  
            "Resource": "*"  
        }  
    ]  
}
```

Creating a VPC destination by using AWS CLI

The following example shows how to create a VPC destination by using AWS CLI.

```
aws --region regions iot create-topic-rule-destination --destination-configuration  
    'vpcConfiguration={subnetIds=["subnet-123456789101230456"],securityGroups=[],vpcId="vpc-  
    123456789101230456",roleArn="arn:aws:iam::123456789012:role/role-name"}'
```

After you run this command, the VPC destination status will be `IN_PROGRESS`. After a few minutes, its status will change to either `ERROR` (if the command isn't successful) or `ENABLED`. When the destination status is `ENABLED`, it's ready to use.

You can use the following command to get the status of your VPC destination.

```
aws --region region iot get-topic-rule-destination --arn "VPCDestinationARN"
```

Creating a VPC destination by using the AWS IoT Core console

The following steps describe how to create a VPC destination by using the AWS IoT Core console.

1. Navigate to the AWS IoT Core console. In the left pane, on the **Act** tab, choose **Destinations**.
2. Enter values for the following fields.

- **VPC ID**

- **Subnet IDs**
 - **Security Group**
3. Select a role that has the permissions required to create network interfaces. The preceding example policy contains these permissions.

When the VPC destination status is **ENABLED**, it's ready to use.

CloudWatch alarms

The CloudWatch alarm (`cloudWatchAlarm`) action changes the state of an Amazon CloudWatch alarm. You can specify the state change reason and value in this call.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `cloudwatch:SetAlarmState` operation. For more information, see [Granting AWS IoT the required access \(p. 444\)](#).

In the AWS IoT console, you can choose or create a role to allow AWS IoT to perform this rule action.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`alarmName`

The CloudWatch alarm name.

Supports [substitution templates \(p. 586\)](#): API and AWS CLI only

`stateReason`

Reason for the alarm change.

Supports [substitution templates \(p. 586\)](#): Yes

`stateValue`

The value of the alarm state. Valid values: OK, ALARM, INSUFFICIENT_DATA.

Supports [substitution templates \(p. 586\)](#): Yes

`roleArn`

The IAM role that allows access to the CloudWatch alarm. For more information, see [Requirements \(p. 459\)](#).

Supports [substitution templates \(p. 586\)](#): No

Examples

The following JSON example defines a CloudWatch alarm action in an AWS IoT rule.

```
{  
    "topicRulePayload": {  
        "sql": "SELECT * FROM 'some/topic'",  
        "ruleDisabled": false,  
        "awsIotSqlVersion": "2016-03-23",  
    }  
}
```

```
"actions": [
    {
        "cloudwatchAlarm": {
            "alarmName": "IoTAlarm",
            "stateReason": "Temperature stabilized.",
            "stateValue": "OK",
            "roleArn": "arn:aws:iam::123456789012:role/aws_iot_cw"
        }
    }
}
```

See also

- [What is Amazon CloudWatch?](#) in the *Amazon CloudWatch User Guide*
- [Using Amazon CloudWatch alarms](#) in the *Amazon CloudWatch User Guide*

CloudWatch Logs

The CloudWatch Logs (`cloudwatchLogs`) action sends data to Amazon CloudWatch Logs. You can specify the log group to which the action sends data.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `logs:CreateLogStream`, `logs:DescribeLogStreams`, and `logs:PutLogEvents` operations. For more information, see [Granting AWS IoT the required access \(p. 444\)](#).

In the AWS IoT console, you can choose or create a role to allow AWS IoT to perform this rule action.
- If you use a customer-managed AWS KMS key (KMS key) to encrypt log data in CloudWatch Logs, the service must have permission to use the KMS key on the caller's behalf. For more information, see [Encrypt log data in CloudWatch Logs using AWS KMS](#) in the *Amazon CloudWatch Logs User Guide*.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`logGroupName`

The CloudWatch log group to which the action sends data.

Supports [substitution templates \(p. 586\)](#): API and AWS CLI only

`roleArn`

The IAM role that allows access to the CloudWatch log group. For more information, see [Requirements \(p. 460\)](#).

Supports [substitution templates \(p. 586\)](#): No

Examples

The following JSON example defines a CloudWatch Logs action in an AWS IoT rule.

```
{  
    "topicRulePayload": {  
        "sql": "SELECT * FROM 'some/topic'",  
        "ruleDisabled": false,  
        "awsIotSqlVersion": "2016-03-23",  
        "actions": [  
            {  
                "cloudwatchLogs": {  
                    "logGroupName": "IotLogs",  
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_cw"  
                }  
            }  
        ]  
    }  
}
```

See also

- [What is Amazon CloudWatch Logs?](#) in the *Amazon CloudWatch Logs User Guide*

CloudWatch metrics

The CloudWatch metric (`cloudwatchMetric`) action captures an Amazon CloudWatch metric. You can specify the metric namespace, name, value, unit, and timestamp.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `cloudwatch:PutMetricData` operation. For more information, see [Granting AWS IoT the required access \(p. 444\)](#).

In the AWS IoT console, you can choose or create a role to allow AWS IoT to perform this rule action.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`metricName`

The CloudWatch metric name.

Supports [substitution templates \(p. 586\)](#): Yes

`metricNamespace`

The CloudWatch metric namespace name.

Supports [substitution templates \(p. 586\)](#): Yes

`metricUnit`

The metric unit supported by CloudWatch.

Supports [substitution templates \(p. 586\)](#): Yes

`metricValue`

A string that contains the CloudWatch metric value.

Supports [substitution templates \(p. 586\)](#): Yes

`metricTimestamp`

(Optional) A string that contains the timestamp, expressed in seconds in Unix epoch time. Defaults to the current Unix epoch time.

Supports [substitution templates \(p. 586\)](#): Yes

`roleArn`

The IAM role that allows access to the CloudWatch metric. For more information, see [Requirements \(p. 461\)](#).

Supports [substitution templates \(p. 586\)](#): No

Examples

The following JSON example defines a CloudWatch metric action in an AWS IoT rule.

```
{  
    "topicRulePayload": {  
        "sql": "SELECT * FROM 'some/topic'",  
        "ruleDisabled": false,  
        "awsIotSqlVersion": "2016-03-23",  
        "actions": [  
            {  
                "cloudwatchMetric": {  
                    "metricName": "IoMetric",  
                    "metricNamespace": "IoNamespace",  
                    "metricUnit": "Count",  
                    "metricValue": "1",  
                    "metricTimestamp": "1456821314",  
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_cw"  
                }  
            }  
        ]  
    }  
}
```

The following JSON example defines a CloudWatch metric action with substitution templates in an AWS IoT rule.

```
{  
    "topicRulePayload": {  
        "sql": "SELECT * FROM 'some/topic'",  
        "ruleDisabled": false,  
        "awsIotSqlVersion": "2016-03-23",  
        "actions": [  
            {  
                "cloudwatchMetric": {  
                    "metricName": "${topic()}",  
                    "metricNamespace": "${namespace}",  
                    "metricUnit": "${unit}",  
                    "metricValue": "${value}",  
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_cw"  
                }  
            }  
        ]  
    }  
}
```

See also

- [What is Amazon CloudWatch?](#) in the *Amazon CloudWatch User Guide*
- [Using Amazon CloudWatch metrics](#) in the *Amazon CloudWatch User Guide*

DynamoDB

The DynamoDB (`dynamodb`) action writes all or part of an MQTT message to an Amazon DynamoDB table.

You can follow a tutorial that shows you how to create and test a rule with a DynamoDB action. For more information, see [Tutorial: Storing device data in a DynamoDB table \(p. 197\)](#).

Note

This rule writes non-JSON data to DynamoDB as binary data. The DynamoDB console displays the data as base64-encoded text.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `dynamodb:PutItem` operation. For more information, see [Granting AWS IoT the required access \(p. 444\)](#).
In the AWS IoT console, you can choose or create a role to allow AWS IoT to perform this rule action.
- If you use a customer-managed AWS KMS key (KMS key) to encrypt data at rest in DynamoDB, the service must have permission to use the KMS key on the caller's behalf. For more information, see [Customer Managed KMS key](#) in the *Amazon DynamoDB Getting Started Guide*.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`tableName`

The name of the DynamoDB table.

Supports [substitution templates \(p. 586\)](#): API and AWS CLI only

`hashKeyField`

The name of the hash key (also called the partition key).

Supports [substitution templates \(p. 586\)](#): API and AWS CLI only

`hashKeyType`

(Optional) The data type of the hash key (also called the partition key). Valid values: `STRING`, `NUMBER`.

Supports [substitution templates \(p. 586\)](#): API and AWS CLI only

`hashKeyValue`

The value of the hash key. Consider using a substitution template such as `#{topic()}` or `#{timestamp()}`.

Supports [substitution templates \(p. 586\)](#): Yes

rangeKeyField

(Optional) The name of the range key (also called the sort key).

Supports [substitution templates \(p. 586\)](#): API and AWS CLI only

rangeKeyType

(Optional) The data type of the range key (also called the sort key). Valid values: STRING, NUMBER.

Supports [substitution templates \(p. 586\)](#): API and AWS CLI only

rangeKeyValue

(Optional) The value of the range key. Consider using a substitution template such as \${topic()} or \${timestamp()}.

Supports [substitution templates \(p. 586\)](#): Yes

payloadField

(Optional) The name of the column where the payload is written. If you omit this value, the payload is written to the column named payload.

Supports [substitution templates \(p. 586\)](#): Yes

operation

(Optional) The type of operation to be performed. Valid values: INSERT, UPDATE, DELETE.

Supports [substitution templates \(p. 586\)](#): Yes

roleARN

The IAM role that allows access to the DynamoDB table. For more information, see [Requirements \(p. 463\)](#).

Supports [substitution templates \(p. 586\)](#): No

The data written to the DynamoDB table is the result from the SQL statement of the rule.

Examples

The following JSON example defines a DynamoDB action in an AWS IoT rule.

```
{  
    "topicRulePayload": {  
        "sql": "SELECT * AS message FROM 'some/topic'",  
        "ruleDisabled": false,  
        "awsIotSqlVersion": "2016-03-23",  
        "actions": [  
            {  
                "dynamoDB": {  
                    "tableName": "my_ddb_table",  
                    "hashKeyField": "key",  
                    "hashKeyValue": "${topic()}",  
                    "rangeKeyField": "timestamp",  
                    "rangeKeyValue": "${timestamp()}",  
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_dynamoDB"  
                }  
            }  
        ]  
    }  
}
```

See also

- [What is Amazon DynamoDB?](#) in the *Amazon DynamoDB Developer Guide*
- [Getting started with DynamoDB](#) in the *Amazon DynamoDB Developer Guide*
- [Tutorial: Storing device data in a DynamoDB table](#) (p. 197)

DynamoDBv2

The DynamoDBv2 (`dynamodbv2`) action writes all or part of an MQTT message to an Amazon DynamoDB table. Each attribute in the payload is written to a separate column in the DynamoDB database.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `dynamodb:PutItem` operation. For more information, see [Granting AWS IoT the required access](#) (p. 444).

In the AWS IoT console, you can choose or create a role to allow AWS IoT to perform this rule action.

- The MQTT message payload must contain a root-level key that matches the table's primary partition key and a root-level key that matches the table's primary sort key, if one is defined.
- If you use a customer-managed AWS KMS key (KMS key) to encrypt data at rest in DynamoDB, the service must have permission to use the KMS key on the caller's behalf. For more information, see [Customer Managed KMS key](#) in the *Amazon DynamoDB Getting Started Guide*.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`putItem`

An object that specifies the DynamoDB table to which the message data will be written. This object must contain the following information:

`tableName`

The name of the DynamoDB table.

Supports [substitution templates](#) (p. 586): API and AWS CLI only

`roleARN`

The IAM role that allows access to the DynamoDB table. For more information, see [Requirements](#) (p. 465).

Supports [substitution templates](#) (p. 586): No

The data written to the DynamoDB table is the result from the SQL statement of the rule.

Examples

The following JSON example defines a DynamoDBv2 action in an AWS IoT rule.

```
{  
    "topicRulePayload": {  
        "sql": "SELECT * AS message FROM 'some/topic'"  
    }  
}
```

```
"ruleDisabled": false,
"awsIotSqlVersion": "2016-03-23",
"actions": [
    {
        "dynamoDBv2": {
            "putItem": {
                "tableName": "my_ddb_table"
            },
            "roleArn": "arn:aws:iam::123456789012:role/aws_iot_dynamoDBv2",
        }
    }
}
```

The following JSON example defines a DynamoDB action with substitution templates in an AWS IoT rule.

```
{
    "topicRulePayload": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2015-10-08",
        "actions": [
            {
                "dynamoDBv2": {
                    "putItem": {
                        "tableName": "${topic()}"
                    },
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_dynamoDBv2"
                }
            }
        ]
    }
}
```

See also

- [What is Amazon DynamoDB?](#) in the *Amazon DynamoDB Developer Guide*
- [Getting started with DynamoDB](#) in the *Amazon DynamoDB Developer Guide*

Elasticsearch

The Elasticsearch (`elasticsearch`) action writes data from MQTT messages to an Amazon OpenSearch Service domain. You can then use tools like OpenSearch Dashboards to query and visualize data in OpenSearch Service.

Warning

The Elasticsearch action can only be used by existing rule actions. To create a new rule action or to update an existing rule action, use the OpenSearch rule action instead. For more information, see [OpenSearch \(p. 507\)](#).

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `es:ESHttpPut` operation. For more information, see [Granting AWS IoT the required access \(p. 444\)](#).

In the AWS IoT console, you can choose or create a role to allow AWS IoT to perform this rule action.

- If you use a customer-managed AWS KMS key (KMS key) to encrypt data at rest in OpenSearch, the service must have permission to use the KMS key on the caller's behalf. For more information, see [Encryption of data at rest for Amazon OpenSearch Service](#) in the *Amazon OpenSearch Service Developer Guide*.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

endpoint

The endpoint of your service domain.

Supports [substitution templates \(p. 586\)](#): API and AWS CLI only

index

The index where you want to store your data.

Supports [substitution templates \(p. 586\)](#): Yes

type

The type of document you are storing.

Supports [substitution templates \(p. 586\)](#): Yes

id

The unique identifier for each document.

Supports [substitution templates \(p. 586\)](#): Yes

roleARN

The IAM role that allows access to the OpenSearch Service domain. For more information, see [Requirements \(p. 466\)](#).

Supports [substitution templates \(p. 586\)](#): No

Examples

The following JSON example defines an Elasticsearch action in an AWS IoT rule and how you can specify the fields for the `elasticsearch` action. For more information, see [ElasticsearchAction](#).

```
{  
    "topicRulePayload": {  
        "sql": "SELECT *, timestamp() as timestamp FROM 'iot/test'",  
        "ruleDisabled": false,  
        "awsIotSqlVersion": "2016-03-23",  
        "actions": [  
            {  
                "elasticsearch": {  
                    "endpoint": "https://my-endpoint",  
                    "index": "my-index",  
                    "type": "my-type",  
                    "id": "${newuuid()}",  
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_es"  
                }  
            }  
        ]  
    }  
}
```

The following JSON example defines an Elasticsearch action with substitution templates in an AWS IoT rule.

```
{  
    "topicRulePayload": {  
        "sql": "SELECT * FROM 'some/topic'",  
        "ruleDisabled": false,  
        "awsIotSqlVersion": "2016-03-23",  
        "actions": [  
            {  
                "elasticsearch": {  
                    "endpoint": "https://my-endpoint",  
                    "index": "${topic()}",  
                    "type": "${type}",  
                    "id": "${newuuid()}",  
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_es"  
                }  
            }  
        ]  
    }  
}
```

See also

- [OpenSearch \(p. 507\)](#)
- [What is Amazon OpenSearch Service?](#)

HTTP

The HTTPS (`http`) action sends data from an MQTT message to a web application or service.

Requirements

This rule action has the following requirements:

- You must confirm and enable HTTPS endpoints before the rules engine can use them. For more information, see [Working with HTTP topic rule destinations \(p. 470\)](#).

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`url`

The HTTPS endpoint where the message is sent using the HTTP POST method. If you use an IP address in place of a hostname, it must be an IPv4 address. IPv6 addresses are not supported.

Supports [substitution templates \(p. 586\)](#): Yes

`confirmationUrl`

(Optional) If specified, AWS IoT uses the confirmation URL to create a matching topic rule destination. You must enable the topic rule destination before using it in an HTTP action. For more information, see [Working with HTTP topic rule destinations \(p. 470\)](#). If you use substitution templates, you must manually create topic rule destinations before the `http` action can be used. `confirmationUrl` must be a prefix of `url`.

The relationship between `url` and `confirmationUrl` is described by the following:

- If `url` is hardcoded and `confirmationUrl` is not provided, we implicitly treat the `url` field as the `confirmationUrl`. AWS IoT creates a topic rule destination for `url`.
- If `url` and `confirmationUrl` are hardcoded, `url` must begin with `confirmationUrl`. AWS IoT creates a topic rule destination for `confirmationUrl`.
- If `url` contains a substitution template, you must specify `confirmationUrl` and `url` must begin with `confirmationUrl`. If `confirmationUrl` contains substitution templates, you must manually create topic rule destinations before the `http` action can be used. If `confirmationUrl` does not contain substitution templates, AWS IoT creates a topic rule destination for `confirmationUrl`.

Supports [substitution templates \(p. 586\)](#): Yes

headers

(Optional) The list of headers to include in HTTP requests to the endpoint. Each header must contain the following information:

key

The key of the header.

Supports [substitution templates \(p. 586\)](#): No

value

The value of the header.

Supports [substitution templates \(p. 586\)](#): Yes

Note

The default content type is `application/json` when the payload is in JSON format. Otherwise, it is `application/octet-stream`. You can overwrite it by specifying the exact content type in the header with the key `content-type` (case insensitive).

auth

(Optional) The authentication used by the rules engine to connect to the endpoint URL specified in the `url` argument. Currently, Signature Version 4 is the only supported authentication type. For more information, see [HTTP Authorization](#).

Supports [substitution templates \(p. 586\)](#): No

Examples

The following JSON example defines an AWS IoT rule with an HTTP action.

```
{  
    "topicRulePayload": {  
        "sql": "SELECT * FROM 'some/topic'",  
        "ruleDisabled": false,  
        "awsIotSqlVersion": "2016-03-23",  
        "actions": [  
            {  
                "http": {  
                    "url": "https://www.example.com/subpath",  
                    "confirmationUrl": "https://www.example.com",  
                    "headers": [  
                        {  
                            "key": "static_header_key",  
                            "value": "static_header_value"  
                        },  
                        {  
                            "key": "another_header_key",  
                            "value": "another_header_value"  
                        }  
                    ]  
                }  
            }  
        ]  
    }  
}
```

```
        "key": "substitutable_header_key",
        "value": "${value_from_payload}"
    }
}
}
}
```

HTTP action retry logic

The AWS IoT rules engine retries the HTTP action according to these rules:

- The rules engine tries to send a message at least once.
- The rules engine retries at most twice. The maximum number of tries is three.
- The rules engine does not attempt a retry if:
 - The previous try provided a response larger than 16384 bytes.
 - The downstream web service or application closes the TCP connection after the try.
 - The total time to complete a request with retries exceeded the request timeout limit.
 - The request returns an HTTP status code other than 429, 500-599.

Note

Standard data transfer costs apply to retries.

See also

- [Working with HTTP topic rule destinations \(p. 470\)](#)
- [Route data directly from AWS IoT Core to your web services in the *Internet of Things on AWS* blog](#)

Working with HTTP topic rule destinations

An HTTP topic rule destination is a web service to which the rules engine can route data from a topic rule. An AWS IoT Core resource describes the web service for AWS IoT. Topic rule destination resources can be shared by different rules.

Before AWS IoT Core can send data to another web service, it must confirm that it can access the service's endpoint.

HTTP topic rule destination overview

An HTTP topic rule destination refers to a web service that supports a confirmation URL and one or more data collection URLs. The HTTP topic rule destination resource contains the confirmation URL of your web service. When you configure an HTTP topic rule action, you specify the actual URL of the endpoint that should receive the data along with the web service's confirmation URL. After your destination has been confirmed, the topic rule sends the result of the SQL statement to the HTTPS endpoint (and not to the confirmation URL).

An HTTP topic rule destination can be in one of the following states:

ENABLED

The destination has been confirmed and can be used by a rule action. A destination must be in the ENABLED state for it to be used in a rule. You can only enable a destination that's in DISABLED status.

DISABLED

The destination has been confirmed but it can't be used by a rule action. This is useful if you want to temporarily prevent traffic to your endpoint without having to go through the confirmation process again. You can only disable a destination that's in ENABLED status.

IN_PROGRESS

Confirmation of the destination is in progress.

ERROR

Destination confirmation timed out.

After an HTTP topic rule destination has been confirmed and enabled, it can be used with any rule in your account.

The following sections describe common actions on HTTP topic rule destinations.

Creating and confirming HTTP topic rule destinations

You create an HTTP topic rule destination by calling the `CreateTopicRuleDestination` operation or by using the AWS IoT console.

After you create a destination, AWS IoT sends a confirmation request to the confirmation URL. The confirmation request has the following format:

```
HTTP POST {confirmationUrl}/?confirmationToken={confirmationToken}
Headers:
x-amz-rules-engine-message-type: DestinationConfirmation
x-amz-rules-engine-destination-arn:"arn:aws:iot:us-east-1:123456789012:ruledestination/
http/7a280e37-b9c6-47a2-a751-0703693f46e4"
Content-Type: application/json
Body:
{
    "arn":"arn:aws:iot:us-east-1:123456789012:ruledestination/http/7a280e37-b9c6-47a2-
a751-0703693f46e4",
    "confirmationToken": "AYADeMXLrPrNY2wqJAKsFNn-...NBJndA",
    "enableUrl": "https://iot.us-east-1.amazonaws.com/confirmdestination/
AYADeMXLrPrNY2wqJAKsFNn-...NBJndA",
    "messageType": "DestinationConfirmation"
}
```

The content of the confirmation request includes the following information:

arn

The Amazon Resource Name (ARN) for the topic rule destination to confirm.

confirmationToken

The confirmation token sent by AWS IoT Core. The token in the example is truncated. Your token will be longer. You'll need this token to confirm your destination with AWS IoT Core.

enableUrl

The URL to which you browse to confirm a topic rule destination.

messageType

The type of message.

To complete the endpoint confirmation process, you must do one of the following after your confirmation URL receives the confirmation request.

- Call the `enableUrl` in the confirmation request, and then call `UpdateTopicRuleDestination` to set the topic rule's status to `ENABLED`.
- Call the `ConfirmTopicRuleDestination` operation and passing the `confirmationToken` from the confirmation request.
- Copy the `confirmationToken` and paste it into the destination's confirmation dialog in the AWS IoT console.

Sending a new confirmation request

To trigger a new confirmation message for a destination, call `UpdateTopicRuleDestination` and set the topic rule destination's status to `IN_PROGRESS`.

You'll need to repeat the confirmation process after you send a new confirmation request.

Disabling and deleting a topic rule destination

To disable a destination, call `UpdateTopicRuleDestination` and set the topic rule destination's status to `DISABLED`. A topic rule in the `DISABLED` state can be enabled again without the need to send a new confirmation request.

To delete a topic rule destination, call `DeleteTopicRuleDestination`.

Certificate authorities supported by HTTPS endpoints in topic rule destinations

The following certificate authorities are supported by HTTPS endpoints in topic rule destinations.

```
Alias name: swisssignplatinumg2ca
Certificate fingerprints:
    MD5: C9:98:27:77:28:1E:3D:0E:15:3C:84:00:B8:85:03:E6
    SHA1: 56:E0:FA:C0:3B:8F:18:23:55:18:E5:D3:11:CA:E8:C2:43:31:AB:66
    SHA256:
3B:22:2E:56:67:11:E9:92:30:0D:C0:B1:5A:B9:47:3D:AF:DE:F8:C8:4D:OC:EF:7D:33:17:B4:C1:82:1D:14:36

Alias name: hellenicacademicandresearchinstitutionsrootca2011
Certificate fingerprints:
    MD5: 73:9F:4C:4B:73:5B:79:E9:FA:BA:1C:EF:6E:CB:D5:C9
    SHA1: FE:45:65:9B:79:03:5B:98:A1:61:B5:51:2E:AC:DA:58:09:48:22:4D
    SHA256:
BC:10:4F:15:A4:8B:E7:09:DC:A5:42:A7:E1:D4:B9:DF:6F:05:45:27:E8:02:EA:A9:2D:59:54:44:25:8A:FE:71

Alias name: teliasonerarootcav1
Certificate fingerprints:
    MD5: 37:41:49:1B:18:56:9A:26:F5:AD:C2:66:FB:40:A5:4C
    SHA1: 43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:82:37
    SHA256:
DD:69:36:FE:21:F8:F0:77:C1:23:A1:A5:21:C1:22:24:F7:22:55:B7:3E:03:A7:26:06:93:E8:A2:4B:0F:A3:89

Alias name: geotrustprimarycertificationauthority
Certificate fingerprints:
    MD5: 02:26:C3:01:5E:08:30:37:43:A9:D0:7D:CF:37:E6:BF
    SHA1: 32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
    SHA256:
37:D5:10:06:C5:12:EA:AB:62:64:21:F1:EC:8C:92:01:3F:C5:F8:2A:E9:8E:E5:33:EB:46:19:B8:DE:B4:D0:6C

Alias name: trustisfpsrootca
```

```
Certificate fingerprints:  
MD5: 30:C9:E7:1E:6B:E6:14:EB:65:B2:16:69:20:31:67:4D  
SHA1: 3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:C0:04  
SHA256:  
C1:B4:82:99:AB:A5:20:8F:E9:63:0A:CE:55:CA:68:A0:3E:DA:5A:51:9C:88:02:A0:D3:A6:73:BE:8F:8E:55:7D  
  
Alias name: quovadisrootca3g3  
Certificate fingerprints:  
MD5: DF:7D:B9:AD:54:6F:68:A1:DF:89:57:03:97:43:B0:D7  
SHA1: 48:12:BD:92:3C:A8:C4:39:06:E7:30:6D:27:96:E6:A4:CF:22:2E:7D  
SHA256:  
88:EF:81:DE:20:2E:B0:18:45:2E:43:F8:64:72:5C:EA:5F:BD:1F:C2:D9:D2:05:73:07:09:C5:D8:B8:69:0F:46  
  
Alias name: buypassclass2ca  
Certificate fingerprints:  
MD5: 46:A7:D2:FE:45:FB:64:5A:A8:59:90:9B:78:44:9B:29  
SHA1: 49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99  
SHA256:  
9A:11:40:25:19:7C:5B:B9:5D:94:E6:3D:55:CD:43:79:08:47:B6:46:B2:3C:DF:11:AD:A4:A0:0E:FF:15:FB:48  
  
Alias name: secureglobalca  
Certificate fingerprints:  
MD5: CF:F4:27:0D:D4:ED:DC:65:16:49:6D:3D:DA:BF:6E:DE  
SHA1: 3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3A:1B  
SHA256:  
42:00:F5:04:3A:C8:59:0E:BB:52:7D:20:9E:D1:50:30:29:FB:CB:D4:1C:A1:B5:06:EC:27:F1:5A:DE:7D:AC:69  
  
Alias name: chunghwaepkirootca  
Certificate fingerprints:  
MD5: 1B:2E:00:CA:26:06:90:3D:AD:FE:6F:15:68:D3:6B:B3  
SHA1: 67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0  
SHA256:  
C0:A6:F4:DC:63:A2:4B:FD:CF:54:EF:2A:6A:08:2A:0A:72:DE:35:80:3E:2F:F5:FF:52:7A:E5:D8:72:06:DF:D5  
  
Alias name: verisignclass2g2ca  
Certificate fingerprints:  
MD5: 2D:BB:E5:25:D3:D1:65:82:3A:B7:0E:FA:E6:EB:E2:E1  
SHA1: B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95:B6:CC:A0:08:1B:67:EC:9D  
SHA256:  
3A:43:E2:20:FE:7F:3E:A9:65:3D:1E:21:74:2E:AC:2B:75:C2:0F:D8:98:03:05:BC:50:2C:AF:8C:2D:9B:41:A1  
  
Alias name: szafirrootca2  
Certificate fingerprints:  
MD5: 11:64:C1:89:B0:24:B1:8C:B1:07:7E:89:9E:51:9E:99  
SHA1: E2:52:FA:95:3F:ED:DB:24:60:BD:6E:28:F3:9C:CC:CF:5E:B3:3F:DE  
SHA256:  
A1:33:9D:33:28:1A:0B:56:E5:57:D3:D3:2B:1C:E7:F9:36:7E:B0:94:BD:5F:A7:2A:7E:50:04:C8:DE:D7:CA:FE  
  
Alias name: quovadisrootca1g3  
Certificate fingerprints:  
MD5: A4:BC:5B:3F:FE:37:9A:FA:64:F0:E2:FA:05:3D:0B:AB  
SHA1: 1B:8E:EA:57:96:29:1A:C9:39:EA:B8:0A:81:1A:73:73:C0:93:79:67  
SHA256:  
8A:86:6F:D1:B2:76:B5:7E:57:8E:92:1C:65:82:8A:2B:ED:58:E9:F2:F2:88:05:41:34:B7:F1:F4:BF:C9:CC:74  
  
Alias name: utndatacorpsgcc  
Certificate fingerprints:  
MD5: B3:A5:3E:77:21:6D:AC:4A:C0:C9:FB:D5:41:3D:CA:06  
SHA1: 58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:D6:D4  
SHA256:  
85:FB:2F:91:DD:12:27:5A:01:45:B6:36:53:4F:84:02:4A:D6:8B:69:B8:EE:88:68:4F:F7:11:37:58:05:B3:48  
  
Alias name: autoridaddecertificacionfirmaprofesionalcifa62634068  
Certificate fingerprints:  
MD5: 73:3A:74:7A:EC:BB:A3:96:A6:C2:E4:E2:C8:9B:C0:C3  
SHA1: AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:B6:FA
```

```
SHA256:  
04:04:80:28:BF:1F:28:64:D4:8F:9A:D4:D8:32:94:36:6A:82:88:56:55:3F:3B:14:30:3F:90:14:7F:5D:40:EF  
  
Alias name: securesignrootca11  
Certificate fingerprints:  
MD5: B7:52:74:E2:92:B4:80:93:F2:75:E4:CC:D7:F2:EA:26  
SHA1: 3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:11:B3  
SHA256:  
BF:0F:EE:FB:9E:3A:58:1A:D5:F9:E9:DB:75:89:98:57:43:D2:61:08:5C:4D:31:4F:6F:5D:72:59:AA:42:16:12  
  
Alias name: amazon-ca-g4-acm2  
Certificate fingerprints:  
MD5: B2:F1:03:2B:93:64:05:80:B8:A8:17:36:B9:1B:52:3C  
SHA1: A7:E6:45:32:1F:7A:B7:AD:C0:70:EA:73:5F:AB:ED:C3:DA:B4:D0:C8  
SHA256:  
D7:A8:7C:69:95:D0:E2:04:2A:32:70:A7:E2:87:FE:A7:E8:F4:C1:70:62:F7:90:C3:EB:BB:53:F2:AC:39:26:BE  
  
Alias name: isrgrootx1  
Certificate fingerprints:  
MD5: 0C:D2:F9:E0:DA:17:73:E9:ED:86:4D:A5:E3:70:E7:4E  
SHA1: CA:BD:2A:79:A1:07:6A:31:F2:1D:25:36:35:CB:03:9D:43:29:A5:E8  
SHA256:  
96:BC:EC:06:26:49:76:F3:74:60:77:9A:CF:28:C5:A7:CF:E8:A3:C0:AA:E1:1A:8F:FC:EE:05:C0:BD:DF:08:C6  
  
Alias name: amazon-ca-g4-acm1  
Certificate fingerprints:  
MD5: E2:F1:18:19:61:5C:43:E0:D4:A8:5D:0B:FA:7C:89:1B  
SHA1: F2:0D:28:B6:29:C2:2C:5E:84:05:E6:02:4D:97:FE:8F:A0:84:93:A0  
SHA256:  
B0:11:A4:F7:29:6C:74:D8:2B:F5:62:DF:87:D7:28:C7:1F:B5:8C:F4:E6:73:F2:78:FC:DA:F3:FF:83:A6:8C:87  
  
Alias name: etugracertificationauthority  
Certificate fingerprints:  
MD5: B8:A1:03:63:B0:BD:21:71:70:8A:6F:13:3A:BB:79:49  
SHA1: 51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:52:39  
SHA256:  
B0:BF:D5:2B:B0:D7:D9:BD:92:BF:5D:4D:C1:3D:A2:55:C0:2C:54:2F:37:83:65:EA:89:39:11:F5:5E:55:F2:3C  
  
Alias name: geotrustuniversalca2  
Certificate fingerprints:  
MD5: 34:FC:B8:D0:36:DB:9E:14:B3:C2:F2:DB:8F:E4:94:C7  
SHA1: 37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:20:79  
SHA256:  
A0:23:4F:3B:C8:52:7C:A5:62:8E:EC:81:AD:5D:69:89:5D:A5:68:0D:C9:1D:1C:B8:47:7F:33:F8:78:B9:5B:0B  
  
Alias name: digicertglobalrootca  
Certificate fingerprints:  
MD5: 79:E4:A9:84:0D:7D:3A:96:D7:C0:4F:E2:43:4C:89:2E  
SHA1: A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36  
SHA256:  
43:48:A0:E9:44:4C:78:CB:26:5E:05:8D:5E:89:44:B4:D8:4F:96:62:BD:26:DB:25:7F:89:34:A4:43:C7:01:61  
  
Alias name: staatderlandenenvrootca  
Certificate fingerprints:  
MD5: FC:06:AF:7B:E8:1A:F1:9A:B4:E8:D2:70:1F:C0:F5:BA  
SHA1: 76:E2:7E:C1:4F:DB:82:C1:C0:A6:75:B5:05:BE:3D:29:B4:ED:DB:BB  
SHA256:  
4D:24:91:41:4C:FE:95:67:46:EC:4C:EF:A6:CF:6F:72:E2:8A:13:29:43:2F:9D:8A:90:7A:C4:CB:5D:AD:C1:5A  
  
Alias name: utnuserfirstclientauthemailca  
Certificate fingerprints:  
MD5: D7:34:3D:EF:1D:27:09:28:E1:31:02:5B:13:2B:DD:F7  
SHA1: B1:72:B1:A5:6D:95:F9:1F:E5:02:87:E1:4D:37:EA:6A:44:63:76:8A  
SHA256:  
43:F2:57:41:2D:44:0D:62:74:76:97:4F:87:7D:A8:F1:FC:24:44:56:5A:36:7A:E6:0E:DD:C2:7A:41:25:31:AE
```

```
Alias name: actalisauthenticationrootca
Certificate fingerprints:
MD5: 69:C1:0D:4F:07:A3:1B:C3:FE:56:3D:04:BC:11:F6:A6
SHA1: F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC
SHA256:
55:92:60:84:EC:96:3A:64:B9:6E:2A:BE:01:CE:0B:A8:6A:64:FB:FE:BC:C7:AA:B5:AF:C1:55:B3:7F:D7:60:66

Alias name: amazonrootca4
Certificate fingerprints:
MD5: 89:BC:27:D5:EB:17:8D:06:6A:69:D5:FD:89:47:B4:CD
SHA1: F6:10:84:07:D6:F8:BB:67:98:0C:C2:E2:44:C2:EB:AE:1C:EF:63:BE
SHA256:
E3:5D:28:41:9E:D0:20:25:CF:A6:90:38:CD:62:39:62:45:8D:A5:C6:95:FB:DE:A3:C2:2B:0B:FB:25:89:70:92

Alias name: amazonrootca3
Certificate fingerprints:
MD5: A0:D4:EF:0B:F7:B5:D8:49:95:2A:EC:F5:C4:FC:81:87
SHA1: 0D:44:DD:8C:3C:8C:1A:1A:58:75:64:81:E9:0F:2E:2A:FF:B3:D2:6E
SHA256:
18:CE:6C:FE:7B:F1:4E:60:B2:E3:47:B8:DF:E8:68:CB:31:D0:2E:BB:3A:DA:27:15:69:F5:03:43:B4:6D:B3:A4

Alias name: amazonrootca2
Certificate fingerprints:
MD5: C8:E5:8D:CE:A8:42:E2:7A:C0:2A:5C:7C:9E:26:BF:66
SHA1: 5A:8C:EF:45:D7:A6:98:59:76:7A:8C:8B:44:96:B5:78:CF:47:4B:1A
SHA256:
1B:A5:B2:AA:8C:65:40:1A:82:96:01:18:F8:0B:EC:4F:62:30:4D:83:CE:C4:71:3A:19:C3:9C:01:1E:A4:6D:B4

Alias name: amazonrootca1
Certificate fingerprints:
MD5: 43:C6:BF:AE:EC:FE:AD:2F:18:C6:88:68:30:FC:C8:E6
SHA1: 8D:A7:F9:65:EC:5E:FC:37:91:0F:1C:6E:59:FD:C1:CC:6A:6E:DE:16
SHA256:
8E:CD:E6:88:4F:3D:87:B1:12:5B:A3:1A:C3:FC:B1:3D:70:16:DE:7F:57:CC:90:4F:E1:CB:97:C6:AE:98:19:6E

Alias name: affirmtrustpremium
Certificate fingerprints:
MD5: C4:5D:0E:48:B6:AC:28:30:4E:0A:BC:F9:38:16:87:57
SHA1: D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
SHA256:
70:A7:3F:7F:37:6B:60:07:42:48:90:45:34:B1:14:82:D5:BF:0E:69:8E:CC:49:8D:F5:25:77:EB:F2:E9:3B:9A

Alias name: keynectisrootca
Certificate fingerprints:
MD5: CC:4D:AE:FB:30:6B:D8:38:FE:50:EB:86:61:4B:D2:26
SHA1: 9C:61:5C:4D:4D:85:10:3A:53:26:C2:4D:BA:EA:E4:A2:D2:D5:CC:97
SHA256:
42:10:F1:99:49:9A:9A:C3:3C:8D:E0:2B:A6:DB:AA:14:40:8B:DD:8A:6E:32:46:89:C1:92:2D:06:97:15:A3:32

Alias name: equifaxsecureglobalebusinessca1
Certificate fingerprints:
MD5: 51:F0:2A:33:F1:F5:55:39:07:F2:16:7A:47:C7:5D:63
SHA1: 3A:74:CB:7A:47:DB:70:DE:89:1F:24:35:98:64:B8:2D:82:BD:1A:36
SHA256:
86:AB:5A:65:71:D3:32:9A:BC:D2:E4:E6:37:66:8B:A8:9C:73:1E:C2:93:B6:CB:A6:0F:71:63:40:A0:91:CE:AE

Alias name: affirmtrustpremiumca
Certificate fingerprints:
MD5: C4:5D:0E:48:B6:AC:28:30:4E:0A:BC:F9:38:16:87:57
SHA1: D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
SHA256:
70:A7:3F:7F:37:6B:60:07:42:48:90:45:34:B1:14:82:D5:BF:0E:69:8E:CC:49:8D:F5:25:77:EB:F2:E9:3B:9A

Alias name: baltimorecodesigningca
Certificate fingerprints:
MD5: 90:F5:28:49:56:D1:5D:2C:B0:53:D4:4B:EF:6F:90:22
```

```
SHA1: 30:46:D8:C8:88:FF:69:30:C3:4A:FC:CD:49:27:08:7C:60:56:7B:0D
SHA256:
A9:15:45:DB:D2:E1:9C:4C:CD:F9:09:AA:71:90:0D:18:C7:35:1C:89:B3:15:F0:F1:3D:05:C1:3A:8F:FB:46:87

Alias name: gdcatrustauthr5root
Certificate fingerprints:
MD5: 63:CC:D9:3D:34:35:5C:6F:53:A3:E2:08:70:48:1F:B4
SHA1: 0F:36:38:5B:81:1A:25:C3:9B:31:4E:83:CA:E9:34:66:70:CC:74:B4
SHA256:
BF:FF:8F:D0:44:33:48:7D:6A:8A:A6:0C:1A:29:76:7A:9F:C2:BB:B0:5E:42:0F:71:3A:13:B9:92:89:1D:38:93

Alias name: certinomisrootca
Certificate fingerprints:
MD5: 14:0A:FD:8D:A8:28:B5:38:69:DB:56:7E:61:22:03:3F
SHA1: 9D:70:BB:01:A5:A4:A0:18:11:2E:F7:1C:01:B9:32:C5:34:E7:88:A8
SHA256:
2A:99:F5:BC:11:74:B7:3C:BB:1D:62:08:84:E0:1C:34:E5:1C:CB:39:78:DA:12:5F:0E:33:26:88:83:BF:41:58

Alias name: verisignclass3publicprimarycertificationauthorityg5
Certificate fingerprints:
MD5: CB:17:E4:31:67:3E:E2:09:FE:45:57:93:F3:0A:FA:1C
SHA1: 4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
SHA256:
9A:CF:AB:7E:43:C8:D8:80:D0:6B:26:2A:94:DE:EE:E4:B4:65:99:89:C3:D0:CA:F1:9B:AF:64:05:E4:1A:B7:DF

Alias name: verisignclass3publicprimarycertificationauthorityg4
Certificate fingerprints:
MD5: 3A:52:E1:E7:FD:6F:3A:E3:6F:F3:6F:99:1B:F9:22:41
SHA1: 22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
SHA256:
69:DD:D7:EA:90:BB:57:C9:3E:13:5D:C8:5E:A6:FC:D5:48:0B:60:32:39:BD:C4:54:FC:75:8B:2A:26:CF:7F:79

Alias name: verisignclass3publicprimarycertificationauthorityg3
Certificate fingerprints:
MD5: CD:68:B6:A7:C7:C4:CE:75:E0:1D:4F:57:44:61:92:09
SHA1: 13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
SHA256:
EB:04:CF:5E:B1:F3:9A:FA:76:2F:2B:B1:20:F2:96:CB:A5:20:C1:B9:7D:B1:58:95:65:B8:1C:B9:A1:7B:72:44

Alias name: swisssignsilverg2ca
Certificate fingerprints:
MD5: E0:06:A1:C9:7D:CF:C9:FC:0D:CO:56:75:96:D8:62:13
SHA1: 9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
SHA256:
BE:6C:4D:A2:BB:B9:BA:59:B6:F3:93:97:68:37:42:46:C3:C0:05:99:3F:A9:8F:02:0D:1D:ED:BE:D4:8A:81:D5

Alias name: swisssignsilvercag2
Certificate fingerprints:
MD5: E0:06:A1:C9:7D:CF:C9:FC:0D:CO:56:75:96:D8:62:13
SHA1: 9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
SHA256:
BE:6C:4D:A2:BB:B9:BA:59:B6:F3:93:97:68:37:42:46:C3:C0:05:99:3F:A9:8F:02:0D:1D:ED:BE:D4:8A:81:D5

Alias name: atotrustedroot2011
Certificate fingerprints:
MD5: AE:B9:C4:32:4B:AC:7F:5D:66:CC:77:94:BB:2A:77:56
SHA1: 2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:AC:21
SHA256:
F3:56:BE:A2:44:B7:A9:1E:B3:5D:53:CA:9A:D7:86:4A:CE:01:8E:2D:35:D5:F8:F9:6D:DF:68:A6:F4:1A:A4:74

Alias name: comodoecccertificationauthority
Certificate fingerprints:
MD5: 7C:62:FF:74:9D:31:53:5E:68:4A:D5:78:AA:1E:BF:23
SHA1: 9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:C3:11
SHA256:
17:93:92:7A:06:14:54:97:89:AD:CE:2F:8F:34:F7:F0:B6:6D:0F:3A:E3:A3:B8:4D:21:EC:15:DB:BA:4F:AD:C7
```

```
Alias name: securetrustca
Certificate fingerprints:
MD5: DC:32:C3:A7:6D:25:57:C7:68:09:9D:EA:2D:A9:A2:D1
SHA1: 87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
SHA256:
F1:C1:B5:0A:E5:A2:0D:D8:03:0E:C9:F6:BC:24:82:3D:D3:67:B5:25:57:59:B4:E7:1B:61:FC:E9:F7:37:5D:73

Alias name: soneraaclass1ca
Certificate fingerprints:
MD5: 33:B7:84:F5:5F:27:D7:68:27:DE:14:DE:12:2A:ED:6F
SHA1: 07:47:22:01:99:CE:74:B9:7C:BO:3D:79:B2:64:A2:C8:55:E9:33:FF
SHA256:
CD:80:82:84:CF:74:6F:F2:FD:6E:B5:8A:A1:D5:9C:4A:D4:B3:CA:56:FD:C6:27:4A:89:26:A7:83:5F:32:31:3D

Alias name: cadisigrootr2
Certificate fingerprints:
MD5: 26:01:FB:D8:27:A7:17:9A:45:54:38:1A:43:01:3B:03
SHA1: B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:D9:71
SHA256:
E2:3D:4A:03:6D:7B:70:E9:F5:95:B1:42:20:79:D2:B9:1E:DF:BB:1F:B6:51:A0:63:3E:AA:8A:9D:C5:F8:07:03

Alias name: cadisigrootr1
Certificate fingerprints:
MD5: BE:EC:11:93:9A:F5:69:21:BC:D7:C1:C0:67:89:CC:2A
SHA1: 8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:56:51:1A:52:C6
SHA256:
F9:6F:23:F4:C3:E7:9C:07:7A:46:98:8D:5A:F5:90:06:76:A0:F0:39:CB:64:5D:D1:75:49:B2:16:C8:24:40:CE

Alias name: verisignclass3g5ca
Certificate fingerprints:
MD5: CB:17:E4:31:67:3E:E2:09:FE:45:57:93:F3:0A:FA:1C
SHA1: 4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
SHA256:
9A:CF:AB:7E:43:C8:D8:80:D0:6B:26:2A:94:DE:EE:E4:B4:65:99:89:C3:D0:CA:F1:9B:AF:64:05:E4:1A:B7:DF

Alias name: utnuserfirsthardwareca
Certificate fingerprints:
MD5: 4C:56:41:E5:0D:BB:2B:E8:CA:A3:ED:18:08:AD:43:39
SHA1: 04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:F9:D7
SHA256:
6E:A5:47:41:D0:04:66:7E:ED:1B:48:16:63:4A:A3:A7:9E:6E:4B:96:95:0F:82:79:DA:FC:8D:9B:D8:81:21:37

Alias name: addtrustqualifiedca
Certificate fingerprints:
MD5: 27:EC:39:47:CD:DA:5A:AF:E2:9A:01:65:21:A9:4C:BB
SHA1: 4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8B:CF
SHA256:
80:95:21:08:05:DB:4B:BC:35:5E:44:28:D8:FD:6E:C2:CD:E3:AB:5F:B9:7A:99:42:98:8E:B8:F4:DC:D0:60:16

Alias name: verisignclass3g3ca
Certificate fingerprints:
MD5: CD:68:B6:A7:C7:C4:CE:75:E0:1D:4F:57:44:61:92:09
SHA1: 13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
SHA256:
EB:04:CF:5E:B1:F3:9A:FA:76:2F:2B:B1:20:F2:96:CB:A5:20:C1:B9:7D:B1:58:95:65:B8:1C:B9:A1:7B:72:44

Alias name: thawtepersonalfreemailca
Certificate fingerprints:
MD5: 53:4B:1D:17:58:58:1A:30:A1:90:F8:6E:5C:F2:CF:65
SHA1: E6:18:83:AE:84:CA:C1:C1:CD:52:AD:E8:E9:25:2B:45:A6:4F:B7:E2
SHA256:
5B:38:BD:12:9E:83:D5:A0:CA:D2:39:21:08:94:90:D5:0D:4A:AE:37:04:28:F8:DD:FF:FF:FA:4C:15:64:E1:84

Alias name: certplusclass3ppprimaryca
Certificate fingerprints:
```

```
MD5: E1:4B:52:73:D7:1B:DB:93:30:E5:BD:E4:09:6E:BE:FB
SHA1: 21:6B:2A:29:E6:2A:00:CE:82:01:46:D8:24:41:41:B9:25:11:B2:79
SHA256:
CC:C8:94:89:37:1B:AD:11:1C:90:61:9B:EA:24:0A:2E:6D:AD:D9:9F:9F:6E:1D:4D:41:E5:8E:D6:DE:3D:02:85

Alias name: swisssigngoldg2ca
Certificate fingerprints:
MD5: 24:77:D9:A8:91:D1:3B:FA:88:2D:C2:FF:F8:CD:33:93
SHA1: D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
SHA256:
62:DD:0B:E9:B9:F5:0A:16:3E:A0:F8:E7:5C:05:3B:1E:CA:57:EA:55:C8:68:8F:64:7C:68:81:F2:C8:35:7B:95

Alias name: swisssigngoldcag2
Certificate fingerprints:
MD5: 24:77:D9:A8:91:D1:3B:FA:88:2D:C2:FF:F8:CD:33:93
SHA1: D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
SHA256:
62:DD:0B:E9:B9:F5:0A:16:3E:A0:F8:E7:5C:05:3B:1E:CA:57:EA:55:C8:68:8F:64:7C:68:81:F2:C8:35:7B:95

Alias name: dtrustrootclass3ca22009
Certificate fingerprints:
MD5: CD:E0:25:69:8D:47:AC:9C:89:35:90:F7:FD:51:3D:2F
SHA1: 58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B:6D:29:D3:FF:8D:5F:00:F0
SHA256:
49:E7:A4:42:AC:F0:EA:62:87:05:00:54:B5:25:64:B6:50:E4:F4:9E:42:E3:48:D6:AA:38:E0:39:E9:57:B1:C1

Alias name: acraizfnmtrcm
Certificate fingerprints:
MD5: E2:09:04:B4:D3:BD:D1:A0:14:FD:1A:D2:47:C4:57:1D
SHA1: EC:50:35:07:B2:15:C4:95:62:19:E2:A8:9A:5B:42:99:2C:4C:2C:20
SHA256:
EB:C5:57:0C:29:01:8C:4D:67:B1:AA:12:7B:AF:12:F7:03:B4:61:1E:BC:17:B7:DA:B5:57:38:94:17:9B:93:FA

Alias name: securitycommunicationevrootca1
Certificate fingerprints:
MD5: 22:2D:A6:01:EA:7C:0A:F7:F0:6C:56:43:3F:77:76:D3
SHA1: FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
SHA256:
A2:2D:BA:68:1E:97:37:6E:2D:39:7D:72:8A:AE:3A:9B:62:96:B9:FD:BA:60:BC:2E:11:F6:47:F2:C6:75:FB:37

Alias name: starfieldclass2ca
Certificate fingerprints:
MD5: 32:4A:4B:BB:C8:63:69:9B:BE:74:9A:C6:DD:1D:46:24
SHA1: AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
SHA256:
14:65:FA:20:53:97:B8:76:FA:A6:F0:A9:95:8E:55:90:E4:0F:CC:7F:AA:4F:B7:C2:C8:67:75:21:FB:5F:B6:58

Alias name: opentrustrootcag3
Certificate fingerprints:
MD5: 21:37:B4:17:16:92:7B:67:46:70:A9:96:D7:A8:13:24
SHA1: 6E:26:64:F3:56:BF:34:55:BF:D1:93:3F:7C:01:DE:D8:13:DA:8A:A6
SHA256:
B7:C3:62:31:70:6E:81:07:8C:36:7C:B8:96:19:8F:1E:32:08:DD:92:69:49:DD:8F:57:09:A4:10:F7:5B:62:92

Alias name: opentrustrootcag2
Certificate fingerprints:
MD5: 57:24:B6:59:24:6B:AE:C8:FE:1C:0C:20:F2:C0:4E:EB
SHA1: 79:5F:88:60:C5:AB:7C:3D:92:E6:CB:F4:8D:E1:45:CD:11:EF:60:0B
SHA256:
27:99:58:29:FE:6A:75:15:C1:BF:E8:48:F9:C4:76:1D:B1:6C:22:59:29:25:7B:F4:0D:08:94:F2:9E:A8:BA:F2

Alias name: buypassclass2rootca
Certificate fingerprints:
MD5: 46:A7:D2:FE:45:FB:64:5A:A8:59:90:9B:78:44:9B:29
SHA1: 49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
```

```
SHA256:  
9A:11:40:25:19:7C:5B:B9:5D:94:E6:3D:55:CD:43:79:08:47:B6:46:B2:3C:DF:11:AD:A4:A0:0E:FF:15:FB:48

Alias name: opentrustrootcag1
Certificate fingerprints:
MD5: 76:00:CC:81:29:CD:55:5E:88:6A:7A:2E:F7:4D:39:DA
SHA1: 79:91:E8:34:F7:E2:EE:DD:08:95:01:52:E9:55:2D:14:E9:58:D5:7E
SHA256:
56:C7:71:28:D9:8C:18:D9:1B:4C:FD:FF:BC:25:EE:91:03:D4:75:8E:A2:AB:AD:82:6A:90:F3:45:7D:46:0E:B4

Alias name: globalsignr2ca
Certificate fingerprints:
MD5: 94:14:77:7E:3E:5E:FD:8F:30:BD:41:B0:CF:E7:D0:30
SHA1: 75:EO:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
SHA256:
CA:42:DD:41:74:5F:D0:B8:1E:B9:02:36:2C:F9:D8:BF:71:9D:A1:BD:1B:1E:FC:94:6F:5B:4C:99:F4:2C:1B:9E

Alias name: buypassclass3rootca
Certificate fingerprints:
MD5: 3D:3B:18:9E:2C:64:5A:E8:D5:88:CE:0E:F9:37:C2:EC
SHA1: DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
SHA256:
ED:F7:EB:BC:A2:7A:2A:38:4D:38:7B:7D:40:10:C6:66:E2:ED:B4:84:3E:4C:29:B4:AE:1D:5B:93:32:E6:B2:4D

Alias name: ecacc
Certificate fingerprints:
MD5: EB:F5:9D:29:0D:61:F9:42:1F:7C:C2:BA:6D:E3:15:09
SHA1: 28:90:3A:63:5B:52:80:FA:E6:77:4C:0B:6D:A7:D6:BA:A6:4A:F2:E8
SHA256:
88:49:7F:01:60:2F:31:54:24:6A:E2:8C:4D:5A:EF:10:F1:D8:7E:BB:76:62:6F:4A:E0:B7:F9:5B:A7:96:87:99

Alias name: epkirootcertificationauthority
Certificate fingerprints:
MD5: 1B:2E:00:CA:26:06:90:3D:AD:FE:6F:15:68:D3:6B:B3
SHA1: 67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
SHA256:
C0:A6:F4:DC:63:A2:4B:FD:CF:54:EF:2A:6A:08:2A:0A:72:DE:35:80:3E:2F:F5:FF:52:7A:E5:D8:72:06:DF:D5

Alias name: verisignclass1g2ca
Certificate fingerprints:
MD5: DB:23:3D:F9:69:FA:4B:B9:95:80:44:73:5E:7D:41:83
SHA1: 27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47
SHA256:
34:1D:E9:8B:13:92:AB:F7:F4:AB:90:A9:60:CF:25:D4:BD:6E:C6:5B:9A:51:CE:6E:D0:67:D0:0E:C7:CE:9B:7F

Alias name: certigna
Certificate fingerprints:
MD5: AB:57:A6:5B:7D:42:82:19:B5:D8:58:26:28:5E:FD:FF
SHA1: B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:94:97
SHA256:
E3:B6:A2:DB:2E:D7:CE:48:84:2F:7A:C5:32:41:C7:B7:1D:54:14:4B:FB:40:C1:1F:3F:1D:0B:42:F5:EE:A1:2D

Alias name: camerfirmaglobalchambersignroot
Certificate fingerprints:
MD5: C5:E6:7B:BF:06:D0:4F:43:ED:C4:7A:65:8A:FB:6B:19
SHA1: 33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:D8:E9
SHA256:
EF:3C:B4:17:FC:8E:BF:6F:97:87:6C:9E:4E:CE:39:DE:1E:A5:FE:64:91:41:D1:02:8B:7D:11:C0:B2:29:8C:ED

Alias name: cfcaevroot
Certificate fingerprints:
MD5: 74:E1:B6:ED:26:7A:7A:44:30:33:94:AB:7B:27:81:30
SHA1: E2:B8:29:4B:55:84:AB:6B:58:C2:90:46:6C:AC:3F:B8:39:8F:84:83
SHA256:
5C:C3:D7:8E:4E:1D:5E:45:54:7A:04:E6:87:3E:64:F9:0C:F9:53:6D:1C:CC:2E:F8:00:F3:55:C4:C5:FD:70:FD
```

```
Alias name: soneraaclass2rootca
Certificate fingerprints:
MD5: A3:EC:75:0F:2E:88:DF:FA:48:01:4E:0B:5C:48:6F:FB
SHA1: 37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
SHA256:
79:08:B4:03:14:C1:38:10:0B:51:8D:07:35:80:7F:FB:FC:F8:51:8A:00:95:33:71:05:BA:38:6B:15:3D:D9:27

Alias name: certumtrustednetworkca
Certificate fingerprints:
MD5: D5:E9:81:40:C5:18:69:FC:46:2C:89:75:62:0F:AA:78
SHA1: 07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
SHA256:
5C:58:46:8D:55:F5:8E:49:7E:74:39:82:D2:B5:00:10:B6:D1:65:37:4A:CF:83:A7:D4:A3:2D:B7:68:C4:40:8E

Alias name: securitycommunicationrootca2
Certificate fingerprints:
MD5: 6C:39:7D:A4:0E:55:59:B2:3F:D6:41:B1:12:50:DE:43
SHA1: 5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
SHA256:
51:3B:2C:EC:B8:10:D4:CD:E5:DD:85:39:1A:DF:C6:C2:DD:60:D8:7B:B7:36:D2:B5:21:48:4A:A4:7A:0E:BE:F6

Alias name: globalseccrootcar5
Certificate fingerprints:
MD5: 9F:AD:3B:1C:02:1E:8A:BA:17:74:38:81:0C:A2:BC:08
SHA1: 1F:24:C6:30:CD:A4:18:EF:20:69:FF:AD:4F:DD:5F:46:3A:1B:69:AA
SHA256:
17:9F:BC:14:8A:3D:D0:0F:D2:4E:A1:34:58:CC:43:BF:A7:F5:9C:81:82:D7:83:A5:13:F6:EB:EC:10:0C:89:24

Alias name: globalseccrootcar4
Certificate fingerprints:
MD5: 20:F0:27:68:D1:7E:A0:9D:0E:E6:2A:CA:DF:5C:89:8E
SHA1: 69:69:56:2E:40:80:F4:24:A1:E7:19:9F:14:BA:F3:EE:58:AB:6A:BB
SHA256:
BE:C9:49:11:C2:95:56:76:DB:6C:0A:55:09:86:D7:6E:3B:A0:05:66:7C:44:2C:97:62:B4:FB:B7:73:DE:22:8C

Alias name: chambersofcommerce2008
Certificate fingerprints:
MD5: 5E:80:9E:84:5A:0E:65:0B:17:02:F3:55:18:2A:3E:D7
SHA1: 78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
SHA256:
06:3E:4A:FA:C4:91:DF:D3:32:F3:08:9B:85:42:E9:46:17:D8:93:D7:FE:94:4E:10:A7:93:7E:E2:9D:96:93:CO

Alias name: pscprocert
Certificate fingerprints:
MD5: E6:24:E9:12:01:AE:0C:DE:8E:85:C4:CE:A3:12:DD:EC
SHA1: 70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:99:B0:3D:50:74
SHA256:
3C:FC:3C:14:D1:F6:84:FF:17:E3:8C:43:CA:44:0C:00:B9:67:EC:93:3E:8B:FE:06:4C:A1:D7:2C:90:F2:AD:BO

Alias name: thawteprimaryrootcag3
Certificate fingerprints:
MD5: FB:1B:5D:43:8A:94:CD:44:C6:76:F2:43:4B:47:E7:31
SHA1: F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
SHA256:
4B:03:F4:58:07:AD:70:F2:1B:FC:2C:AE:71:C9:FD:E4:60:4C:06:4C:F5:FF:B6:86:BA:E5:DB:AA:D7:FD:D3:4C

Alias name: quovadisrootca
Certificate fingerprints:
MD5: 27:DE:36:FE:72:B7:00:03:00:9D:F4:F0:1E:6C:04:24
SHA1: DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
SHA256:
A4:5E:DE:3B:BB:F0:9C:8A:E1:5C:72:EF:C0:72:68:D6:93:A2:1C:99:6F:D5:1E:67:CA:07:94:60:FD:6D:88:73

Alias name: thawteprimaryrootcag2
Certificate fingerprints:
MD5: 74:9D:EA:60:24:C4:FD:22:53:3E:CC:3A:72:D9:29:4F
```

```
SHA1: AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12
SHA256:
A4:31:0D:50:AF:18:A6:44:71:90:37:2A:86:AF:AF:8B:95:1F:FB:43:1D:83:7F:1E:56:88:B4:59:71:ED:15:57

Alias name: deprecateditsecca
Certificate fingerprints:
MD5: A5:96:0C:F6:B5:AB:27:E5:01:C6:00:88:9E:60:33:E5
SHA1: 12:12:0B:03:0E:15:14:54:F4:DD:B3:F5:DE:13:6E:83:5A:29:72:9D
SHA256:
9A:59:DA:86:24:1A:FD:BA:A3:39:FA:9C:FD:21:6A:0B:06:69:4D:E3:7E:37:52:6B:BE:63:C8:BC:83:74:2E:CB

Alias name: usertrustrsacertificationauthority
Certificate fingerprints:
MD5: 1B:FE:69:D1:91:B7:19:33:A3:72:A8:0F:E1:55:E5:B5
SHA1: 2B:8F:1B:57:33:0D:BB:A2:D0:7A:6C:51:F7:0E:E9:0D:DA:B9:AD:8E
SHA256:
E7:93:C9:B0:2F:D8:AA:13:E2:1C:31:22:8A:CC:B0:81:19:64:3B:74:9C:89:89:64:B1:74:6D:46:C3:D4:CB:D2

Alias name: entrustrootcag2
Certificate fingerprints:
MD5: 4B:E2:C9:91:96:65:0C:F4:0E:5A:93:92:A0:0A:FE:B2
SHA1: 8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4
SHA256:
43:DF:57:74:B0:3E:7F:EF:5F:E4:0D:93:1A:7B:ED:F1:BB:2E:6B:42:73:8C:4E:6D:38:41:10:3D:3A:A7:F3:39

Alias name: networksolutionscertificateauthority
Certificate fingerprints:
MD5: D3:F3:A6:16:C0:FA:6B:1D:59:B1:2D:96:4D:0E:11:2E
SHA1: 74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:DF:CE
SHA256:
15:F0:BA:00:A3:AC:7A:F3:AC:88:4C:07:2B:10:11:A0:77:BD:77:C0:97:F4:01:64:B2:F8:59:8A:BD:83:86:0C

Alias name: trustcenterclass4caii
Certificate fingerprints:
MD5: 9D:FB:F9:AC:ED:89:33:22:F4:28:48:83:25:23:5B:E0
SHA1: A6:9A:91:FD:05:7F:13:6A:42:63:0B:B1:76:0D:2D:51:12:0C:16:50
SHA256:
32:66:96:7E:59:CD:68:00:8D:9D:D3:20:81:11:85:C7:04:20:5E:8D:95:FD:D8:4F:1C:7B:31:1E:67:04:FC:32

Alias name: oistewisekeyglobalrootgaca
Certificate fingerprints:
MD5: BC:6C:51:33:A7:E9:D3:66:63:54:15:72:1B:21:92:93
SHA1: 59:22:A1:E1:5A:EA:16:35:21:F8:98:39:6A:46:46:B0:44:1B:0F:A9
SHA256:
41:C9:23:86:6A:B4:CA:D6:B7:AD:57:80:81:58:2E:02:07:97:A6:CB:DF:4F:FF:78:CE:83:96:B3:89:37:D7:F5

Alias name: verisignuniversalrootcertificationauthority
Certificate fingerprints:
MD5: 8E:AD:B5:01:AA:4D:81:E4:8C:1D:D1:E1:14:00:95:19
SHA1: 36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
SHA256:
23:99:56:11:27:A5:71:25:DE:8C:EF:EA:61:0D:DF:2F:A0:78:B5:C8:06:7F:4E:82:82:90:BF:B8:60:E8:4B:3C

Alias name: ttelesecglobalrootclass3ca
Certificate fingerprints:
MD5: CA:FB:40:A8:4E:39:92:8A:1D:FE:8E:2F:C4:27:EA:EF
SHA1: 55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
SHA256:
FD:73:DA:D3:1C:64:4F:F1:B4:3B:EF:0C:CD:DA:96:71:0B:9C:D9:87:5E:CA:7E:31:70:7A:F3:E9:6D:52:2B:BD

Alias name: starfieldservicesrootg2ca
Certificate fingerprints:
MD5: 17:35:74:AF:7B:61:1C:EB:F4:F9:3C:E2:EE:40:F9:A2
SHA1: 92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
SHA256:
56:8D:69:05:A2:C8:87:08:A4:B3:02:51:90:ED:CF:ED:B1:97:4A:60:6A:13:C6:E5:29:0F:CB:2A:E6:3E:DA:B5
```

```
Alias name: addtrustexternalroot
Certificate fingerprints:
    MD5: 1D:35:54:04:85:78:B0:3F:42:42:4D:BF:20:73:0A:3F
    SHA1: 02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
    SHA256:
        68:7F:A4:51:38:22:78:FF:F0:C8:B1:1F:8D:43:D5:76:67:1C:6E:B2:BC:EA:B4:13:FB:83:D9:65:D0:6D:2F:F2

Alias name: turktrustelektroniksertifikahizmetssaglayicisih5
Certificate fingerprints:
    MD5: DA:70:8E:F0:22:DF:93:26:F6:5F:9F:D3:15:06:52:4E
    SHA1: C4:18:F6:4D:46:D1:DF:00:3D:27:30:13:72:43:A9:12:11:C6:75:FB
    SHA256:
        49:35:1B:90:34:44:C1:85:CC:DC:5C:69:3D:24:D8:55:5C:B2:08:D6:A8:14:13:07:69:9F:4A:F0:63:19:9D:78

Alias name: camerfirmachambersca
Certificate fingerprints:
    MD5: 5E:80:9E:84:5A:0E:65:0B:17:02:F3:55:18:2A:3E:D7
    SHA1: 78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
    SHA256:
        06:3E:4A:FA:C4:91:DF:D3:32:F3:08:9B:85:42:E9:46:17:D8:93:D7:FE:94:4E:10:A7:93:7E:E2:9D:96:93:CO

Alias name: certsignrootca
Certificate fingerprints:
    MD5: 18:98:C0:D6:E9:3A:FC:F9:B0:F5:0C:F7:4B:01:44:17
    SHA1: FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2F:9B
    SHA256:
        EA:A9:62:C4:FA:4A:6B:AF:EB:E4:15:19:6D:35:1C:CD:88:8D:4F:53:F3:FA:8A:E6:D7:C4:66:A9:4E:60:42:BB

Alias name: verisignuniversalrootca
Certificate fingerprints:
    MD5: 8E:AD:B5:01:AA:4D:81:E4:8C:1D:D1:E1:14:00:95:19
    SHA1: 36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
    SHA256:
        23:99:56:11:27:A5:71:25:DE:8C:EF:EA:61:0D:DF:2F:A0:78:B5:C8:06:7F:4E:82:82:90:BF:B8:60:E8:4B:3C

Alias name: geotrustuniversalca
Certificate fingerprints:
    MD5: 92:65:58:8B:A2:1A:31:72:73:68:5C:B4:A5:7A:07:48
    SHA1: E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79
    SHA256:
        A0:45:9B:9F:63:B2:25:59:F5:FA:5D:4C:6D:B3:F9:F7:2F:F1:93:42:03:35:78:F0:73:BF:1D:1B:46:CB:B9:12

Alias name: luxtrustglobalroot2
Certificate fingerprints:
    MD5: B2:E1:09:00:61:AF:F7:F1:91:6F:C4:AD:8D:5E:3B:7C
    SHA1: 1E:0E:56:19:0A:D1:8B:25:98:B2:04:44:FF:66:8A:04:17:99:5F:3F
    SHA256:
        54:45:5F:71:29:C2:0B:14:47:C4:18:F9:97:16:8F:24:C5:8F:C5:02:3B:F5:DA:5B:E2:EB:6E:1D:D8:90:2E:D5

Alias name: twcaglobalrootca
Certificate fingerprints:
    MD5: F9:03:7E:CF:E6:9E:3C:73:7A:2A:90:07:69:FF:2B:96
    SHA1: 9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:AF:65
    SHA256:
        59:76:90:07:F7:68:5D:0F:CD:50:87:2F:9F:95:D5:75:5A:5B:2B:45:7D:81:F3:69:2B:61:0A:98:67:2F:0E:1B

Alias name: tubitakkamusmsslkoksertifikasisurum1
Certificate fingerprints:
    MD5: DC:00:81:DC:69:2F:3E:2F:B0:3B:F6:3D:5A:91:8E:49
    SHA1: 31:43:64:9B:EC:CE:27:EC:ED:3A:3F:0B:8F:0D:E4:E8:91:DD:EE:CA
    SHA256:
        46:ED:C3:68:90:46:D5:3A:45:3F:B3:10:4A:B8:0D:CA:EC:65:8B:26:60:EA:16:29:DD:7E:86:79:90:64:87:16

Alias name: affirmtrustnetworkingca
Certificate fingerprints:
```

```
MD5: 42:65:CA:BE:01:9A:9A:4C:A9:8C:41:49:CD:C0:D5:7F
SHA1: 29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
SHA256:
0A:81:EC:5A:92:97:77:F1:45:90:4A:F3:8D:5D:50:9F:66:B5:E2:C5:8F:CD:B5:31:05:8B:0E:17:F3:F0:B4:1B

Alias name: affirmtrustcommercialca
Certificate fingerprints:
MD5: 82:92:BA:5B:EF:CD:8A:6F:A6:3D:55:F9:84:F6:D6:B7
SHA1: F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
SHA256:
03:76:AB:1D:54:C5:F9:80:3C:E4:B2:E2:01:A0:EE:7E:EF:7B:57:B6:36:E8:A9:3C:9B:8D:48:60:C9:6F:5F:A7

Alias name: godaddyrootcertificateauthorityg2
Certificate fingerprints:
MD5: 80:3A:BC:22:C1:E6:FB:8D:9B:3B:27:4A:32:1B:9A:01
SHA1: 47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
SHA256:
45:14:0B:32:47:EB:9C:C8:C5:B4:F0:D7:B5:30:91:F7:32:92:08:9E:6E:5A:63:E2:74:9D:D3:AC:A9:19:8E:DA

Alias name: starfieldrootg2ca
Certificate fingerprints:
MD5: D6:39:81:C6:52:7E:96:69:FC:FC:CA:66:ED:05:F2:96
SHA1: B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
SHA256:
2C:E1:CB:0B:F9:D2:F9:E1:02:99:3F:BE:21:51:52:C3:B2:DD:0C:AB:DE:1C:68:E5:31:9B:83:91:54:DB:B7:F5

Alias name: dtrustrootclass3ca2ev2009
Certificate fingerprints:
MD5: AA:C6:43:2C:5E:2D:CD:C4:34:CO:50:4F:11:02:4F:B6
SHA1: 96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83
SHA256:
EE:C5:49:6B:98:8C:E9:86:25:B9:34:09:2E:EC:29:08:BE:D0:B0:F3:16:C2:D4:73:0C:84:EA:F1:F3:D3:48:81

Alias name: buypassclass3ca
Certificate fingerprints:
MD5: 3D:3B:18:9E:2C:64:5A:E8:D5:88:CE:0E:F9:37:C2:EC
SHA1: DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
SHA256:
ED:F7:EB:BC:A2:7A:2A:38:4D:38:7B:7D:40:10:C6:66:E2:ED:B4:84:3E:4C:29:B4:AE:1D:5B:93:32:E6:B2:4D

Alias name: verisignclass2g3ca
Certificate fingerprints:
MD5: F8:BE:C4:63:22:C9:A8:46:74:8B:B8:1D:1E:4A:2B:F6
SHA1: 61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:63:11
SHA256:
92:A9:D9:83:3F:E1:94:4D:B3:66:E8:BF:AE:7A:95:B6:48:0C:2D:6C:6C:2A:1B:E6:5D:42:36:B6:08:FC:A1:BB

Alias name: digicerttrustedrootg4
Certificate fingerprints:
MD5: 78:F2:FC:AA:60:1F:2F:B4:EB:C9:37:BA:53:2E:75:49
SHA1: DD:FB:16:CD:49:31:C9:73:A2:03:7D:3F:C8:3A:4D:7D:77:5D:05:E4
SHA256:
55:2F:7B:DC:F1:A7:AF:9E:6C:E6:72:01:7F:4F:12:AB:F7:72:40:C7:8E:76:1A:C2:03:D1:D9:D2:0A:C8:99:88

Alias name: quovadisrootca2g3
Certificate fingerprints:
MD5: AF:0C:86:6E:BF:40:2D:7F:0B:3E:12:50:BA:12:3D:06
SHA1: 09:3C:61:F3:8B:8B:DC:7D:55:DF:75:38:02:05:00:E1:25:F5:C8:36
SHA256:
8F:E4:FB:0A:F9:3A:4D:0D:67:DB:0B:EB:B2:3E:37:C7:1B:F3:25:DC:BC:DD:24:0E:A0:4D:AF:58:B4:7E:18:40

Alias name: geotrustprimarycertificationauthorityg3
Certificate fingerprints:
MD5: B5:E8:34:36:C9:10:44:58:48:70:6D:2E:83:D4:B8:05
SHA1: 03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
```

```
SHA256:  
B4:78:B8:12:25:0D:F8:78:63:5C:2A:A7:EC:7D:15:5E:AA:62:5E:E8:29:16:E2:CD:29:43:61:88:6C:D1:FB:D4

Alias name: geotrustprimarycertificationauthorityg2
Certificate fingerprints:
MD5: 01:5E:D8:6B:BD:6F:3D:8E:A1:31:F8:12:E0:98:73:6A
SHA1: 8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
SHA256:
5E:DB:7A:C4:3B:82:A0:6A:87:61:E8:D7:BE:49:79:EB:F2:61:1F:7D:D7:9B:F9:1C:1C:6B:56:6A:21:9E:D7:66

Alias name: godaddyclass2ca
Certificate fingerprints:
MD5: 91:DE:06:25:AB:DA:FD:32:17:0C:BB:25:17:2A:84:67
SHA1: 27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
SHA256:
C3:84:6B:F2:4B:9E:93:CA:64:27:4C:0E:C6:7C:1E:CC:5E:02:4F:FC:AC:D2:D7:40:19:35:0E:81:FE:54:6A:E4

Alias name: trustcorecal
Certificate fingerprints:
MD5: 27:92:23:1D:0A:F5:40:7C:E9:E6:6B:9D:D8:F5:E7:6C
SHA1: 58:D1:DF:95:95:67:6B:63:C0:F0:5B:1C:17:4D:8B:84:0B:C8:78:BD
SHA256:
5A:88:5D:B1:9C:01:D9:12:C5:75:93:88:93:8C:AF:BB:DF:03:1A:B2:D4:8E:91:EE:15:58:9B:42:97:1D:03:9C

Alias name: hellenicacademicandresearchinstitutionseccrootca2015
Certificate fingerprints:
MD5: 81:E5:B4:17:EB:C2:F5:E1:4B:0D:41:7B:49:92:FE:EF
SHA1: 9F:F1:71:8D:92:D5:9A:F3:7D:74:97:B4:BC:6F:84:68:0B:BA:B6:66
SHA256:
44:B5:45:AA:8A:25:E6:5A:73:CA:15:DC:27:FC:36:D2:4C:1C:B9:95:3A:06:65:39:B1:15:82:DC:48:7B:48:33

Alias name: utnuserfirstobjectca
Certificate fingerprints:
MD5: A7:F2:E4:16:06:41:11:50:30:6B:9C:E3:B4:9C:B0:C9
SHA1: E1:2D:FB:4B:41:D7:D9:C3:2B:30:51:4B:AC:1D:81:D8:38:5E:2D:46
SHA256:
6F:FF:78:E4:00:A7:0C:11:01:1C:D8:59:77:C4:59:FB:5A:F9:6A:3D:F0:54:08:20:D0:F4:B8:60:78:75:E5:8F

Alias name: ttelesecglobalrootclass3
Certificate fingerprints:
MD5: CA:FB:40:A8:4E:39:92:8A:1D:FE:8E:2F:C4:27:EA:EF
SHA1: 55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
SHA256:
FD:73:DA:D3:1C:64:4F:F1:B4:3B:EF:0C:CD:DA:96:71:0B:9C:D9:87:5E:CA:7E:31:70:7A:F3:E9:6D:52:2B:BD

Alias name: ttelesecglobalrootclass2
Certificate fingerprints:
MD5: 2B:9B:9E:E4:7B:6C:1F:00:72:1A:CC:C1:77:79:DF:6A
SHA1: 59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
SHA256:
91:E2:F5:78:8D:58:10:EB:A7:BA:58:73:7D:E1:54:8A:8E:CA:CD:01:45:98:BC:0B:14:3E:04:1B:17:05:25:52

Alias name: addtrustclass1ca
Certificate fingerprints:
MD5: 1E:42:95:02:33:92:6B:B9:5F:CO:7F:DA:D6:B2:4B:FC
SHA1: CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:94:9D
SHA256:
8C:72:09:27:9A:C0:4E:27:5E:16:D0:7F:D3:B7:75:E8:01:54:B5:96:80:46:E3:1F:52:DD:25:76:63:24:E9:A7

Alias name: amzninternalrootca
Certificate fingerprints:
MD5: 08:09:73:AC:E0:78:41:7C:0A:26:33:51:E8:CF:E6:60
SHA1: A7:B7:F6:15:8A:FF:1E:C8:85:13:38:BC:93:EB:A2:AB:A4:09:EF:06
SHA256:
OE:DE:63:C1:DC:7A:8E:11:F1:AB:BC:05:4F:59:EE:49:9D:62:9A:2F:DE:9C:A7:16:32:A2:64:29:3E:8B:66:AA
```

```
Alias name: starfieldrootcertificateauthorityg2
Certificate fingerprints:
MD5: D6:39:81:C6:52:7E:96:69:FC:FC:CA:66:ED:05:F2:96
SHA1: B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
SHA256:
2C:E1:CB:0B:F9:D2:F9:E1:02:99:3F:BE:21:51:52:C3:B2:DD:0C:AB:DE:1C:68:E5:31:9B:83:91:54:DB:B7:F5

Alias name: camerfirmachambersignca
Certificate fingerprints:
MD5: 9E:80:FF:78:01:0C:2E:C1:36:BD:FE:96:90:6E:08:F3
SHA1: 4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
SHA256:
13:63:35:43:93:34:A7:69:80:16:A0:D3:24:DE:72:28:4E:07:9D:7B:52:20:BB:8F:BD:74:78:16:EE:BE:BA:CA

Alias name: secomscrootca2
Certificate fingerprints:
MD5: 6C:39:7D:A4:0E:55:59:B2:3F:D6:41:B1:12:50:DE:43
SHA1: 5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
SHA256:
51:3B:2C:EC:B8:10:D4:CD:E5:DD:85:39:1A:DF:C6:C2:DD:60:D8:7B:B7:36:D2:B5:21:48:4A:A4:7A:0E:BE:F6

Alias name: entrustevca
Certificate fingerprints:
MD5: D6:A5:C3:ED:5D:DD:3E:00:C1:3D:87:92:1F:1D:3F:E4
SHA1: B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
SHA256:
73:C1:76:43:4F:1B:C6:D5:AD:F4:5B:0E:76:E7:27:28:7C:8D:E5:76:16:C1:E6:E6:14:1A:2B:2C:BC:7D:8E:4C

Alias name: secomscrootca1
Certificate fingerprints:
MD5: F1:BC:63:6A:54:E0:B5:27:F5:CD:E7:1A:E3:4D:6E:4A
SHA1: 36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
SHA256:
E7:5E:72:ED:9F:56:0E:EC:6E:B4:80:00:73:A4:3F:C3:AD:19:19:5A:39:22:82:01:78:95:97:4A:99:02:6B:6C

Alias name: affirmtrustcommercial
Certificate fingerprints:
MD5: 82:92:BA:5B:EF:CD:8A:6F:A6:3D:55:F9:84:F6:D6:B7
SHA1: F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
SHA256:
03:76:AB:1D:54:C5:F9:80:3C:E4:B2:E2:01:A0:EE:7E:EF:7B:57:B6:36:E8:A9:3C:9B:8D:48:60:C9:6F:5F:A7

Alias name: digicertassuredidrootg3
Certificate fingerprints:
MD5: 7C:7F:65:31:0C:81:DF:8D:BA:3E:99:E2:5C:AD:6E:FB
SHA1: F5:17:A2:4F:9A:48:C6:C9:F8:A2:00:26:9F:DC:0F:48:2C:AB:30:89
SHA256:
7E:37:CB:8B:4C:47:09:0C:AB:36:55:1B:A6:F4:5D:B8:40:68:0F:BA:16:6A:95:2D:B1:00:71:7F:43:05:3F:C2

Alias name: affirmtrustnetworking
Certificate fingerprints:
MD5: 42:65:CA:BE:01:9A:9A:4C:A9:8C:41:49:CD:C0:D5:7F
SHA1: 29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
SHA256:
0A:81:EC:5A:92:97:77:F1:45:90:4A:F3:8D:5D:50:9F:66:B5:E2:C5:8F:CD:B5:31:05:8B:0E:17:F3:F0:B4:1B

Alias name: izenpecom
Certificate fingerprints:
MD5: A6:B0:CD:85:80:DA:5C:50:34:A3:39:90:2F:55:67:73
SHA1: 2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:E9:19
SHA256:
25:30:CC:8E:98:32:15:02:BA:D9:6F:9B:1F:BA:1B:09:9E:2D:29:9E:0F:45:48:BB:91:4F:36:3B:C0:D4:53:1F

Alias name: amazon-ca-g4-legacy
Certificate fingerprints:
MD5: 6C:E5:BD:67:A4:4F:E3:FD:C2:4C:46:E6:06:5B:6D:55
```

```
SHA1: EA:E7:DE:F9:0A:BE:9F:0B:68:CE:B7:24:0D:80:74:03:BF:6E:B1:6E
SHA256:
CD:72:C4:7F:B4:AD:28:A4:67:2B:E1:86:47:D4:40:E9:3B:16:2D:95:DB:3C:2F:94:BB:81:D9:09:F7:91:24:5E

Alias name: digicertassuredidrootg2
Certificate fingerprints:
MD5: 92:38:B9:F8:63:24:82:65:2C:57:33:E6:FE:81:8F:9D
SHA1: A1:4B:48:D9:43:EE:0A:0E:40:90:4F:3C:EO:A4:C0:91:93:51:5D:3F
SHA256:
7D:05:EB:B6:82:33:9F:8C:94:51:EE:09:4E:EB:FE:FA:79:53:A1:14:ED:B2:F4:49:49:45:2F:AB:7D:2F:C1:85

Alias name: comodoaaaaservicesroot
Certificate fingerprints:
MD5: 49:79:04:B0:EB:87:19:AC:47:B0:BC:11:51:9B:74:D0
SHA1: D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
SHA256:
D7:A7:A0:FB:5D:7E:27:31:D7:71:E9:48:4E:BC:DE:F7:1D:5F:0C:3E:0A:29:48:78:2B:C8:3E:E0:EA:69:9E:F4

Alias name: entrustnetpremium2048secureserverca
Certificate fingerprints:
MD5: EE:29:31:BC:32:7E:9A:E6:E8:B5:F7:51:B4:34:71:90
SHA1: 50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
SHA256:
6D:C4:71:72:E0:1C:BC:B0:BF:62:58:0D:89:5F:E2:B8:AC:9A:D4:F8:73:80:1E:0C:10:B9:C8:37:D2:1E:B1:77

Alias name: trustcorrootcertca2
Certificate fingerprints:
MD5: A2:E1:F8:18:0B:BA:45:D5:C7:41:2A:BB:37:52:45:64
SHA1: B8:BE:6D:CB:56:F1:55:B9:63:D4:12:CA:4E:06:34:C7:94:B2:1C:C0
SHA256:
07:53:E9:40:37:8C:1B:D5:E3:83:6E:39:5D:AE:A5:CB:83:9E:50:46:F1:BD:0E:AE:19:51:CF:10:FE:C7:C9:65

Alias name: entrust2048ca
Certificate fingerprints:
MD5: EE:29:31:BC:32:7E:9A:E6:E8:B5:F7:51:B4:34:71:90
SHA1: 50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
SHA256:
6D:C4:71:72:E0:1C:BC:B0:BF:62:58:0D:89:5F:E2:B8:AC:9A:D4:F8:73:80:1E:0C:10:B9:C8:37:D2:1E:B1:77

Alias name: trustcorrootcertca1
Certificate fingerprints:
MD5: 6E:85:F1:DC:1A:00:D3:22:D5:B2:B2:AC:6B:37:05:45
SHA1: FF:BD:CD:E7:82:C8:43:5E:3C:6F:26:86:5C:CA:A8:3A:45:5B:C3:0A
SHA256:
D4:0E:9C:86:CD:8F:E4:68:C1:77:69:59:F4:9E:A7:74:FA:54:86:84:B6:C4:06:F3:90:92:61:F4:DC:E2:57:5C

Alias name: baltimorecybertrustroot
Certificate fingerprints:
MD5: AC:B6:94:A5:9C:17:E0:D7:91:52:9B:B1:97:06:A6:E4
SHA1: D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
SHA256:
16:AF:57:A9:F6:76:B0:AB:12:60:95:AA:5E:BA:DE:F2:2A:B3:11:19:D6:44:AC:95:CD:4B:93:DB:F3:F2:6A:EB

Alias name: eecertificationcentreroottca
Certificate fingerprints:
MD5: 43:5E:88:D4:7D:1A:4A:7E:FD:84:2E:52:EB:01:D4:6F
SHA1: C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7:25:EB:AF:C3:7B:27:CC:D7
SHA256:
3E:84:BA:43:42:90:85:16:E7:75:73:C0:99:2F:09:79:CA:08:4E:46:85:68:1F:F1:95:CC:BA:8A:22:9B:8A:76

Alias name: dstacescax6
Certificate fingerprints:
MD5: 21:D8:4C:82:2B:99:09:33:A2:EB:14:24:8D:8E:5F:E8
SHA1: 40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:90:1D
SHA256:
76:7C:95:5A:76:41:2C:89:AF:68:8E:90:A1:C7:0F:55:6C:FD:6B:60:25:DB:EA:10:41:6D:7E:B6:83:1F:8C:40
```

```
Alias name: comodocertificationauthority
Certificate fingerprints:
    MD5: 5C:48:DC:F7:42:72:EC:56:94:6D:1C:CC:71:35:80:75
    SHA1: 66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C:BA:6A:BE:D1:F7:BD:EF:7B
    SHA256:
        0C:2C:D6:3D:F7:80:6F:A3:99:ED:E8:09:11:6B:57:5B:F8:79:89:F0:65:18:F9:80:8C:86:05:03:17:8B:AF:66

Alias name: thawteserverca
Certificate fingerprints:
    MD5: EE:FE:61:69:65:6E:F8:9C:C6:2A:F4:D7:2B:63:EF:A2
    SHA1: 9F:AD:91:A6:CE:6A:C6:C5:00:47:C4:4E:C9:D4:A5:0D:92:D8:49:79
    SHA256:
        87:C6:78:BF:B8:B2:5F:38:F7:E9:7B:33:69:56:BB:CF:14:4B:BA:CA:A5:36:47:E6:1A:23:25:BC:10:55:31:6B

Alias name: secomvalicertclass1ca
Certificate fingerprints:
    MD5: 65:58:AB:15:AD:57:6C:1E:A8:A7:B5:69:AC:BF:FF:EB
    SHA1: E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:E4:8E
    SHA256:
        F4:C1:49:55:1A:30:13:A3:5B:C7:BF:FE:17:A7:F3:44:9B:C1:AB:5B:5A:0A:E7:4B:06:C2:3B:90:00:4C:01:04

Alias name: godaddyrootg2ca
Certificate fingerprints:
    MD5: 80:3A:BC:22:C1:E6:FB:8D:9B:3B:27:4A:32:1B:9A:01
    SHA1: 47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
    SHA256:
        45:14:0B:32:47:EB:9C:C8:C5:B4:F0:D7:B5:30:91:F7:32:92:08:9E:6E:5A:63:E2:74:9D:D3:AC:A9:19:8E:DA

Alias name: globalchambersignroot2008
Certificate fingerprints:
    MD5: 9E:80:FF:78:01:0C:2E:C1:36:BD:FE:96:90:6E:08:F3
    SHA1: 4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
    SHA256:
        13:63:35:43:93:34:A7:69:80:16:A0:D3:24:DE:72:28:4E:07:9D:7B:52:20:BB:8F:BD:74:78:16:EE:BE:BA:CA

Alias name: equifaxsecureebusinessca1
Certificate fingerprints:
    MD5: 14:C0:08:E5:A3:85:03:A3:BE:78:E9:67:4F:27:CA:EE
    SHA1: AE:E6:3D:70:E3:76:FB:C7:3A:EB:B0:A1:C1:D4:C4:7A:A7:40:B3:F4
    SHA256:
        2E:3A:2B:B5:11:25:05:83:6C:A8:96:8B:E2:CB:37:27:CE:9B:56:84:5C:6E:E9:8E:91:85:10:4A:FB:9A:F5:96

Alias name: quovaldisrootca3
Certificate fingerprints:
    MD5: 31:85:3C:62:94:97:63:B9:AA:FD:89:4E:AF:6F:E0:CF
    SHA1: 1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
    SHA256:
        18:F1:FC:7F:20:5D:F8:AD:DD:EB:7F:E0:07:DD:57:E3:AF:37:5A:9C:4D:8D:73:54:6B:F4:F1:FE:D1:E1:8D:35

Alias name: usertrustecccertificationauthority
Certificate fingerprints:
    MD5: FA:68:BC:D9:B5:7F:AD:FD:C9:1D:06:83:28:CC:24:C1
    SHA1: D1:CB:CA:5D:B2:D5:2A:7F:69:3B:67:4D:E5:F0:5A:1D:0C:95:7D:F0
    SHA256:
        4F:F4:60:D5:4B:9C:86:DA:BF:BC:FC:57:12:E0:40:0D:2B:ED:3F:BC:4D:4F:BD:AA:86:E0:6A:DC:D2:A9:AD:7A

Alias name: quovaldisrootca2
Certificate fingerprints:
    MD5: 5E:39:7B:DD:F8:BA:EC:82:E9:AC:62:BA:0C:54:00:2B
    SHA1: CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
    SHA256:
        85:A0:DD:7D:D7:20:AD:B7:FF:05:F8:3D:54:2B:20:9D:C7:FF:45:28:F7:D6:77:B1:83:89:FE:A5:E5:C4:9E:86

Alias name: soneraaclass2ca
Certificate fingerprints:
```

```
MD5: A3:EC:75:0F:2E:88:DF:FA:48:01:4E:0B:5C:48:6F:FB
SHA1: 37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
SHA256:
79:08:B4:03:14:C1:38:10:0B:51:8D:07:35:80:7F:FB:FC:F8:51:8A:00:95:33:71:05:BA:38:6B:15:3D:D9:27

Alias name: twcarootcertificationauthority
Certificate fingerprints:
MD5: AA:08:8F:F6:F9:7B:B7:F2:B1:A7:1E:9B:EA:EA:BD:79
SHA1: CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:23:48
SHA256:
BF:D8:8F:E1:10:1C:41:AE:3E:80:1B:F8:BE:56:35:0E:E9:BA:D1:A6:B9:BD:51:5E:DC:5C:6D:5B:87:11:AC:44

Alias name: baltimorecybertrustca
Certificate fingerprints:
MD5: AC:B6:94:A5:9C:17:E0:D7:91:52:9B:B1:97:06:A6:E4
SHA1: D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
SHA256:
16:AF:57:A9:F6:76:B0:AB:12:60:95:AA:5E:BA:DE:F2:2A:B3:11:19:D6:44:AC:95:CD:4B:93:DB:F3:F2:6A:EB

Alias name: cia-crt-g3-01-ca
Certificate fingerprints:
MD5: E3:66:DD:D6:A0:D5:40:8F:FF:29:E2:C0:CB:6E:62:1A
SHA1: 2B:EE:2C:BA:A3:1D:B5:FE:60:40:41:95:08:ED:46:82:39:4D:ED:E2
SHA256:
20:48:AD:4C:EC:90:7F:FA:4A:15:D4:CE:45:E3:C8:E4:2C:EA:78:33:DC:C7:D3:40:48:FC:60:47:27:42:99:EC

Alias name: entrustrootcertificationauthorityg2
Certificate fingerprints:
MD5: 4B:E2:C9:91:96:65:0C:F4:0E:5A:93:92:A0:0A:FE:B2
SHA1: 8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4
SHA256:
43:DF:57:74:B0:3E:7F:EF:5F:E4:0D:93:1A:7B:ED:F1:BB:2E:6B:42:73:8C:4E:6D:38:41:10:3D:3A:A7:F3:39

Alias name: verisignclass3g4ca
Certificate fingerprints:
MD5: 3A:52:E1:E7:FD:6F:3A:E3:6F:F3:6F:99:1B:F9:22:41
SHA1: 22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
SHA256:
69:DD:D7:EA:90:BB:57:C9:3E:13:5D:C8:5E:A6:FC:D5:48:0B:60:32:39:BD:C4:54:FC:75:8B:2A:26:CF:7F:79

Alias name: xrampglobalcaroot
Certificate fingerprints:
MD5: A1:0B:44:B3:CA:10:D8:00:6E:9D:0F:D8:0F:92:0A:D1
SHA1: B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
SHA256:
CE:CD:DC:90:50:99:D8:DA:DF:C5:B1:D2:09:B7:37:CB:E2:C1:8C:FB:2C:10:CO:FF:OB:CF:OD:32:86:FC:1A:A2

Alias name: identrustcommercialrootca1
Certificate fingerprints:
MD5: B3:3E:77:73:75:EE:A0:D3:E3:7E:49:63:49:59:BB:C7
SHA1: DF:71:7E:AA:4A:D9:4E:C9:55:84:99:60:2D:48:DE:5F:BC:F0:3A:25
SHA256:
5D:56:49:9B:E4:D2:E0:8B:CF:CA:D0:8A:3E:38:72:3D:50:50:3B:DE:70:69:48:E4:2F:55:60:30:19:E5:28:AE

Alias name: camerfirmachamberscommerceca
Certificate fingerprints:
MD5: B0:01:EE:14:D9:AF:29:18:94:76:8E:F1:69:33:2A:84
SHA1: 6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
SHA256:
0C:25:8A:12:A5:67:4A:EF:25:F2:8B:A7:DC:FA:EC:EE:A3:48:E5:41:E6:F5:CC:4E:E6:3B:71:B3:61:60:6A:C3

Alias name: verisignclass3g2ca
Certificate fingerprints:
MD5: A2:33:9B:4C:74:78:73:D4:6C:E7:C1:F3:8D:CB:5C:E9
SHA1: 85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
```

```
SHA256:  
83:CE:3C:12:29:68:8A:59:3D:48:5F:81:97:3C:0F:91:95:43:1E:DA:37:CC:5E:36:43:0E:79:C7:A8:88:63:8B  
  
Alias name: deutschetelekomrootca2  
Certificate fingerprints:  
MD5: 74:01:4A:91:B1:08:C4:58:CE:47:CD:F0:DD:11:53:08  
SHA1: 85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF  
SHA256:  
B6:19:1A:50:D0:C3:97:7F:7D:A9:9B:CD:AA:C8:6A:22:7D:AE:B9:67:9E:C7:0B:A3:B0:C9:D9:22:71:C1:70:D3  
  
Alias name: certumca  
Certificate fingerprints:  
MD5: 2C:8F:9F:66:1D:18:90:B1:47:26:9D:8E:86:82:8C:A9  
SHA1: 62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:81:18  
SHA256:  
D8:E0:FE:BC:1D:B2:E3:8D:00:94:0F:37:D2:7D:41:34:4D:99:3E:73:4B:99:D5:65:6D:97:78:D4:D8:14:36:24  
  
Alias name: cybertrustglobalroot  
Certificate fingerprints:  
MD5: 72:E4:4A:87:E3:69:40:80:77:EA:BC:E3:F4:FF:F0:E1  
SHA1: 5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:34:C6  
SHA256:  
96:0A:DF:00:63:E9:63:56:75:0C:29:65:DD:0A:08:67:DA:0B:9C:BD:6E:77:71:4A:EA:FB:23:49:AB:39:3D:A3  
  
Alias name: globalsignrootca  
Certificate fingerprints:  
MD5: 3E:45:52:15:09:51:92:E1:B7:5D:37:9F:B1:87:29:8A  
SHA1: B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C  
SHA256:  
EB:D4:10:40:E4:BB:3E:C7:42:C9:E3:81:D3:1E:F2:A4:1A:48:B6:68:5C:96:E7:CE:F3:C1:DF:6C:D4:33:1C:99  
  
Alias name: secomevrootca1  
Certificate fingerprints:  
MD5: 22:2D:A6:01:EA:7C:0A:F7:F0:6C:56:43:3F:77:76:D3  
SHA1: FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D  
SHA256:  
A2:2D:BA:68:1E:97:37:6E:2D:39:7D:72:8A:AE:3A:9B:62:96:B9:FD:BA:60:BC:2E:11:F6:47:F2:C6:75:FB:37  
  
Alias name: globalsignr3ca  
Certificate fingerprints:  
MD5: C5:DF:B8:49:CA:05:13:55:EE:2D:BA:1A:C3:3E:B0:28  
SHA1: D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD  
SHA256:  
CB:B5:22:D7:B7:F1:27:AD:6A:01:13:86:5B:DF:1C:D4:10:2E:7D:07:59:AF:63:5A:7C:F4:72:0D:C9:63:C5:3B  
  
Alias name: staatdernederlandenrootcag3  
Certificate fingerprints:  
MD5: 0B:46:67:07:DB:10:2F:19:8C:35:50:60:D1:0B:F4:37  
SHA1: D8:EB:6B:41:51:92:59:E0:F3:E7:85:00:C0:3D:B6:88:97:C9:EE:FC  
SHA256:  
3C:4F:B0:B9:5A:B8:B3:00:32:F4:32:B8:6F:53:5F:E1:72:C1:85:D0:FD:39:86:58:37:CF:36:18:7F:A6:F4:28  
  
Alias name: staatdernederlandenrootcag2  
Certificate fingerprints:  
MD5: 7C:A5:0F:F8:5B:9A:7D:6D:30:AE:54:5A:E3:42:A2:8A  
SHA1: 59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:EB:04:DD:B7:16  
SHA256:  
66:8C:83:94:7D:A6:3B:72:4B:EC:E1:74:3C:31:A0:E6:AE:D0:DB:8E:C5:B3:1B:E3:77:BB:78:4F:91:B6:71:6F  
  
Alias name: aolrootca2  
Certificate fingerprints:  
MD5: D6:ED:3C:CA:E2:66:0F:AF:10:43:0D:77:9B:04:09:BF  
SHA1: 85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:92:84  
SHA256:  
7D:3B:46:5A:60:14:E5:26:C0:AF:FC:EE:21:27:D2:31:17:27:AD:81:1C:26:84:2D:00:6A:F3:73:06:CC:80:BD
```

```
Alias name: dstrootcax3
Certificate fingerprints:
MD5: 41:03:52:DC:0F:F7:50:1B:16:F0:02:8E:BA:6F:45:C5
SHA1: DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7C:13
SHA256:
06:87:26:03:31:A7:24:03:D9:09:F1:05:E6:9B:CF:0D:32:E1:BD:24:93:FF:C6:D9:20:6D:11:BC:D6:77:07:39

Alias name: trustcenteruniversalca1
Certificate fingerprints:
MD5: 45:E1:A5:72:C5:A9:36:64:40:9E:F5:E4:58:84:67:8C
SHA1: 6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:39:F3
SHA256:
EB:F3:C0:2A:87:89:B1:FB:7D:51:19:95:D6:63:B7:29:06:D9:13:CE:0D:5E:10:56:8A:8A:77:E2:58:61:67:E7

Alias name: aolrootca1
Certificate fingerprints:
MD5: 14:F1:08:AD:9D:FA:64:E2:89:E7:1C:CF:A8:AD:7D:5E
SHA1: 39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
SHA256:
77:40:73:12:C6:3A:15:3D:5B:C0:0B:4E:51:75:9C:DF:DA:C2:37:DC:2A:33:B6:79:46:E9:8E:9B:FA:68:0A:E3

Alias name: affirmtrustpremiumecc
Certificate fingerprints:
MD5: 64:B0:09:55:CF:B1:D5:99:E2:BE:13:AB:A6:5D:EA:4D
SHA1: B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
SHA256:
BD:71:FD:F6:DA:97:E4:CF:62:D1:64:7A:DD:25:81:B0:7D:79:AD:F8:39:7E:B4:EC:BA:9C:5E:84:88:82:14:23

Alias name: microseceszignorootca2009
Certificate fingerprints:
MD5: F8:49:F4:03:BC:44:2D:83:BE:48:69:7D:29:64:FC:B1
SHA1: 89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:31:8E
SHA256:
3C:5F:81:FE:A5:FA:B8:2C:64:BF:A2:EA:EC:AF:CD:E8:E0:77:FC:86:20:A7:CA:E5:37:16:3D:F3:6E:DB:F3:78

Alias name: verisignclass1g3ca
Certificate fingerprints:
MD5: B1:47:BC:18:57:D1:18:A0:78:2D:EC:71:E8:2A:95:73
SHA1: 20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:46:A5
SHA256:
CB:B5:AF:18:5E:94:2A:24:02:F9:EA:CB:C0:ED:5B:B8:76:EE:A3:C1:22:36:23:D0:04:47:E4:F3:BA:55:4B:65

Alias name: certplusrootcag2
Certificate fingerprints:
MD5: A7:EE:C4:78:2D:1B:EE:2D:B9:29:CE:D6:A7:96:32:31
SHA1: 4F:65:8E:1F:E9:06:D8:28:02:E9:54:47:41:C9:54:25:5D:69:CC:1A
SHA256:
6C:C0:50:41:E6:44:5E:74:69:6C:4C:FB:C9:F8:0F:54:3B:7E:AB:BB:44:B4:CE:6F:78:7C:6A:99:71:C4:2F:17

Alias name: certplusrootcag1
Certificate fingerprints:
MD5: 7F:09:9C:F7:D9:B9:5C:69:69:56:D5:37:3E:14:0D:42
SHA1: 22:FD:D0:B7:FD:A2:4E:0D:AC:49:2C:A0:AC:A6:7B:6A:1F:E3:F7:66
SHA256:
15:2A:40:2B:FC:DF:2C:D5:48:05:4D:22:75:B3:9C:7F:CA:3E:C0:97:80:78:B0:F0:EA:76:E5:61:A6:C7:43:3E

Alias name: addtrustexternalalca
Certificate fingerprints:
MD5: 1D:35:54:04:85:78:B0:3F:42:42:4D:BF:20:73:0A:3F
SHA1: 02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
SHA256:
68:7F:A4:51:38:22:78:FF:F0:C8:B1:1F:8D:43:D5:76:67:1C:6E:B2:BC:EA:B4:13:FB:83:D9:65:D0:6D:2F:F2

Alias name: entrustrootcertificationauthority
Certificate fingerprints:
MD5: D6:A5:C3:ED:5D:DD:3E:00:C1:3D:87:92:1F:1D:3F:E4
```

```
SHA1: B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
SHA256:
73:C1:76:43:4F:1B:C6:D5:AD:F4:5B:0E:76:E7:27:28:7C:8D:E5:76:16:C1:E6:E6:14:1A:2B:2C:BC:7D:8E:4C

Alias name: verisignclass3ca
Certificate fingerprints:
MD5: EF:5A:F1:33:EF:F1:CD:BB:51:02:EE:12:14:4B:96:C4
SHA1: A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
SHA256:
A4:B6:B3:99:6F:C2:F3:06:B3:FD:86:81:BD:63:41:3D:8C:50:09:CC:4F:A3:29:C2:CC:F0:E2:FA:1B:14:03:05

Alias name: digicertassuredidrootca
Certificate fingerprints:
MD5: 87:CE:OB:7B:2A:0E:49:00:E1:58:71:9B:37:A8:93:72
SHA1: 05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:43
SHA256:
3E:90:99:B5:01:5E:8F:48:6C:00:BC:EA:9D:11:1E:E7:21:FA:BA:35:5A:89:BC:F1:DF:69:56:1E:3D:C6:32:5C

Alias name: globalsignrootcar3
Certificate fingerprints:
MD5: C5:DF:B8:49:CA:05:13:55:EE:2D:BA:1A:C3:3E:B0:28
SHA1: D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
SHA256:
CB:B5:22:D7:B7:F1:27:AD:6A:01:13:86:5B:DF:1C:D4:10:2E:7D:07:59:AF:63:5A:7C:F4:72:0D:C9:63:C5:3B

Alias name: globalsignrootcar2
Certificate fingerprints:
MD5: 94:14:77:7E:3E:5E:FD:8F:30:BD:41:B0:CF:E7:D0:30
SHA1: 75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
SHA256:
CA:42:DD:41:74:5F:D0:B8:1E:B9:02:36:2C:F9:D8:BF:71:9D:A1:BD:1B:1E:FC:94:6F:5B:4C:99:F4:2C:1B:9E

Alias name: verisignclass1ca
Certificate fingerprints:
MD5: 86:AC:DE:2B:C5:6D:C3:D9:8C:28:88:D3:8D:16:13:1E
SHA1: CE:6A:64:A3:09:E4:2F:BB:D9:85:1C:45:3E:64:09:EA:E8:7D:60:F1
SHA256:
51:84:7C:8C:BD:2E:9A:72:C9:1E:29:2D:2A:E2:47:D7:DE:1E:3F:D2:70:54:7A:20:EF:7D:61:0F:38:B8:84:2C

Alias name: thawtepremiumserverca
Certificate fingerprints:
MD5: A6:6B:60:90:23:9B:3F:2D:BB:98:6F:D6:A7:19:0D:46
SHA1: E0:AB:05:94:20:72:54:93:05:60:62:02:36:70:F7:CD:2E:FC:66:66
SHA256:
3F:9F:27:D5:83:20:4B:9E:09:C8:A3:D2:06:6C:4B:57:D3:A2:47:9C:36:93:65:08:80:50:56:98:10:5D:BC:E9

Alias name: verisigntsaca
Certificate fingerprints:
MD5: F2:89:95:6E:4D:05:F0:F1:A7:21:55:7D:46:11:BA:47
SHA1: 20:CE:B1:F0:F5:1C:0E:19:A9:F3:8D:B1:AA:8E:03:8C:AA:7A:C7:01
SHA256:
CB:6B:05:D9:E8:E5:7C:D8:82:B1:0B:4D:B7:0D:E4:BB:1D:E4:2B:A4:8A:7B:D0:31:8B:63:5B:F6:E7:78:1A:9D

Alias name: thawteprimaryrootca
Certificate fingerprints:
MD5: 8C:CA:DC:0B:22:CE:F5:BE:72:AC:41:1A:11:A8:D8:12
SHA1: 91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
SHA256:
8D:72:2F:81:A9:C1:13:C0:79:1D:F1:36:A2:96:6D:B2:6C:95:0A:97:1D:B4:6B:41:99:F4:EA:54:B7:8B:FB:9F

Alias name: visaecommerceroot
Certificate fingerprints:
MD5: FC:11:B8:D8:08:93:30:00:6D:23:F9:7E:EB:52:1E:02
SHA1: 70:17:9B:86:8C:00:A4:FA:60:91:52:22:3F:9F:3E:32:BD:E0:05:62
SHA256:
69:FA:C9:BD:55:FB:0A:C7:8D:53:BB:EE:5C:F1:D5:97:98:9F:D0:AA:AB:20:A2:51:51:BD:F1:73:3E:E7:D1:22
```

```
Alias name: digicertglobalrootg3
Certificate fingerprints:
    MD5: F5:5D:A4:50:A5:FB:28:7E:1E:0F:0D:CC:96:57:56:CA
    SHA1: 7E:04:DE:89:6A:3E:66:6D:00:E6:87:D3:3F:FA:D9:3B:E8:3D:34:9E
    SHA256:
        31:AD:66:48:F8:10:41:38:C7:38:F3:9E:A4:32:01:33:39:3E:3A:18:CC:02:29:6E:F9:7C:2A:C9:EF:67:31:D0

Alias name: xrampglobalca
Certificate fingerprints:
    MD5: A1:0B:44:B3:CA:10:D8:00:6E:9D:0F:D8:0F:92:0A:D1
    SHA1: B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
    SHA256:
        CE:CD:DC:90:50:99:D8:DA:DF:C5:B1:D2:09:B7:37:CB:E2:C1:8C:FB:2C:10:CO:FF:OB:CF:OD:32:86:FC:1A:A2

Alias name: digicertglobalrootg2
Certificate fingerprints:
    MD5: E4:A6:8A:C8:54:AC:52:42:46:0A:FD:72:48:1B:2A:44
    SHA1: DF:3C:24:F9:BF:D6:66:76:1B:26:80:73:FE:06:D1:CC:8D:4F:82:A4
    SHA256:
        CB:3C:CB:B7:60:31:E5:E0:13:8F:8D:D3:9A:23:F9:DE:47:FF:C3:5E:43:C1:14:4C:EA:27:D4:6A:5A:B1:CB:5F

Alias name: valicertclass2ca
Certificate fingerprints:
    MD5: A9:23:75:9B:BA:49:36:6E:31:C2:DB:F2:E7:66:BA:87
    SHA1: 31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:FC:A6
    SHA256:
        58:D0:17:27:9C:D4:DC:63:AB:DD:B1:96:A6:C9:90:6C:30:C4:E0:87:83:EA:E8:C1:60:99:54:D6:93:55:59:6B

Alias name: geotrustprimaryca
Certificate fingerprints:
    MD5: 02:26:C3:01:5E:08:30:37:43:A9:D0:7D:CF:37:E6:BF
    SHA1: 32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
    SHA256:
        37:D5:10:06:C5:12:EA:AB:62:64:21:F1:EC:8C:92:01:3F:C5:F8:2A:E9:8E:E5:33:EB:46:19:B8:DE:B4:D0:6C

Alias name: netlockaranyclassgoldfotanusitvany
Certificate fingerprints:
    MD5: C5:A1:B7:FF:73:DD:D6:D7:34:32:18:DF:FC:3C:AD:88
    SHA1: 06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:09:91
    SHA256:
        6C:61:DA:C3:A2:DE:F0:31:50:6B:E0:36:D2:A6:FE:40:19:94:FB:D1:3D:F9:C8:D4:66:59:92:74:C4:46:EC:98

Alias name: geotrustglobalca
Certificate fingerprints:
    MD5: F7:75:AB:29:FB:51:4E:B7:77:5E:FF:05:3C:99:8E:F5
    SHA1: DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
    SHA256:
        FF:85:6A:2D:25:1D:CD:88:D3:66:56:F4:50:12:67:98:CF:AB:AA:DE:40:79:9C:72:2D:E4:D2:B5:DB:36:A7:3A

Alias name: oistewisekeyglobalrootgbca
Certificate fingerprints:
    MD5: A4:EB:B9:61:28:2E:B7:2F:98:B0:35:26:90:99:51:1D
    SHA1: 0F:F9:40:76:18:D3:D7:6A:4B:98:F0:A8:35:9E:0C:FD:27:AC:CC:ED
    SHA256:
        6B:9C:08:E8:6E:B0:F7:67:CF:AD:65:CD:98:B6:21:49:E5:49:4A:67:F5:84:5E:7B:D1:ED:01:9F:27:B8:6B:D6

Alias name: certumtrustednetworkca2
Certificate fingerprints:
    MD5: 6D:46:9E:D9:25:6D:08:23:5B:5E:74:7D:1E:27:DB:F2
    SHA1: D3:DD:48:3E:2B:BF:4C:05:E8:AF:10:F5:FA:76:26:CF:D3:DC:30:92
    SHA256:
        B6:76:F2:ED:DA:E8:77:5C:D3:6C:B0:F6:3C:D1:D4:60:39:61:F4:9E:62:65:BA:01:3A:2F:03:07:B6:DO:B8:04

Alias name: starfieldservicesrootcertificateauthorityg2
Certificate fingerprints:
```

```
MD5: 17:35:74:AF:7B:61:1C:EB:F4:F9:3C:E2:EE:40:F9:A2
SHA1: 92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
SHA256:
56:8D:69:05:A2:C8:87:08:A4:B3:02:51:90:ED:CF:ED:B1:97:4A:60:6A:13:C6:E5:29:0F:CB:2A:E6:3E:DA:B5

Alias name: comodorsacertificationauthority
Certificate fingerprints:
MD5: 1B:31:B0:71:40:36:CC:14:36:91:AD:C4:3E:FD:EC:18
SHA1: AF:E5:D2:44:A8:D1:19:42:30:FF:47:9F:E2:F8:97:BB:CD:7A:8C:B4
SHA256:
52:F0:E1:C4:E5:8E:C6:29:29:1B:60:31:7F:07:46:71:B8:5D:7E:A8:0D:5B:07:27:34:63:53:4B:32:B4:02:34

Alias name: comodoaaaca
Certificate fingerprints:
MD5: 49:79:04:B0:EB:87:19:AC:47:B0:BC:11:51:9B:74:D0
SHA1: D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
SHA256:
D7:A7:A0:FB:5D:7E:27:31:D7:71:E9:48:4E:BC:DE:F7:1D:5F:0C:3E:0A:29:48:78:2B:C8:3E:E0:EA:69:9E:F4

Alias name: identrustpublicsectorrootca1
Certificate fingerprints:
MD5: 37:06:A5:B0:FC:89:9D:BA:F4:6B:8C:1A:64:CD:D5:BA
SHA1: BA:29:41:60:77:98:3F:F4:F3:EF:F2:31:05:3B:2E:EA:6D:4D:45:FD
SHA256:
30:D0:89:5A:9A:44:8A:26:20:91:63:55:22:D1:F5:20:10:B5:86:7A:CA:E1:2C:78:EF:95:8F:D4:F4:38:9F:2F

Alias name: certplusclass2primaryca
Certificate fingerprints:
MD5: 88:2C:8C:52:B8:A2:3C:F3:F7:BB:03:EA:AE:AC:42:0B
SHA1: 74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB
SHA256:
0F:99:3C:8A:EF:97:BA:AF:56:87:14:0E:D5:9A:D1:82:1B:B4:AF:AC:F0:AA:9A:58:B5:D5:7A:33:8A:3A:FB:CB

Alias name: ttelesecglobalrootclass2ca
Certificate fingerprints:
MD5: 2B:9B:9E:E4:7B:6C:1F:00:72:1A:CC:C1:77:79:DF:6A
SHA1: 59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
SHA256:
91:E2:F5:78:8D:58:10:EB:A7:BA:58:73:7D:E1:54:8A:8E:CA:CD:01:45:98:BC:0B:14:3E:04:1B:17:05:25:52

Alias name: accvraiz1
Certificate fingerprints:
MD5: D0:A0:5A:EE:05:B6:09:94:21:A1:7D:F1:B2:29:82:02
SHA1: 93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17
SHA256:
9A:6E:C0:12:E1:A7:DA:9D:BE:34:19:4D:47:8A:D7:C0:DB:18:22:FB:07:1D:F1:29:81:49:6E:D1:04:38:41:13

Alias name: digicerthighassuranceevrootca
Certificate fingerprints:
MD5: D4:74:DE:57:5C:39:B2:D3:9C:85:83:C5:C0:65:49:8A
SHA1: 5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
SHA256:
74:31:E5:F4:C3:C1:CE:46:90:77:4F:0B:61:E0:54:40:88:3B:A9:A0:1E:D0:0B:A6:AB:D7:80:6E:D3:B1:18:CF

Alias name: amzninternalinfoseccag3
Certificate fingerprints:
MD5: E9:34:94:02:BA:BB:31:6B:22:E6:2B:A9:C4:F0:26:04
SHA1: B9:B1:CA:38:F7:BF:9C:D2:D4:95:E7:B6:5E:75:32:9B:A8:78:2E:F6
SHA256:
81:03:0B:C7:E2:54:DA:7B:F8:B7:45:DB:DD:41:15:89:B5:A3:81:86:FB:4B:29:77:1F:84:0A:18:D9:67:6D:68

Alias name: cia-crt-g3-02-ca
Certificate fingerprints:
MD5: FD:B9:23:FD:D3:EB:2D:3E:57:EF:56:FF:DB:D3:E4:B9
SHA1: 96:4A:BB:A7:BD:DA:FC:97:34:CO:0A:2D:F0:05:98:F7:E6:C6:6F:09
```

```
SHA256:  
93:F1:72:FB:BA:43:31:5C:06:EE:0F:9F:04:89:B8:F6:88:BC:75:15:3C:BE:B4:80:AC:A7:14:3A:F6:FC:4A:C1  
  
Alias name: entrustrootcertificationauthorityec1  
Certificate fingerprints:  
MD5: B6:7E:1D:F0:58:C5:49:6C:24:3B:3D:ED:98:18:ED:BC  
SHA1: 20:D8:06:40:DF:9B:25:F5:12:25:3A:11:EA:F7:59:8A:EB:14:B5:47  
SHA256:  
02:ED:0E:B2:8C:14:DA:45:16:5C:56:67:91:70:0D:64:51:D7:FB:56:F0:B2:AB:1D:3B:8E:B0:70:E5:6E:DF:F5  
  
Alias name: securitycommunicationrootca  
Certificate fingerprints:  
MD5: F1:BC:63:6A:54:E1:B5:27:F5:CD:E7:1A:E3:4D:6E:4A  
SHA1: 36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7  
SHA256:  
E7:5E:72:ED:9F:56:0E:EC:6E:B4:80:00:73:A4:3F:C3:AD:19:19:5A:39:22:82:01:78:95:97:4A:99:02:6B:6C  
  
Alias name: globalsignca  
Certificate fingerprints:  
MD5: 3E:45:52:15:09:51:92:E1:B7:5D:37:9F:B1:87:29:8A  
SHA1: B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C  
SHA256:  
EB:D4:10:40:E4:BB:3E:C7:42:C9:E3:81:D3:1E:F2:A4:1A:48:B6:68:5C:96:E7:CE:F3:C1:DF:6C:D4:33:1C:99  
  
Alias name: trustcenterclass2caii  
Certificate fingerprints:  
MD5: CE:78:33:5C:59:78:01:6E:18:EA:B9:36:A0:B9:2E:23  
SHA1: AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:79:42:21:15:6E  
SHA256:  
E6:B8:F8:76:64:85:F8:07:AE:7F:8D:AC:16:70:46:1F:07:C0:A1:3E:EF:3A:1F:F7:17:53:8D:7A:BA:D3:91:B4  
  
Alias name: camerfirmachambersofcommerceroott  
Certificate fingerprints:  
MD5: B0:01:EE:14:D9:AF:29:18:94:76:8E:F1:69:33:2A:84  
SHA1: 6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1  
SHA256:  
0C:25:8A:12:A5:67:4A:EF:25:F2:8B:A7:DC:FA:EC:EE:A3:48:E5:41:E6:F5:CC:4E:E6:3B:71:B3:61:60:6A:C3  
  
Alias name: geotrustprimarycag3  
Certificate fingerprints:  
MD5: B5:E8:34:36:C9:10:44:58:48:70:6D:2E:83:D4:B8:05  
SHA1: 03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD  
SHA256:  
B4:78:B8:12:25:0D:F8:78:63:5C:2A:A7:EC:7D:15:5E:AA:62:5E:E8:29:16:E2:CD:29:43:61:88:6C:D1:FB:D4  
  
Alias name: geotrustprimarycag2  
Certificate fingerprints:  
MD5: 01:5E:D8:6B:BD:6F:3D:8E:A1:31:F8:12:E0:98:73:6A  
SHA1: 8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0  
SHA256:  
5E:DB:7A:C4:3B:82:A0:6A:87:61:E8:D7:BE:49:79:EB:F2:61:1F:7D:D7:9B:F9:1C:1C:6B:56:6A:21:9E:D7:66  
  
Alias name: hongkongpostrootca1  
Certificate fingerprints:  
MD5: A8:0D:6F:39:78:B9:43:6D:77:42:6D:98:5A:CC:23:CA  
SHA1: D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:B2:58:B2:8A:58  
SHA256:  
F9:E6:7D:33:6C:51:00:2A:C0:54:C6:32:02:2D:66:DD:A2:E7:E3:FF:F1:0A:D0:61:ED:31:D8:BB:B4:10:CF:B2  
  
Alias name: affirmtrustpremiumeccca  
Certificate fingerprints:  
MD5: 64:B0:09:55:CF:B1:D5:99:E2:BE:13:AB:A6:5D:EA:4D  
SHA1: B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB  
SHA256:  
BD:71:FD:F6:DA:97:E4:CF:62:D1:64:7A:DD:25:81:B0:7D:79:AD:F8:39:7E:B4:EC:BA:9C:5E:84:88:82:14:23
```

```
Alias name: hellenicacademicandresearchinstitutionsrootca2015
Certificate fingerprints:
    MD5: CA:FF:E2:DB:03:D9:CB:4B:E9:0F:AD:84:FD:7B:18:CE
    SHA1: 01:0C:06:95:A6:98:19:14:FF:BF:5F:C6:B0:B6:95:EA:29:E9:12:A6
    SHA256:
        A0:40:92:9A:02:CE:53:B4:AC:F4:F2:FF:C6:98:1C:E4:49:6F:75:5E:6D:45:FE:0B:2A:69:2B:CD:52:52:3F:36
```

IoT Analytics

The IoT Analytics (`iotAnalytics`) action sends data from an MQTT message to an AWS IoT Analytics channel.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `iotanalytics:BatchPutMessage` operation. For more information, see [Granting AWS IoT the required access \(p. 444\)](#).

In the AWS IoT console, you can choose or create a role to allow AWS IoT to perform this rule action.

The policy attached to the role you specify should look like the following example.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iotanalytics:BatchPutMessage",
            "Resource": [
                "arn:aws:iotanalytics:us-west-2:account-id:channel/mychannel"
            ]
        }
    ]
}
```

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`batchMode`

(Optional) Whether to process the action as a batch. The default value is `false`.

When `batchMode` is `true` and the rule SQL statement evaluates to an Array, each Array element is delivered as a separate message when passed by `BatchPutMessage` to the AWS IoT Analytics channel. The resulting array can't have more than 100 messages.

Supports [substitution templates \(p. 586\)](#): No

`channelName`

The name of the AWS IoT Analytics channel to which to write the data.

Supports [substitution templates \(p. 586\)](#): API and AWS CLI only

`roleArn`

The IAM role that allows access to the AWS IoT Analytics channel. For more information, see [Requirements \(p. 495\)](#).

Supports [substitution templates \(p. 586\)](#): No

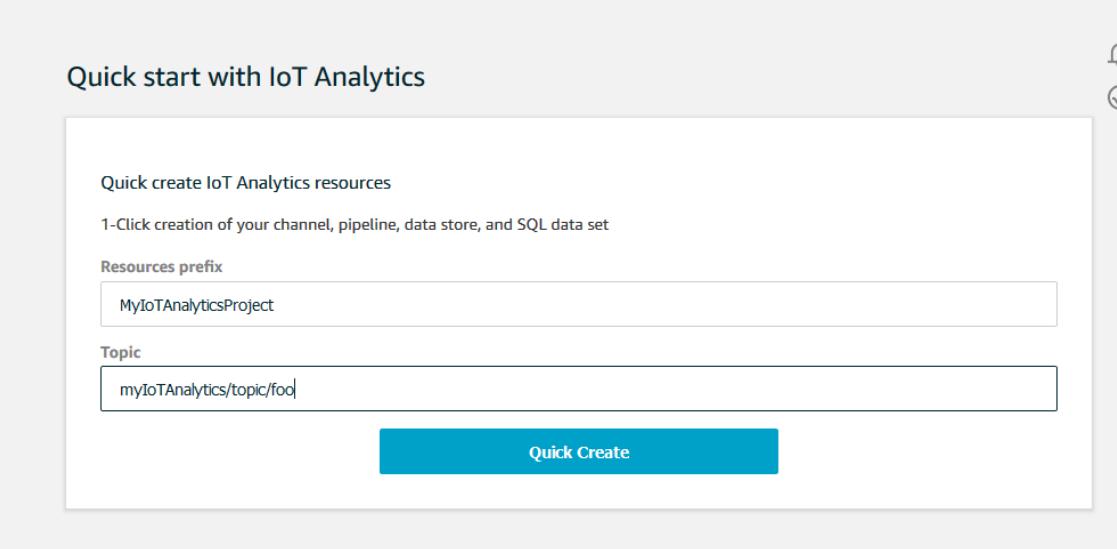
Examples

The following JSON example defines an IoT Analytics action in an AWS IoT rule.

```
{  
    "topicRulePayload": {  
        "sql": "SELECT * FROM 'some/topic'",  
        "ruleDisabled": false,  
        "awsIotSqlVersion": "2016-03-23",  
        "actions": [  
            {  
                "iotAnalytics": {  
                    "channelName": "mychannel",  
                    "roleArn": "arn:aws:iam::123456789012:role/analyticsRole",  
                }  
            }  
        ]  
    }  
}
```

See also

- [What is AWS IoT Analytics?](#) in the *AWS IoT Analytics User Guide*
- The AWS IoT Analytics console also has a **Quick start** feature that lets you create a channel, data store, pipeline, and data store with one click. For more information, see [AWS IoT Analytics console quickstart guide](#) in the *AWS IoT Analytics User Guide*.



IoT Events

The IoT Events (`iotEvents`) action sends data from an MQTT message to an AWS IoT Events input.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `iotevents:BatchPutMessage` operation. For more information, see [Granting AWS IoT the required access \(p. 444\)](#).

In the AWS IoT console, you can choose or create a role to allow AWS IoT to perform this rule action.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`batchMode`

(Optional) Whether to process the event actions as a batch. The default value is `false`.

When `batchMode` is `true` and the rule SQL statement evaluates to an Array, each Array element is treated as a separate message when it's sent to AWS IoT Events by calling [BatchPutMessage](#). The resulting array can't have more than 10 messages.

When `batchMode` is `true`, you can't specify a `messageId`.

Supports [substitution templates \(p. 586\)](#): No

`inputName`

The name of the AWS IoT Events input.

Supports [substitution templates \(p. 586\)](#): API and AWS CLI only

`messageId`

(Optional) Use this to ensure that only one input (message) with a given `messageId` is processed by an AWS IoT Events detector. You can use the `#{newuuid()}` substitution template to generate a unique ID for each request.

When `batchMode` is `true`, you can't specify a `messageId`--a new UUID value will be assigned.

Supports [substitution templates \(p. 586\)](#): Yes

`roleArn`

The IAM role that allows AWS IoT to send an input to an AWS IoT Events detector. For more information, see [Requirements \(p. 496\)](#).

Supports [substitution templates \(p. 586\)](#): No

Examples

The following JSON example defines an IoT Events action in an AWS IoT rule.

```
{  
    "topicRulePayload": {  
        "sql": "SELECT * FROM 'some/topic'",  
        "ruleDisabled": false,  
        "awsIotSqlVersion": "2016-03-23",  
        "actions": [  
            {  
                "iotEvents": {  
                    "inputName": "MyIoTEventsInput",  
                    "messageId": " #{newuuid()} ",  
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_events"  
                }  
            }  
        ]  
    }  
}
```

```
        ]
    }
}
```

See also

- [What is AWS IoT Events?](#) in the *AWS IoT Events Developer Guide*

IoT SiteWise

The IoT SiteWise (`iotSiteWise`) action sends data from an MQTT message to asset properties in AWS IoT SiteWise.

You can follow a tutorial that shows you how to ingest data from AWS IoT things. For more information, see [Ingesting data to AWS IoT SiteWise from AWS IoT things](#) in the *AWS IoT SiteWise User Guide*.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `iotsitewise:BatchPutAssetPropertyValue` operation. For more information, see [Granting AWS IoT the required access \(p. 444\)](#).

You can attach the following example trust policy to the role.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iotsitewise:BatchPutAssetPropertyValue",
            "Resource": "*"
        }
    ]
}
```

To improve security, you can specify an AWS IoT SiteWise asset hierarchy path in the `Condition` property. The following example is a trust policy that specifies an asset hierarchy path.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iotsitewise:BatchPutAssetPropertyValue",
            "Resource": "*",
            "Condition": {
                "StringLike": {
                    "iotsitewise:assetHierarchyPath": [
                        "/root node asset ID",
                        "/root node asset ID/*"
                    ]
                }
            }
        }
    ]
}
```

}

- When you send data to AWS IoT SiteWise with this action, your data must meet the requirements of the `BatchPutAssetPropertyValue` operation. For more information, see [BatchPutAssetPropertyValue](#) in the *AWS IoT SiteWise API Reference*.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`putAssetPropertyValueEntries`

A list of asset property value entries that each contain the following information:

`propertyAlias`

(Optional) The property alias associated with your asset property. You must specify either a `propertyAlias` or both an `assetId` and a `propertyId`. For more information about property aliases, see [Mapping industrial data streams to asset properties](#) in the *AWS IoT SiteWise User Guide*.

Supports [substitution templates \(p. 586\)](#): Yes

`assetId`

(Optional) The ID of the AWS IoT SiteWise asset. You must specify either a `propertyAlias` or both an `assetId` and a `propertyId`.

Supports [substitution templates \(p. 586\)](#): Yes

`propertyId`

(Optional) The ID of the asset's property. You must specify either a `propertyAlias` or both an `assetId` and a `propertyId`.

Supports [substitution templates \(p. 586\)](#): API and AWS CLI only

`entryId`

(Optional) A unique identifier for this entry. You can define the `entryId` to better track which message caused an error in case of failure. Defaults to a new UUID.

Supports [substitution templates \(p. 586\)](#): Yes

`propertyValues`

A list of property values to insert that each contain timestamp, quality, and value (TQV) in the following format:

`timestamp`

A timestamp structure that contains the following information:

`timeInSeconds`

A string that contains the time in seconds in Unix epoch time. If your message payload doesn't have a timestamp, you can use [timestamp\(\) \(p. 578\)](#), which returns the current time in milliseconds. To convert that time to seconds, you can use the following substitution template: `#{floor(timestamp() / 1E3)}`.

Supports [substitution templates \(p. 586\)](#): Yes

`offsetInNanos`

(Optional) A string that contains the nanosecond time offset from the time in seconds. If your message payload doesn't have a timestamp, you can use [timestamp\(\) \(p. 578\)](#),

which returns the current time in milliseconds. To calculate the nanosecond offset from that time, you can use the following substitution template: `$(timestamp() % 1E3) * 1E6}`.

Supports [substitution templates \(p. 586\)](#): Yes

With respect to Unix epoch time, AWS IoT SiteWise accepts only entries that have a timestamp of up to 7 days in the past and up to 5 minutes in the future.

`quality`

(Optional) A string that describes the quality of the value. Valid values: GOOD, BAD, UNCERTAIN.

Supports [substitution templates \(p. 586\)](#): Yes

`value`

A value structure that contains one of the following value fields, depending on the asset property's data type:

`booleanValue`

(Optional) A string that contains the Boolean value of the value entry.

Supports [substitution templates \(p. 586\)](#): Yes

`doubleValue`

(Optional) A string that contains the double value of the value entry.

Supports [substitution templates \(p. 586\)](#): Yes

`integerValue`

(Optional) A string that contains the integer value of the value entry.

Supports [substitution templates \(p. 586\)](#): Yes

`stringValue`

(Optional) The string value of the value entry.

Supports [substitution templates \(p. 586\)](#): Yes

`roleArn`

The ARN of the IAM role that grants AWS IoT permission to send an asset property value to AWS IoT SiteWise. For more information, see [Requirements \(p. 498\)](#).

Supports [substitution templates \(p. 586\)](#): No

Examples

The following JSON example defines a basic IoT SiteWise action in an AWS IoT rule.

```
{  
    "topicRulePayload": {  
        "sql": "SELECT * FROM 'some/topic'",  
        "ruleDisabled": false,  
        "awsIotSqlVersion": "2016-03-23",  
        "actions": [  
            {  
                "iotSiteWise": {
```

```
        "putAssetPropertyValueEntries": [
            {
                "propertyAlias": "/some/property/alias",
                "propertyValues": [
                    {
                        "timestamp": {
                            "timeInSeconds": "${my.payload.timeInSeconds}"
                        },
                        "value": {
                            "integerValue": "${my.payload.value}"
                        }
                    }
                ]
            }
        ],
        "roleArn": "arn:aws:iam::123456789012:role/aws_iot_sitewise"
    }
}
]
```

The following JSON example defines an IoT SiteWise action in an AWS IoT rule. This example uses the topic as the property alias and the `timestamp()` function. For example, if you publish data to `/company/windfarm/3/turbine/7/rpm`, this action sends the data to the asset property with a property alias that is the same as the topic that you specified.

```
{
    "topicRulePayload": {
        "sql": "SELECT * FROM '/company/windfarm/+/{turbine}/+'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "iotSiteWise": {
                    "putAssetPropertyValueEntries": [
                        {
                            "propertyAlias": "${topic()}",
                            "propertyValues": [
                                {
                                    "timestamp": {
                                        "timeInSeconds": "${floor(timestamp() / 1E3)}",
                                        "offsetInNanos": "${(timestamp() % 1E3) * 1E6}"
                                    },
                                    "value": {
                                        "doubleValue": "${my.payload.value}"
                                    }
                                }
                            ]
                        }
                    ],
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_sitewise"
                }
            }
        ]
    }
}
```

See also

- [What is AWS IoT SiteWise?](#) in the [AWS IoT SiteWise User Guide](#)
 - [Ingesting data using AWS IoT Core rules](#) in the [AWS IoT SiteWise User Guide](#)

- Ingesting data to AWS IoT SiteWise from [AWS IoT things](#) in the [AWS IoT SiteWise User Guide](#)
- Troubleshooting an AWS IoT SiteWise rule action in the [AWS IoT SiteWise User Guide](#)

Kinesis Data Firehose

The Kinesis Data Firehose([firehose](#)) action sends data from an MQTT message to an Amazon Kinesis Data Firehose stream.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `firehose:PutRecord` operation. For more information, see [Granting AWS IoT the required access \(p. 444\)](#).
In the AWS IoT console, you can choose or create a role to allow AWS IoT to perform this rule action.
- If you use Kinesis Data Firehose to send data to an Amazon S3 bucket, and you use an AWS Key Management Service (AWS KMS) customer-managed AWS KMS key (KMS key) to encrypt data at rest in Amazon S3, Kinesis Data Firehose must have access to your bucket and permission to use the AWS KMS key on the caller's behalf. For more information, see [Grant Kinesis Data Firehose access to an Amazon S3 destination](#) in the [Amazon Kinesis Data Firehose Developer Guide](#).

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`batchMode`

(Optional) Whether to deliver the Kinesis Data Firehose stream as a batch by using [PutRecordBatch](#). The default value is `false`.

When `batchMode` is `true` and the rule's SQL statement evaluates to an Array, each Array element forms one record in the `PutRecordBatch` request. The resulting array can't have more than 500 records.

Supports [substitution templates \(p. 586\)](#): No

`deliveryStreamName`

The Kinesis Data Firehose stream to which to write the message data.

Supports [substitution templates \(p. 586\)](#): API and AWS CLI only

`separator`

(Optional) A character separator that is used to separate records written to the Kinesis Data Firehose stream. If you omit this parameter, the stream uses no separator. Valid values: `,` (comma), `\t` (tab), `\n` (newline), `\r\n` (Windows newline).

Supports [substitution templates \(p. 586\)](#): No

`roleArn`

The IAM role that allows access to the Kinesis Data Firehose stream. For more information, see [Requirements \(p. 502\)](#).

Supports [substitution templates \(p. 586\)](#): No

Examples

The following JSON example defines a Kinesis Data Firehose action in an AWS IoT rule.

```
{  
    "topicRulePayload": {  
        "sql": "SELECT * FROM 'some/topic'",  
        "ruleDisabled": false,  
        "awsIotSqlVersion": "2016-03-23",  
        "actions": [  
            {  
                "firehose": {  
                    "deliveryStreamName": "my_firehose_stream",  
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_firehose"  
                }  
            }  
        ]  
    }  
}
```

The following JSON example defines a Kinesis Data Firehose action with substitution templates in an AWS IoT rule.

```
{  
    "topicRulePayload": {  
        "sql": "SELECT * FROM 'some/topic'",  
        "ruleDisabled": false,  
        "awsIotSqlVersion": "2016-03-23",  
        "actions": [  
            {  
                "firehose": {  
                    "deliveryStreamName": "${topic()}",  
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_firehose"  
                }  
            }  
        ]  
    }  
}
```

See also

- [What is Amazon Kinesis Data Firehose?](#) in the *Amazon Kinesis Data Firehose Developer Guide*

Kinesis Data Streams

The Kinesis Data Streams (`kinesis`) action writes data from an MQTT message to Amazon Kinesis Data Streams.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `kinesis:PutRecord` operation. For more information, see [Granting AWS IoT the required access \(p. 444\)](#).

In the AWS IoT console, you can choose or create a role to allow AWS IoT to perform this rule action.

- If you use an AWS Key Management Service (AWS KMS) customer-managed AWS KMS key (KMS key) to encrypt data at rest in Kinesis Data Streams, the service must have permission to use the AWS KMS key

on the caller's behalf. For more information, see [Permissions to use user-generated AWS KMS keys](#) in the *Amazon Kinesis Data Streams Developer Guide*.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`stream`

The Kinesis data stream to which to write data.

Supports [substitution templates \(p. 586\)](#): API and AWS CLI only

`partitionKey`

The partition key used to determine to which shard the data is written. The partition key is usually composed of an expression (for example, `#{topic()}` or `#{timestamp()}`).

Supports [substitution templates \(p. 586\)](#): Yes

`roleArn`

The ARN of the IAM role that grants AWS IoT permission to access the Kinesis data stream. For more information, see [Requirements \(p. 503\)](#).

Supports [substitution templates \(p. 586\)](#): No

Examples

The following JSON example defines a Kinesis Data Streams action in an AWS IoT rule.

```
{  
    "topicRulePayload": {  
        "sql": "SELECT * FROM 'some/topic'",  
        "ruleDisabled": false,  
        "awsIotSqlVersion": "2016-03-23",  
        "actions": [  
            {  
                "kinesis": {  
                    "streamName": "my_kinesis_stream",  
                    "partitionKey": " #{topic()}",  
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_kinesis"  
                }  
            }  
        ]  
    }  
}
```

The following JSON example defines a Kinesis action with substitution templates in an AWS IoT rule.

```
{  
    "topicRulePayload": {  
        "sql": "SELECT * FROM 'some/topic'",  
        "ruleDisabled": false,  
        "awsIotSqlVersion": "2016-03-23",  
        "actions": [  
            {  
                "kinesis": {  
                    "streamName": " #{topic()}",  
                    "partitionKey": " #{timestamp()}",  
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_kinesis"  
                }  
            }  
        ]  
    }  
}
```

```
        }
    ]
}
}
```

See also

- [What is Amazon Kinesis Data Streams?](#) in the *Amazon Kinesis Data Streams Developer Guide*

Lambda

A Lambda (`lambda`) action invokes an AWS Lambda function, passing in an MQTT message. AWS IoT invokes Lambda functions asynchronously.

You can follow a tutorial that shows you how to create and test a rule with a Lambda action. For more information, see [Tutorial: Formatting a notification by using an AWS Lambda function \(p. 203\)](#).

Requirements

This rule action has the following requirements:

- For AWS IoT to invoke a Lambda function, you must configure a policy that grants the `lambda:InvokeFunction` permission to AWS IoT. You can only invoke a Lambda function defined in the same AWS Region where your Lambda policy exists. Lambda functions use resource-based policies, so you must attach the policy to the Lambda function itself.

Use the following AWS CLI command to attach a policy that grants the `lambda:InvokeFunction` permission.

```
aws lambda add-permission --function-name function_name --region region --principal iot.amazonaws.com --source-arn arn:aws:iot:region:account-id:rule/rule_name --source-account account_id --statement-id unique_id --action "lambda:InvokeFunction"
```

The `add-permission` command expects the following parameters:

`--function-name`

Name of the Lambda function. You add a new permission to update the function's resource policy.
`--region`

The AWS Region of the function.
`--principal`

The principal that gets the permission. This should be `iot.amazonaws.com` to allow AWS IoT permission to call the Lambda function.
`--source-arn`

The ARN of the rule. You can use the `get-topic-rule` AWS CLI command to get the ARN of a rule.
`--source-account`

The AWS account where the rule is defined.
`--statement-id`

A unique statement identifier.

--action

The Lambda action you want to allow in this statement. To allow AWS IoT to invoke a Lambda function, specify `lambda:InvokeFunction`.

Important

If you add a permission for an AWS IoT principal without providing the `source-arn` or `source-account`, any AWS account that creates a rule with your Lambda action can trigger rules to invoke your Lambda function from AWS IoT.

For more information, see [AWS Lambda permissions](#).

- If you use an AWS Key Management Service (AWS KMS) customer-managed AWS KMS key (KMS key) to encrypt data at rest in Lambda, the service must have permission to use the AWS KMS key on the caller's behalf. For more information, see [Encryption at rest](#) in the *AWS Lambda Developer Guide*.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`functionArn`

The ARN of the Lambda function to invoke. AWS IoT must have permission to invoke the function. For more information, see [Requirements \(p. 505\)](#).

If you don't specify a version or alias for your Lambda function, the most recent version of the function is executed. You can specify a version or alias if you want to execute a specific version of your Lambda function. To specify a version or alias, append the version or alias to the ARN of the Lambda function.

```
arn:aws:lambda:us-east-2:123456789012:function:myLambdaFunction:someAlias
```

For more information about versioning and aliases see [AWS Lambda function versioning and aliases](#).

Supports [substitution templates \(p. 586\)](#): API and AWS CLI only

Examples

The following JSON example defines a Lambda action in an AWS IoT rule.

```
{
    "topicRulePayload": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "lambda": {
                    "functionArn": "arn:aws:lambda:us-
east-2:123456789012:function:myLambdaFunction"
                }
            }
        ]
    }
}
```

The following JSON example defines a Lambda action with substitution templates in an AWS IoT rule.

```
{  
    "topicRulePayload": {  
        "sql": "SELECT * FROM 'some/topic'",  
        "ruleDisabled": false,  
        "awsIotSqlVersion": "2016-03-23",  
        "actions": [  
            {  
                "lambda": {  
                    "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:  
${topic()}"  
                }  
            }  
        ]  
    }  
}
```

See also

- [What is AWS Lambda?](#) in the *AWS Lambda Developer Guide*
- [Tutorial: Formatting a notification by using an AWS Lambda function \(p. 203\)](#)

OpenSearch

The OpenSearch (`openSearch`) action writes data from MQTT messages to an Amazon OpenSearch Service domain. You can then use tools like OpenSearch Dashboards to query and visualize data in OpenSearch Service.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `es:ESHttpPut` operation. For more information, see [Granting AWS IoT the required access \(p. 444\)](#).

In the AWS IoT console, you can choose or create a role to allow AWS IoT to perform this rule action.

- If you use a customer-managed AWS KMS key (KMS key) to encrypt data at rest in OpenSearch Service, the service must have permission to use the KMS key on the caller's behalf. For more information, see [Encryption of data at rest for Amazon OpenSearch Service](#) in the *Amazon OpenSearch Service Developer Guide*.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`endpoint`

The endpoint of your Amazon OpenSearch Service domain.

Supports [substitution templates \(p. 586\)](#): API and AWS CLI only

`index`

The OpenSearch index where you want to store your data.

Supports [substitution templates \(p. 586\)](#): Yes

type

The type of document you are storing.

Supports [substitution templates \(p. 586\)](#): Yes

id

The unique identifier for each document.

Supports [substitution templates \(p. 586\)](#): Yes

roleARN

The IAM role that allows access to the OpenSearch Service domain. For more information, see [Requirements \(p. 507\)](#).

Supports [substitution templates \(p. 586\)](#): No

Examples

The following JSON example defines an OpenSearch action in an AWS IoT rule and how you can specify the fields for the OpenSearch action. For more information, see [OpenSearchAction](#).

```
{  
    "topicRulePayload": {  
        "sql": "SELECT *, timestamp() as timestamp FROM 'iot/test'",  
        "ruleDisabled": false,  
        "awsIotSqlVersion": "2016-03-23",  
        "actions": [  
            {  
                "openSearch": {  
                    "endpoint": "https://my-endpoint",  
                    "index": "my-index",  
                    "type": "my-type",  
                    "id": "${newuuid()}",  
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_os"  
                }  
            }  
        ]  
    }  
}
```

The following JSON example defines an OpenSearch action with substitution templates in an AWS IoT rule.

```
{  
    "topicRulePayload": {  
        "sql": "SELECT * FROM 'some/topic'",  
        "ruleDisabled": false,  
        "awsIotSqlVersion": "2016-03-23",  
        "actions": [  
            {  
                "openSearch": {  
                    "endpoint": "https://my-endpoint",  
                    "index": "${topic()}",  
                    "type": "${type}",  
                    "id": "${newuuid()}",  
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_os"  
                }  
            }  
        ]  
    }  
}
```

}

See also

[What is Amazon OpenSearch Service?](#) in the *Amazon OpenSearch Service Developer Guide*

Republish

The republish (`republish`) action republishes an MQTT message to another MQTT topic.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `iot:Publish` operation. For more information, see [Granting AWS IoT the required access \(p. 444\)](#).

In the AWS IoT console, you can choose or create a role to allow AWS IoT to perform this rule action.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

topic

The MQTT topic to which to republish the message.

To republish to a reserved topic, which begins with \$, use \$\$ instead. For example, to republish to the device shadow topic `$aws/things/MyThing/shadow/update`, specify the topic as `$$aws/things/MyThing/shadow/update`.

Note

Republishing to [reserved job topics \(p. 101\)](#) is not supported.

Supports [substitution templates \(p. 586\)](#): Yes

qos

(Optional) The Quality of Service (QoS) level to use when republishing messages. Valid values: 0, 1. The default value is 0. For more information about MQTT QoS, see [MQTT \(p. 80\)](#).

Supports [substitution templates \(p. 586\)](#): No

roleArn

The IAM role that allows AWS IoT to publish to the MQTT topic. For more information, see [Requirements \(p. 509\)](#).

Supports [substitution templates \(p. 586\)](#): No

Examples

The following JSON example defines a republish action in an AWS IoT rule.

```
{  
    "topicRulePayload": {  
        "sql": "SELECT * FROM 'some/topic'",  
        "ruleDisabled": false,  
    },  
}
```

```
"awsIotSqlVersion": "2016-03-23",
"actions": [
    {
        "republish": {
            "topic": "another/topic",
            "qos": 1,
            "roleArn": "arn:aws:iam::123456789012:role/aws_iot_republish"
        }
    }
]
```

The following JSON example defines a republish action with substitution templates in an AWS IoT rule.

```
{
    "topicRulePayload": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "republish": {
                    "topic": "${topic()}/republish",
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_republish"
                }
            }
        ]
    }
}
```

S3

The S3 (`s3`) action writes the data from an MQTT message to an Amazon Simple Storage Service (Amazon S3) bucket.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `s3:PutObject` operation. For more information, see [Granting AWS IoT the required access \(p. 444\)](#).
In the AWS IoT console, you can choose or create a role to allow AWS IoT to perform this rule action.
- If you use an AWS Key Management Service (AWS KMS) customer-managed AWS KMS key (KMS key) to encrypt data at rest in Amazon S3, the service must have permission to use the AWS KMS key on the caller's behalf. For more information, see [AWS managed AWS KMS keys and customer managed AWS KMS keys](#) in the *Amazon Simple Storage Service Developer Guide*.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`bucket`

The Amazon S3 bucket to which to write data.

Supports [substitution templates \(p. 586\)](#): API and AWS CLI only

cannedacl

(Optional) The Amazon S3 canned ACL that controls access to the object identified by the object key. For more information, including allowed values, see [Canned ACL](#).

Supports [substitution templates \(p. 586\)](#): No
key

The path to the file where the data is written.

Consider an example where this parameter is \${topic()}/\${timestamp()} and the rule receives a message where the topic is some/topic. If the current timestamp is 1460685389, then this action writes the data to a file called 1460685389 in the some/topic folder of the S3 bucket.

Note

If you use a static key, AWS IoT overwrites a single file each time the rule invokes. We recommend that you use the message timestamp or another unique message identifier so that a new file is saved in Amazon S3 for each message received.

Supports [substitution templates \(p. 586\)](#): Yes
roleArn

The IAM role that allows access to the Amazon S3 bucket. For more information, see [Requirements \(p. 510\)](#).

Supports [substitution templates \(p. 586\)](#): No

Examples

The following JSON example defines an S3 action in an AWS IoT rule.

```
{  
    "topicRulePayload": {  
        "sql": "SELECT * FROM 'some/topic'",  
        "ruleDisabled": false,  
        "awsIotSqlVersion": "2016-03-23",  
        "actions": [  
            {  
                "s3": {  
                    "bucketName": "my-bucket",  
                    "cannedacl": "public-read",  
                    "key": "${topic()}/${timestamp()}",  
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_s3"  
                }  
            }  
        ]  
    }  
}
```

See also

- [What is Amazon S3? in the Amazon Simple Storage Service User Guide](#)

Salesforce IoT

The Salesforce IoT (salesforce) action sends data from the MQTT message that triggered the rule to a Salesforce IoT input stream.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`url`

The URL exposed by the Salesforce IoT input stream. The URL is available from the Salesforce IoT platform when you create an input stream. For more information, see the Salesforce IoT documentation.

Supports [substitution templates \(p. 586\)](#): No

`token`

The token used to authenticate access to the specified Salesforce IoT input stream. The token is available from the Salesforce IoT platform when you create an input stream. For more information, see the Salesforce IoT documentation.

Supports [substitution templates \(p. 586\)](#): No

Examples

The following JSON example defines a Salesforce IoT action in an AWS IoT rule.

```
{  
    "topicRulePayload": {  
        "sql": "SELECT * FROM 'some/topic'",  
        "ruleDisabled": false,  
        "awsIotSqlVersion": "2016-03-23",  
        "actions": [  
            {  
                "salesforce": {  
                    "token": "ABCDEFGHI123456789abcdefghi123456789",  
                    "url": "https://ingestion-cluster-id.my-env.sfdcnow.com/streams/stream-  
id/connection-id/my-event"  
                }  
            }  
        ]  
    }  
}
```

SNS

The SNS (`sns`) action sends the data from an MQTT message as an Amazon Simple Notification Service (Amazon SNS) push notification.

You can follow a tutorial that shows you how to create and test a rule with an SNS action. For more information, see [Tutorial: Sending an Amazon SNS notification \(p. 190\)](#).

Note

The SNS action doesn't support [Amazon SNS FIFO \(First-In-First-Out\) topics](#). Because the rules engine is a fully distributed service, there is no guarantee of message order when the SNS action is invoked.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `sns:Publish` operation. For more information, see [Granting AWS IoT the required access \(p. 444\)](#).

In the AWS IoT console, you can choose or create a role to allow AWS IoT to perform this rule action.

- If you use an AWS Key Management Service (AWS KMS) customer-managed AWS KMS key (KMS key) to encrypt data at rest in Amazon SNS, the service must have permission to use the AWS KMS key on the caller's behalf. For more information, see [Key management](#) in the *Amazon Simple Notification Service Developer Guide*.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

targetArn

The SNS topic or individual device to which the push notification is sent.

Supports [substitution templates \(p. 586\)](#): API and AWS CLI only

messageFormat

(Optional) The message format. Amazon SNS uses this setting to determine if the payload should be parsed and if relevant platform-specific parts of the payload should be extracted. Valid values: JSON, RAW. Defaults to RAW.

Supports [substitution templates \(p. 586\)](#): No

roleArn

The IAM role that allows access to SNS. For more information, see [Requirements \(p. 512\)](#).

Supports [substitution templates \(p. 586\)](#): No

Examples

The following JSON example defines an SNS action in an AWS IoT rule.

```
{  
    "topicRulePayload": {  
        "sql": "SELECT * FROM 'some/topic'",  
        "ruleDisabled": false,  
        "awsIotSqlVersion": "2016-03-23",  
        "actions": [  
            {  
                "sns": {  
                    "targetArn": "arn:aws:sns:us-east-2:123456789012:my_sns_topic",  
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot sns"  
                }  
            }  
        ]  
    }  
}
```

The following JSON example defines an SNS action with substitution templates in an AWS IoT rule.

```
{  
    "topicRulePayload": {  
        "sql": "SELECT * FROM 'some/topic'",  
        "ruleDisabled": false,  
        "awsIotSqlVersion": "2016-03-23",  
        "actions": [  
            {  
                "sns": {  
                    "targetArn": "arn:aws:sns:us-east-2:  
                }  
            }  
        ]  
    }  
}
```

```
        "targetArn": "arn:aws:sns:us-east-1:123456789012:${topic()}",
        "messageFormat": "JSON",
        "roleArn": "arn:aws:iam::123456789012:role/aws_iot_sns"
    }
}
}
```

See also

- [What is Amazon Simple Notification Service?](#) in the *Amazon Simple Notification Service Developer Guide*
- [Tutorial: Sending an Amazon SNS notification \(p. 190\)](#)

SQS

The SQS (`sqs`) action sends data from an MQTT message to an Amazon Simple Queue Service (Amazon SQS) queue.

Note

The SQS action doesn't support [Amazon SQS FIFO \(First-In-First-Out\) queues](#). Because the rules engine is a fully distributed service, there is no guarantee of message order when the SQS action is triggered.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `sqs:SendMessage` operation. For more information, see [Granting AWS IoT the required access \(p. 444\)](#).

In the AWS IoT console, you can choose or create a role to allow AWS IoT to perform this rule action.

- If you use an AWS Key Management Service (AWS KMS) customer-managed AWS KMS key (KMS key) to encrypt data at rest in Amazon SQS, the service must have permission to use the AWS KMS key on the caller's behalf. For more information, see [Key management](#) in the *Amazon Simple Queue Service Developer Guide*.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`queueUrl`

The URL of the Amazon SQS queue to which to write the data.

Supports [substitution templates \(p. 586\)](#): API and AWS CLI only

`useBase64`

Set this parameter to `true` to configure the rule action to base64-encode the message data before it writes the data to the Amazon SQS queue. Defaults to `false`.

Supports [substitution templates \(p. 586\)](#): No

`roleArn`

The IAM role that allows access to the Amazon SQS queue. For more information, see [Requirements \(p. 514\)](#).

Supports [substitution templates \(p. 586\)](#): No

Examples

The following JSON example defines an SQS action in an AWS IoT rule.

```
{  
    "topicRulePayload": {  
        "sql": "SELECT * FROM 'some/topic'",  
        "ruleDisabled": false,  
        "awsIotSqlVersion": "2016-03-23",  
        "actions": [  
            {  
                "sq": {  
                    "queueUrl": "https://sqs.us-east-2.amazonaws.com/123456789012/  
my_sqs_queue",  
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_sq"  
                }  
            }  
        ]  
    }  
}
```

The following JSON example defines an SQS action with substitution templates in an AWS IoT rule.

```
{  
    "topicRulePayload": {  
        "sql": "SELECT * FROM 'some/topic'",  
        "ruleDisabled": false,  
        "awsIotSqlVersion": "2016-03-23",  
        "actions": [  
            {  
                "sq": {  
                    "queueUrl": "https://sqs.us-east-2.amazonaws.com/123456789012/  
${topic()}",  
                    "useBase64": true,  
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_sq"  
                }  
            }  
        ]  
    }  
}
```

See also

- [What is Amazon Simple Queue Service? in the *Amazon Simple Queue Service Developer Guide*](#)

Step Functions

The Step Functions (`stepFunctions`) action starts an AWS Step Functions state machine.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `states:StartExecution` operation. For more information, see [Granting AWS IoT the required access \(p. 444\)](#).

In the AWS IoT console, you can choose or create a role to allow AWS IoT to perform this rule action.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`stateMachineName`

The name of the Step Functions state machine to start.

Supports [substitution templates \(p. 586\)](#): API and AWS CLI only

`executionNamePrefix`

(Optional) The name given to the state machine execution consists of this prefix followed by a UUID. Step Functions creates a unique name for each state machine execution if one is not provided.

Supports [substitution templates \(p. 586\)](#): Yes

`roleArn`

The ARN of the role that grants AWS IoT permission to start the state machine. For more information, see [Requirements \(p. 515\)](#).

Supports [substitution templates \(p. 586\)](#): No

Examples

The following JSON example defines a Step Functions action in an AWS IoT rule.

```
{  
    "topicRulePayload": {  
        "sql": "SELECT * FROM 'some/topic'",  
        "ruleDisabled": false,  
        "awsIotSqlVersion": "2016-03-23",  
        "actions": [  
            {  
                "stepFunctions": {  
                    "stateMachineName": "myStateMachine",  
                    "executionNamePrefix": "myExecution",  
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_step_functions"  
                }  
            }  
        ]  
    }  
}
```

See also

- [What is AWS Step Functions? in the AWS Step Functions Developer Guide](#)

Timestream

The Timestream rule action writes attributes (measures) from an MQTT message into an Amazon Timestream table. For more information about Amazon Timestream, see [What Is Amazon Timestream?](#).

Note

Amazon Timestream is not available in all AWS Regions. If Amazon Timestream is not available in your Region, it won't appear in the list of rule actions.

The attributes that this rule stores in the Timestream database are those that result from the rule's query statement. The value of each attribute in the query statement's result is parsed to infer its data type (as in a [the section called "DynamoDBv2" \(p. 465\)](#) action). Each attribute's value is written to its own record in the Timestream table. To specify or change an attribute's data type, use the [cast\(\) \(p. 547\)](#) function in the query statement. For more information about the contents of each Timestream record, see [the section called "Amazon Timestream record content" \(p. 518\)](#).

Note

With SQL V2 (2016-03-23), numeric values that are whole numbers, such as 10 . 0, are converted their Integer representation (10). Explicitly casting them to a Decimal value, such as by using the [cast\(\) \(p. 547\)](#) function, does not prevent this behavior—the result is still an Integer value. This can cause type mismatch errors that prevent data from being recorded in the Timestream database. To reliably process whole number numeric values as Decimal values, use SQL V1 (2015-10-08) for the rule query statement.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `timestream:DescribeEndpoints` and `timestream:WriteRecords` operations. For more information, see [Granting AWS IoT the required access \(p. 444\)](#).

In the AWS IoT console, you can choose, update, or create a role to allow AWS IoT to perform this rule action.

- If you use a customer-managed AWS Key Management Service (AWS KMS) to encrypt data at rest in Timestream, the service must have permission to use the AWS KMS key on the caller's behalf. For more information, see [How AWS services use AWS KMS](#).

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

databaseName

The name of an Amazon Timestream database that has the table to receive the records this action creates. See also `tableName`.

Supports [substitution templates \(p. 586\)](#): API and AWS CLI only

dimensions

Metadata attributes of the time series that are written in each measure record. For example, the name and Availability Zone of an EC2 instance or the name of the manufacturer of a wind turbine are dimensions.

name

The metadata dimension name. This is the name of the column in the database table record.

Dimensions can't be named: `measure_name`, `measure_value`, or `time`. These names are reserved. Dimension names can't start with `ts_` or `measure_value` and they can't contain the colon (:) character.

Supports [substitution templates \(p. 586\)](#): No

value

The value to write in this column of the database record.

Supports [substitution templates \(p. 586\)](#): Yes

roleArn

The Amazon Resource Name (ARN) of the role that grants AWS IoT permission to write to the Timestream database table. For more information, see [Requirements \(p. 517\)](#).

Supports [substitution templates \(p. 586\)](#): No

tableName

The name of the database table into which to write the measure records. See also [databaseName](#).

Supports [substitution templates \(p. 586\)](#): API and AWS CLI only

timestamp

The value to use for the entry's timestamp. If blank, the time that the entry was processed is used.
unit

The precision of the timestamp value that results from the expression described in **value**.

Valid values: SECONDS | MILLISECONDS | MICROSECONDS | NANOSECONDS. The default is MILLISECONDS.

value

An expression that returns a long epoch time value.

You can use the [the section called "time_to_epoch\(String, String\)" \(p. 577\)](#) function to create a valid timestamp from a date or time value passed in the message payload.

Amazon Timestream record content

The data written to the Amazon Timestream table by this action include a timestamp, metadata from the Timestream rule action, and the result of the rule's query statement.

For each attribute (measure) in the result of the query statement, this rule action writes a record to the specified Timestream table with these columns.

Column name	Attribute type	Value	Comments
<i>dimension-name</i>	DIMENSION	The value specified in the Timestream rule action entry.	Each Dimension specified in the rule action entry creates a column in the Timestream database with the dimension's name.
measure_name	MEASURE_NAME	The attribute's name	The name of the attribute in the result of the query statement whose value is specified in the measure_value::datatype column.

Column name	Attribute type	Value	Comments
measure_value:: <i>data-type</i>	MEASURE_VALUE	The value of the attribute in the result of the query statement. The attribute's name is in the measure_name column.	The value is interpreted* and cast as the best match of: bigint, boolean, double, or varchar. Amazon Timestream creates a separate column for each data type. The value in the message can be cast to another data type by using the cast() (p. 547) function in the rule's query statement.
time	TIMESTAMP	The date and time of the record in the database.	This value is assigned by rules engine or the timestamp property, if it is defined.

* The attribute value read from the message payload is interpreted as follows. See the [the section called "Examples" \(p. 519\)](#) for an illustration of each of these cases.

- An unquoted value of true or false is interpreted as a boolean type.
- A decimal numeric is interpreted as a double type.
- A numeric value without a decimal point is interpreted as a bigint type.
- A quoted string is interpreted as a varchar type.
- Objects and array values are converted to JSON strings and stored as a varchar type.

Examples

The following JSON example defines a Timestream rule action with a substitution template in an AWS IoT rule.

```
{
  "topicRulePayload": {
    "sql": "SELECT * FROM 'iot/topic'",
    "ruleDisabled": false,
    "awsIotSqlVersion": "2016-03-23",
    "actions": [
      {
        "timestream": {
          "roleArn": "arn:aws:iam::123456789012:role/aws_iot_timestream",
          "tableName": "devices_metrics",
          "dimensions": [
            {
              "name": "device_id",
              "value": "${clientId()}"
            },
            {
              "name": "device_firmware_sku",
              "value": "My Static Metadata"
            }
          ],
        }
      }
    ]
  }
}
```

```

        "databaseName": "record_devices"
    }
}
}
}
```

Using the Timestream topic rule action defined in the previous example with the following message payload results in the Amazon Timestream records written in the table that follows.

```
{
    "boolean_value": true,
    "integer_value": 123456789012,
    "double_value": 123.456789012,
    "string_value": "String value",
    "boolean_value_as_string": "true",
    "integer_value_as_string": "123456789012",
    "double_value_as_string": "123.456789012",
    "array_of_integers": [23,36,56,72],
    "array of strings": ["red", "green","blue"],
    "complex_value": {
        "simple_element": 42,
        "array_of_integers": [23,36,56,72],
        "array of strings": ["red", "green","blue"]
    }
}
```

The following table displays the database columns and records that using the specified topic rule action to process the previous message payload creates. The `device_firmware_sku` and `device_id` columns are the DIMENSIONS defined in the topic rule action. The Timestream topic rule action creates the `time` column and the `measure_name` and `measure_value::*` columns, which it fills with the values from the result of the topic rule action's query statement.

device_firm	device_id	measure_na	measure_va	measure_va	measure_va	measure_va	time
My Static Metadata	iotconsole-1595EXAMPLE738-0	1595EXAMPLE738-0	{"simple_element":42,"array_of_integers": [23,36,56,72],"array of strings": ["red","green","blue"]}				2020-08-26 22:42:16.423
My Static Metadata	iotconsole-1595EXAMPLE738-0	1595EXAMPLE738-0_string	123456789012		-		2020-08-26 22:42:16.423
My Static Metadata	iotconsole-1595EXAMPLE738-0	1595EXAMPLE738-0	-	-	TRUE		2020-08-26 22:42:16.423
My Static Metadata	iotconsole-1595EXAMPLE738-0	1595EXAMPLE738-0	123456789012	-	-		2020-08-26 22:42:16.423
My Static Metadata	iotconsole-1595EXAMPLE738-0	1595EXAMPLE738-0	String value	-	-		2020-08-26 22:42:16.423
My Static Metadata	iotconsole-1595EXAMPLE738-0	1595EXAMPLE738-0	[23,36,56,72]		-		2020-08-26 22:42:16.423
My Static Metadata	iotconsole-1595EXAMPLE738-0	1595EXAMPLE738-0	["red","green","blue"]		-		2020-08-26 22:42:16.423
My Static Metadata	iotconsole-1595EXAMPLE738-0	1595EXAMPLE738-0	TRUE	-	-		2020-08-26 22:42:16.423

device_firm	device_id	measure_na	measure_va	measure_va	measure_va	measure_va	time
My Static Metadata	iotconsole-150EXAMPLe123	150EXAMPLe123	-	123.456789012			2020-08-26 22:42:16.423
My Static Metadata	iotconsole-150EXAMPLe123	150EXAMPLe123	string	123.45679	-	-	2020-08-26 22:42:16.423

Troubleshooting a rule

If you have an issue with your rules, you should enable CloudWatch Logs. You can analyze your logs to determine whether the issue is authorization or whether, for example, a WHERE clause condition didn't match. For more information, see [Setting Up CloudWatch Logs](#).

Accessing cross-account resources using AWS IoT rules

You can configure AWS IoT rules for cross-account access so that data ingested on MQTT topics of one account can be routed into the AWS services, such as Amazon SQS and Lambda, of another account. The following explains how to set up AWS IoT rules for cross-account data ingestion, from an MQTT topic in one account, to a destination in another account.

Cross-account rules can be configured using [resource-based permissions](#) on the destination resource. Therefore, only destinations that support resource-based permissions can be enabled for the cross-account access with AWS IoT rules. The supported destinations include Amazon SQS, Amazon SNS, Amazon S3, and AWS Lambda.

Prerequisites

- Familiarity with [AWS IoT rules](#)
- An understanding of IAM [users](#), [roles](#), and [resource-based permission](#)
- Having [AWS CLI](#) installed

Cross-account setup for Amazon SQS

Scenario: Account A sends data from an MQTT message to account B's Amazon SQS queue.

AWS account	Account referred to as	Description
1111-1111-1111	Account A	Rule action: <code>sqss:SendMessage</code>
2222-2222-2222	Account B	Amazon SQS queue <ul style="list-style-type: none"> • ARN: <code>arn:aws:sqs:region:2222-2222-2222:ExampleQueue</code> • URL: <code>https://sqss.region.amazonaws.com/2222-2222-2222/ExampleQueue</code>

Do the Account A tasks

Note

To run the following commands, your IAM user should have permissions to `iot:CreateTopicRule` with the rule's Amazon Resource Name (ARN) as a resource, and permissions to `iam:PassRole` action with a resource as the role's ARN.

1. [Configure AWS CLI](#) using account A's IAM user.
2. Create an IAM role that trusts AWS IoT rules engine, and attaches a policy that allows access to account B's Amazon SQS queue. See example commands and policy documents in [Granting AWS IoT the required access](#).
3. To create a rule that is attached to a topic, run the [create-topic-rule command](#).

```
aws iot create-topic-rule --rule-name myRule --topic-rule-payload file://./my-rule.json
```

The following is an example payload file with a rule that inserts all messages sent to the `iot/test` topic into the specified Amazon SQS queue. The SQL statement filters the messages and the role ARN grants AWS IoT permissions to add the message to the Amazon SQS queue.

```
{
  "sql": "SELECT * FROM 'iot/test'",
  "ruleDisabled": false,
  "awsIotSqlVersion": "2016-03-23",
  "actions": [
    {
      "sq": {
        "queueUrl": "https://sqs.region.amazonaws.com/2222-2222-2222/ExampleQueue",
        "roleArn": "arn:aws:iam::1111-1111-1111:role/my-iot-role",
        "useBase64": false
      }
    }
  ]
}
```

For more information about how to define an Amazon SQS action in an AWS IoT rule, see [AWS IoT rule actions - Amazon SQS](#).

Do the Account B tasks

1. [Configure AWS CLI](#) using account B's IAM user.
2. To give permissions for the Amazon SQS queue resource to account A, run the [add-permission command](#).

```
aws sqs add-permission --queue-url https://sqs.region.amazonaws.com/2222-2222-2222/ExampleQueue --label SendMessagesToMyQueue --aws-account-ids 1111-1111-1111 --actions SendMessage
```

Cross-account setup for Amazon SNS

Scenario: Account A sends data from an MQTT message to an Amazon SNS topic of account B.

AWS account	Account referred to as	Description
1111-1111-1111	Account A	Rule action: <code>sns:Publish</code>

AWS account	Account referred to as	Description
2222-2222-2222	Account B	Amazon SNS topic ARN: <i>arn:aws:sns:region:2222-2222-2222:ExampleTopic</i>

Do the Account A tasks

Notes

To run the following commands, your IAM user should have permissions to `iot:CreateTopicRule` with rule ARN as a resource and permissions to the `iam:PassRole` action with a resource as role ARN.

1. [Configure AWS CLI](#) using account A's IAM user.
2. Create an IAM role that trusts AWS IoT rules engine, and attaches a policy that allows access to account B's Amazon SNS topic. For example commands and policy documents, see [Granting AWS IoT the required access](#).
3. To create a rule that is attached to a topic, run the [create-topic-rule command](#).

```
aws iot create-topic-rule --rule-name myRule --topic-rule-payload file://./my-rule.json
```

The following is an example payload file with a rule that inserts all messages sent to the `iot/test` topic into the specified Amazon SNS topic. The SQL statement filters the messages, and the role ARN grants AWS IoT permissions to send the message to the Amazon SNS topic.

```
{
  "sql": "SELECT * FROM 'iot/test'",
  "ruleDisabled": false,
  "awsIotSqlVersion": "2016-03-23",
  "actions": [
    {
      "sns": {
        "targetArn": "arn:aws:sns:region:2222-2222-2222:ExampleTopic",
        "roleArn": "arn:aws:iam::1111-1111-1111:role/my-iot-role"
      }
    }
  ]
}
```

For more information about how to define an Amazon SNS action in an AWS IoT rule, see [AWS IoT rule actions - Amazon SNS](#).

Do the Account B tasks

1. [Configure AWS CLI](#) using account B's IAM user.
2. To give permission on the Amazon SNS topic resource to account A, run the [add-permission command](#).

```
aws sns add-permission --topic-arn arn:aws:sns:region:2222-2222-2222:ExampleTopic --
label Publish-Permission --aws-account-id 1111-1111-1111 --action-name Publish
```

Cross-account setup for Amazon S3

Scenario: Account A sends data from an MQTT message to an Amazon S3 bucket of account B.

AWS account	Account referred to as	Description
1111-1111-1111	Account A	Rule action: <code>s3:PutObject</code>
2222-2222-2222	Account B	Amazon S3 bucket ARN: <code>arn:aws:s3:::ExampleBucket</code>

Do the Account A tasks

Note

To run the following commands, your IAM user should have permissions to `iot:CreateTopicRule` with the rule ARN as a resource and permissions to `iam:PassRole` action with a resource as role ARN.

1. [Configure AWS CLI](#) using account A's IAM user.
2. Create an IAM role that trusts AWS IoT rules engine and attaches a policy that allows access to account B's Amazon S3 bucket. For example commands and policy documents, see [Granting AWS IoT the required access](#).
3. To create a rule that is attached to your target S3 bucket, run the [create-topic-rule command](#).

```
aws iot create-topic-rule --rule-name my-rule --topic-rule-payload file://./my-rule.json
```

The following is an example payload file with a rule that inserts all messages sent to the `iot/test` topic into the specified Amazon S3 bucket. The SQL statement filters the messages, and the role ARN grants AWS IoT permissions to add the message to the Amazon S3 bucket.

```
{
  "sql": "SELECT * FROM 'iot/test'",
  "ruleDisabled": false,
  "awsIotSqlVersion": "2016-03-23",
  "actions": [
    {
      "s3": {
        "bucketName": "ExampleBucket",
        "key": "${topic()}/${timestamp()}",
        "roleArn": "arn:aws:iam::1111-1111-1111:role/my-iot-role"
      }
    }
  ]
}
```

For more information about how to define an Amazon S3 action in an AWS IoT rule, see [AWS IoT rule actions - Amazon S3](#).

Do the Account B tasks

1. [Configure AWS CLI](#) using account B's IAM user.
2. Create a bucket policy that trusts account A's principal.

The following is an example payload file that defines a bucket policy that trusts the principal of another account.

```
{
  "Version": "2012-10-17",
  "Statement": [
```

```
{
  "Sid": "AddCannedAcl",
  "Effect": "Allow",
  "Principal": {
    "AWS": [
      "arn:aws:iam::1111-1111-1111:root"
    ]
  },
  "Action": "s3:PutObject",
  "Resource": "arn:aws:s3:::ExampleBucket/*"
}
}
```

For more information, see [bucket policy examples](#).

- To attach the bucket policy to the specified bucket, run the [put-bucket-policy command](#).

```
aws s3api put-bucket-policy --bucket ExampleBucket --policy file://./my-bucket-
policy.json
```

- To make the cross-account access work, make sure you have the correct **Block all public access** settings. For more information, see [Security Best Practices for Amazon S3](#).

Cross-account setup for AWS Lambda

Scenario: Account A invokes an AWS Lambda function of account B, passing in an MQTT message.

AWS account	Account referred to as	Description
1111-1111-1111	Account A	Rule action: lambda:InvokeFunction
2222-2222-2222	Account B	Lambda function ARN: <i>arn:aws:lambda:region:2222-2222-2222:function:example- function</i>

Do the Account A tasks

Notes

To run the following commands, your IAM user should have permissions to `iot:CreateTopicRule` with rule ARN as a resource, and permissions to `iam:PassRole` action with resource as role ARN.

- [Configure AWS CLI](#) using account A's IAM user.
- Run the [create-topic-rule command](#) to create a rule that defines cross-account access to account B's Lambda function.

```
aws iot create-topic-rule --rule-name my-rule --topic-rule-payload file://./my-
rule.json
```

The following is an example payload file with a rule that inserts all messages sent to the `iot/test` topic into the specified Lambda function. The SQL statement filters the messages and the role ARN grants AWS IoT permission to pass in the data to the Lambda function.

```
{
  "sql": "SELECT * FROM 'iot/test'",
```

```
"ruleDisabled": false,  
"awsIoTSqlVersion": "2016-03-23",  
"actions": [  
  {  
    "lambda": {  
      "functionArn": "arn:aws:lambda:region:2222-2222-2222:function:example-function"  
    }  
  }  
]
```

For more information about how to define an AWS Lambda action in an AWS IoT rule, read [AWS IoT rule actions - Lambda](#).

Do the Account B tasks

1. [Configure AWS CLI](#) using account B's IAM user.
2. Run [Lambda's add-permission command](#) to give AWS IoT rules permission to trigger the Lambda function. To run the following command, your IAM user should have permission to `lambda:AddPermission` action.

```
aws lambda add-permission --function-name example-function --region us-east-1 --principal iot.amazonaws.com --source-arn arn:aws:iot:region:1111-1111-1111:rule/example-rule --source-account 1111-1111-1111 --statement-id "unique_id" --action "lambda:InvokeFunction"
```

Options:

--principal

This field gives permission to AWS IoT (represented by `iot.amazonaws.com`) to call the Lambda function.

--source-arn

This field ensures that only `arn:aws:iot:region:1111-1111-1111:rule/example-rule` in AWS IoT triggers this Lambda function and no other rule in the same or different account can trigger this Lambda function.

--source-account

This field ensures that AWS IoT triggers this Lambda function only on behalf of the `1111-1111-1111` account.

Notes

If you see an error message "The rule could not be found" from your AWS Lambda function's console under [Configuration](#), ignore the error message and proceed to test the connection.

Error handling (error action)

When AWS IoT receives a message from a device, the rules engine checks to see if the message matches a rule. If so, the rule's query statement is evaluated and the rule's actions are triggered, passing the query statement's result.

If a problem occurs when triggering an action, the rules engine triggers an error action, if one is specified for the rule. This might happen when:

- A rule doesn't have permission to access an Amazon S3 bucket.
- A user error causes DynamoDB provisioned throughput to be exceeded.

Error action message format

A single message is generated per rule and message. For example, if two rule actions in the same rule fail, the error action receives one message that contains both errors.

The error action message looks like the following example.

```
{  
    "ruleName": "TestAction",  
    "topic": "testme/action",  
    "cloudwatchTraceId": "7e146a2c-95b5-6caf-98b9-50e3969734c7",  
    "clientId": "iotconsole-1511213971966-0",  
    "base64OriginalPayload": "ewogICJtZXNzYWdIjogIkhlbGxvIHZyb20gQVdTIElvVCBjb25zb2xlIgp9",  
    "failures": [  
        {  
            "failedAction": "S3Action",  
            "failedResource": "us-east-1-s3-verify-user",  
            "errorMessage": "Failed to put S3 object. The error received was The specified bucket does not exist (Service: Amazon S3; Status Code: 404; Error Code: NoSuchBucket; Request ID: 9DF5416B9B47B9AF; S3 Extended Request ID: yMah1cwPhqTH267QLPhTKeVPKJB8B05ndBHzOmWtxLTM6uAvwYYuqieAKyb6qRPTxP1tHXCoR4Y=). Message arrived on: error/action, Action: s3, Bucket: us-east-1-s3-verify-user, Key: \"aaa\". Value of x-amz-id-2: yMah1cwPhqTH267QLPhTKeVPKJB8B05ndBHzOmWtxLTM6uAvwYYuqieAKyb6qRPTxP1tHXCoR4Y="  
        }  
    ]  
}
```

ruleName

The name of the rule that triggered the error action.

topic

The topic on which the original message was received.

cloudwatchTraceId

A unique identity referring to the error logs in CloudWatch.

clientId

The client ID of the message publisher.

base64OriginalPayload

The original message payload Base64-encoded.

failures

failedAction

The name of the action that failed to complete (for example, "S3Action").

failedResource

The name of the resource (for example, the name of an S3 bucket).

errorMessage

The description and explanation of the error.

Error action example

Here is an example of a rule with an added error action. The following rule has an action that writes message data to a DynamoDB table and an error action that writes data to an Amazon S3 bucket:

```
{  
    "sql" : "SELECT * FROM ..."  
    "actions" : [  
        {  
            "dynamoDB" : {  
                "table" : "PoorlyConfiguredTable",  
                "hashKeyField" : "AConstantString",  
                "hashKeyValue" : "AHashKey"}  
            },  
        {  
            "errorAction" : {  
                "s3" : {  
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_s3",  
                    "bucketName" : "message-processing-errors",  
                    "key" : "${replace(topic(), '/', '-') + '-' + timestamp() + '-' + newuuid()}"  
                }  
            }  
        }  
    ]  
}
```

You can use any function or substitution in an error action's SQL statement, except for external functions (for example, `get_thing_shadow`, `aws_lambda`, and `machinelearning_predict`.)

For more information about rules and how to specify an error action, see [Creating an AWS IoT Rule](#).

For more information about using CloudWatch to monitor the success or failure of rules, see [AWS IoT metrics and dimensions \(p. 410\)](#).

Reducing messaging costs with basic ingest

Basic Ingest enables you to securely send device data to the AWS services supported by [AWS IoT rule actions \(p. 450\)](#) without incurring [messaging costs](#). Basic Ingest optimizes data flow by removing the publish/subscribe message broker from the ingestion path, so it's more cost effective.

To use Basic Ingest, you send messages from your devices or applications with topic names that start with `$aws/rules/rule-name` as their first three levels, where *rule-name* is the name of your AWS IoT rule to trigger.

You can use an existing rule with Basic Ingest simply by adding the Basic Ingest prefix (`$aws/rules/rule-name`) to the message topic by which you normally trigger the rule. For example, if you have a rule named `BuildingManager` that is triggered by messages with topics like `Buildings/Building5/Floor2/Room201/Lights` ("sql": "SELECT * FROM 'Buildings/#'"), you can trigger the same rule with Basic Ingest by sending a message with topic `$aws/rules/BuildingManager/Buildings/Building5/Floor2/Room201/Lights`.

Be aware that:

- Your devices and rules cannot subscribe to Basic Ingest reserved topics. For more information, see [Reserved topics \(p. 95\)](#).
- If you need a publish/subscribe broker to distribute messages to multiple subscribers (for example, to deliver messages to other devices and the rules engine), you should continue to use the AWS IoT message broker to handle the message distribution. Just publish your messages on topics other than Basic Ingest topics.

Using basic ingest

Make sure your device or application is using a [policy \(p. 314\)](#) that has publish permissions on `$aws/rules/*`. Or you can specify permission for individual rules with `$aws/rules/rule-name/*` in the policy. Otherwise, your devices and applications can continue to use their existing connections with AWS IoT Core.

When the message reaches the rules engine, there is no difference in execution or error handling between rules triggered from Basic Ingest and those triggered through message broker subscriptions.

You can, of course, create rules for use with Basic Ingest. Keep in mind the following:

- The initial prefix of a Basic Ingest topic (`$aws/rules/rule-name`) isn't available to the [topic\(Decimal\) \(p. 578\)](#) function.
- If you define a rule that is triggered only with Basic Ingest, the `FROM` clause is optional in the `sql` field of the `rule` definition. It's still required if the rule is also triggered by other messages that must be sent through the message broker (for example, because those other messages must be distributed to multiple subscribers). For more information, see [AWS IoT SQL reference \(p. 529\)](#).
- The first three levels of the Basic Ingest topic (`$aws/rules/rule-name`) are not counted toward the eight segment length limit or toward the 256 total character limit for a topic. Otherwise, the same restrictions apply as documented in [AWS IoT Limits](#).
- If a message is received with a Basic Ingest topic that specifies an inactive rule or a rule that doesn't exist, an error log is created in an Amazon CloudWatch log to help you with debugging. For more information, see [Rules engine log entries \(p. 429\)](#). A `RuleNotFound` metric is indicated and you can create alarms on this metric. For more information, see Rule Metrics in [Rule metrics \(p. 411\)](#).
- You can still publish with QoS 1 on Basic Ingest topics. You receive a `PUBACK` after the message is successfully delivered to the rules engine. Receiving a `PUBACK` does not mean that your rule actions were completed successfully. You can configure an error action to handle errors during action execution. See [Error handling \(error action\) \(p. 526\)](#).

AWS IoT SQL reference

In AWS IoT, rules are defined using an SQL-like syntax. SQL statements are composed of three types of clauses:

SELECT

Required. Extracts information from the payload of an incoming message and performs transformations on the information. The messages to use are identified by the [topic filter \(p. 94\)](#) specified in the `FROM` clause.

The `SELECT` clause supports [Data types \(p. 533\)](#), [Operators \(p. 536\)](#), [Functions \(p. 541\)](#), [Literals \(p. 583\)](#), [Case statements \(p. 584\)](#), [JSON extensions \(p. 584\)](#), [Substitution templates \(p. 586\)](#), [Nested object queries \(p. 587\)](#), and [Binary payloads \(p. 588\)](#).

FROM

The MQTT message [topic filter \(p. 94\)](#) that identifies the messages to extract data from. The rule is triggered for each message sent to an MQTT topic that matches the topic filter specified here. Required for rules that are triggered by messages that pass through the message broker. Optional for rules that are only triggered using the [Basic Ingest \(p. 528\)](#) feature.

WHERE

(Optional) Adds conditional logic that determines whether the actions specified by a rule are carried out.

The WHERE clause supports [Data types \(p. 533\)](#), [Operators \(p. 536\)](#), [Functions \(p. 541\)](#), [Literals \(p. 583\)](#), [Case statements \(p. 584\)](#), [JSON extensions \(p. 584\)](#), [Substitution templates \(p. 586\)](#), and [Nested object queries \(p. 587\)](#).

An example SQL statement looks like this:

```
SELECT color AS rgb FROM 'topic/subtopic' WHERE temperature > 50
```

An example MQTT message (also called an incoming payload) looks like this:

```
{  
    "color": "red",  
    "temperature": 100  
}
```

If this message is published on the 'topic/subtopic' topic, the rule is triggered and the SQL statement is evaluated. The SQL statement extracts the value of the `color` property if the `"temperature"` property is greater than 50. The WHERE clause specifies the condition `temperature > 50`. The AS keyword renames the `"color"` property to `"rgb"`. The result (also called an *outgoing payload*) looks like this:

```
{  
    "rgb": "red"  
}
```

This data is then forwarded to the rule's action, which sends the data for more processing. For more information about rule actions, see [AWS IoT rule actions \(p. 450\)](#).

Note

Comments are not currently supported in AWS IoT SQL syntax.

SELECT clause

The AWS IoT SELECT clause is essentially the same as the ANSI SQL SELECT clause, with some minor differences.

The SELECT clause supports [Data types \(p. 533\)](#), [Operators \(p. 536\)](#), [Functions \(p. 541\)](#), [Literals \(p. 583\)](#), [Case statements \(p. 584\)](#), [JSON extensions \(p. 584\)](#), [Substitution templates \(p. 586\)](#), [Nested object queries \(p. 587\)](#), and [Binary payloads \(p. 588\)](#).

You can use the SELECT clause to extract information from incoming MQTT messages. You can also use `SELECT *` to retrieve the entire incoming message payload. For example:

```
Incoming payload published on topic 'topic/subtopic': {"color": "red", "temperature": 50}  
SQL statement: SELECT * FROM 'topic/subtopic'  
Outgoing payload: {"color": "red", "temperature": 50}
```

If the payload is a JSON object, you can reference keys in the object. Your outgoing payload contains the key-value pair. For example:

```
Incoming payload published on topic 'topic/subtopic': {"color": "red", "temperature": 50}  
SQL statement: SELECT color FROM 'topic/subtopic'  
Outgoing payload: {"color": "red"}
```

You can use the AS keyword to rename keys. For example:

```
Incoming payload published on topic 'topic/subtopic': {"color": "red", "temperature": 50}
SQL: SELECT color AS my_color FROM 'topic/subtopic'
Outgoing payload: {"my_color": "red"}
```

You can select multiple items by separating them with a comma. For example:

```
Incoming payload published on topic 'topic/subtopic': {"color": "red", "temperature": 50}
SQL: SELECT color as my_color, temperature as fahrenheit FROM 'topic/subtopic'
Outgoing payload: {"my_color": "red", "fahrenheit": 50}
```

You can select multiple items including '*' to add items to the incoming payload. For example:

```
Incoming payload published on topic 'topic/subtopic': {"color": "red", "temperature": 50}
SQL: SELECT *, 15 as speed FROM 'topic/subtopic'
Outgoing payload: {"color": "red", "temperature": 50, "speed": 15}
```

You can use the "VALUE" keyword to produce outgoing payloads that are not JSON objects. With SQL version 2015-10-08, you can select only one item. With SQL version 2016-03-23 or later, you can also select an array to output as a top-level object.

Example

```
Incoming payload published on topic 'topic/subtopic': {"color": "red", "temperature": 50}
SQL: SELECT VALUE color FROM 'topic/subtopic'
Outgoing payload: "red"
```

You can use '.' syntax to drill into nested JSON objects in the incoming payload. For example:

```
Incoming payload published on topic 'topic/subtopic': {"color": {"red": 255, "green": 0, "blue": 0}, "temperature": 50}
SQL: SELECT color.red as red_value FROM 'topic/subtopic'
Outgoing payload: {"red_value": 255}
```

For information about how to use JSON object and property names that include reserved characters, such as numbers or the hyphen (minus) character, see [JSON extensions \(p. 584\)](#)

You can use functions (see [Functions \(p. 541\)](#)) to transform the incoming payload. You can use parentheses for grouping. For example:

```
Incoming payload published on topic 'topic/subtopic': {"color": "red", "temperature": 50}
SQL: SELECT (temperature - 32) * 5 / 9 AS celsius, upper(color) as my_color FROM 'topic/
subtopic'
Outgoing payload: {"celsius": 10, "my_color": "RED"}
```

FROM clause

The FROM clause subscribes your rule to a [topic \(p. 93\)](#) or [topic filter \(p. 94\)](#). You must enclose the topic or topic filter in single quotes (''). The rule is triggered for each message sent to an MQTT topic that matches the topic filter specified here. A topic filter allows you to subscribe to a group of similar topics.

Example:

Incoming payload published on topic 'topic/subtopic': {temperature: 50}

Incoming payload published on topic 'topic/subtopic-2': {temperature: 50}

SQL: "SELECT temperature AS t FROM 'topic/subtopic'".

The rule is subscribed to 'topic/subtopic', so the incoming payload is passed to the rule. The outgoing payload, passed to the rule actions, is: {t: 50}. The rule is not subscribed to 'topic/subtopic-2', so the rule is not triggered for the message published on 'topic/subtopic-2'.

Wildcard Example:

You can use the '#' (multi-level) wildcard character to match one or more particular path elements:

Incoming payload published on topic 'topic/subtopic':{temperature: 50}.

Incoming payload published on topic 'topic/subtopic-2':{temperature: 60}.

Incoming payload published on topic 'topic/subtopic-3/details':{temperature: 70}.

Incoming payload published on topic 'topic-2/subtopic-x':{temperature: 80}.

SQL: "SELECT temperature AS t FROM 'topic/#'".

The rule is subscribed to any topic that begins with 'topic', so it is executed three times, sending outgoing payloads of {t: 50} (for topic/subtopic), {t: 60} (for topic/subtopic-2), and {t: 70} (for topic/subtopic-3/details) to its actions. It is not subscribed to 'topic-2/subtopic-x', so the rule is not triggered for the {temperature: 80} message.

+ Wildcard Example:

You can use the '+' (single-level) wildcard character to match any one particular path element:

Incoming payload published on topic 'topic/subtopic':{temperature: 50}.

Incoming payload published on topic 'topic/subtopic-2':{temperature: 60}.

Incoming payload published on topic 'topic/subtopic-3/details':{temperature: 70}.

Incoming payload published on topic 'topic-2/subtopic-x':{temperature: 80}.

SQL: "SELECT temperature AS t FROM 'topic/+'".

The rule is subscribed to all topics with two path elements where the first element is 'topic'. The rule is executed for the messages sent to 'topic/subtopic' and 'topic/subtopic-2', but not 'topic/subtopic-3/details' (it has more levels than the topic filter) or 'topic-2/subtopic-x' (it does not start with topic).

WHERE clause

The WHERE clause determines if the actions specified by a rule are carried out. If the WHERE clause evaluates to true, the rule actions are performed. Otherwise, the rule actions are not performed.

The WHERE clause supports [Data types \(p. 533\)](#), [Operators \(p. 536\)](#), [Functions \(p. 541\)](#), [Literals \(p. 583\)](#), [Case statements \(p. 584\)](#), [JSON extensions \(p. 584\)](#), [Substitution templates \(p. 586\)](#), and [Nested object queries \(p. 587\)](#).

Example:

Incoming payload published on topic/subtopic: {"color": "red", "temperature": 40}.

SQL: SELECT color AS my_color FROM 'topic/subtopic' WHERE temperature > 50 AND color <> 'red'.

In this case, the rule would be triggered, but the actions specified by the rule would not be performed. There would be no outgoing payload.

You can use functions and operators in the WHERE clause. However, you cannot reference any aliases created with the AS keyword in the SELECT. The WHERE clause is evaluated first, to determine if SELECT is evaluated.

Data types

The AWS IoT rules engine supports all JSON data types.

Supported data types

Type	Meaning
Int	A discrete Int. 34 digits maximum.
Decimal	A Decimal with a precision of 34 digits, with a minimum non-zero magnitude of 1E-999 and a maximum magnitude 9.999...E999. Note Some functions return Decimal values with double precision rather than 34-digit precision. With SQL V2 (2016-03-23), numeric values that are whole numbers, such as 10.0, are processed as an Int value (10) instead of the expected Decimal value (10.0). To reliably process whole number numeric values as Decimal values, use SQL V1 (2015-10-08) for the rule query statement.
Boolean	True or False.
String	A UTF-8 string.
Array	A series of values that don't have to have the same type.
Object	A JSON value consisting of a key and a value. Keys must be strings. Values can be any type.
Null	Null as defined by JSON. It's an actual value that represents the absence of a value. You can explicitly create a Null value by using the Null keyword in your SQL statement. For example: "SELECT NULL AS n FROM 'topic/subtopic'"
Undefined	Not a value. This isn't explicitly representable in JSON except by omitting the value. For example, in the object { "foo": null}, the key "foo" returns NULL, but the key "bar" returns Undefined. Internally, the SQL language treats Undefined as a value, but it isn't representable in JSON, so when serialized to JSON, the results are Undefined. <pre>{"foo":null, "bar":undefined}</pre> <p>is serialized to JSON as:</p> <pre>{"foo":null}</pre>

Type	Meaning
	Similarly, <code>Undefined</code> is converted to an empty string when serialized by itself. Functions called with invalid arguments (for example, wrong types, wrong number of arguments, and so on) return <code>Undefined</code> .

Conversions

The following table lists the results when a value of one type is converted to another type (when a value of the incorrect type is given to a function). For example, if the absolute value function "abs" (which expects an `Int` or `Decimal`) is given a `String`, it attempts to convert the `String` to a `Decimal`, following these rules. In this case, `'abs("-5.123")'` is treated as `'abs(-5.123)'`.

Note

There are no attempted conversions to `Array`, `Object`, `Null`, or `Undefined`.

To decimal

Argument type	Result
<code>Int</code>	A <code>Decimal</code> with no decimal point.
<code>Decimal</code>	The source value.
<code>Boolean</code>	<code>Undefined</code> . (You can explicitly use the <code>cast</code> function to transform <code>true</code> = <code>1.0</code> , <code>false</code> = <code>0.0</code> .)
<code>String</code>	The SQL engine tries to parse the string as a <code>Decimal</code> . AWS IoT attempts to parse strings matching the regular expression: <code>^-?\d+(\.\d+)?((?i)E-?\d+)?\$. "0", "-1.2", "5E-12"</code> are all examples of strings that are converted automatically to <code>Decimals</code> .
<code>Array</code>	<code>Undefined</code> .
<code>Object</code>	<code>Undefined</code> .
<code>Null</code>	<code>Null</code> .
<code>Undefined</code>	<code>Undefined</code> .

To int

Argument type	Result
<code>Int</code>	The source value.
<code>Decimal</code>	The source value rounded to the nearest <code>Int</code> .
<code>Boolean</code>	<code>Undefined</code> . (You can explicitly use the <code>cast</code> function to transform <code>true</code> = <code>1.0</code> , <code>false</code> = <code>0.0</code> .)
<code>String</code>	The SQL engine tries to parse the string as a <code>Decimal</code> . AWS IoT attempts to parse strings matching the regular expression: <code>^-?\d+(\.\d+)?((?i)E-?\d+)?\$. "0", "-1.2", "5E-12"</code>

Argument type	Result
	are all examples of strings that are converted automatically to Decimals. AWS IoT attempts to convert the String to a Decimal, and then truncates the decimal places of that Decimal to make an Int.
Array	Undefined.
Object	Undefined.
Null	Null.
Undefined	Undefined.

To Boolean

Argument type	Result
Int	Undefined. (You can explicitly use the cast function to transform 0 = False, any_nonzero_value = True.)
Decimal	Undefined. (You can explicitly use the cast function to transform 0 = False, any_nonzero_value = True.)
Boolean	The original value.
String	"true"=True and "false"=False (case insensitive). Other string values are Undefined.
Array	Undefined.
Object	Undefined.
Null	Undefined.
Undefined	Undefined.

To string

Argument type	Result
Int	A string representation of the Int in standard notation.
Decimal	A string representing the Decimal value, possibly in scientific notation.
Boolean	"true" or "false". All lowercase.
String	The original value.
Array	The Array serialized to JSON. The resultant string is a comma-separated list, enclosed in square brackets. A String is quoted. A Decimal, Int, Boolean, and Null is not.

Argument type	Result
Object	The object serialized to JSON. The resultant string is a comma-separated list of key-value pairs and begins and ends with curly braces. A String is quoted. A Decimal, Int, Boolean, and Null is not.
Null	Undefined.
Undefined	Undefined.

Operators

The following operators can be used in SELECT and WHERE clauses.

AND operator

Returns a Boolean result. Performs a logical AND operation. Returns true if left and right operands are true. Otherwise, returns false. Boolean operands or case insensitive "true" or "false" string operands are required.

Syntax: `expression AND expression`.

AND operator

Left operand	Right operand	Output
Boolean	Boolean	Boolean. True if both operands are true. Otherwise, false.
String/Boolean	String/Boolean	If all strings are "true" or "false" (case insensitive), they are converted to Boolean and processed normally as <code>boolean AND boolean</code> .
Other value	Other value	Undefined.

OR operator

Returns a Boolean result. Performs a logical OR operation. Returns true if either the left or the right operands are true. Otherwise, returns false. Boolean operands or case insensitive "true" or "false" string operands are required.

Syntax: `expression OR expression`.

OR operator

Left operand	Right operand	Output
Boolean	Boolean	Boolean. True if either operand is true. Otherwise, false.
String/Boolean	String/Boolean	If all strings are "true" or "false" (case insensitive), they are converted to Booleans and processed normally as <code>boolean OR boolean</code> .
Other value	Other value	Undefined.

NOT operator

Returns a Boolean result. Performs a logical NOT operation. Returns true if the operand is false. Otherwise, returns true. A Boolean operand or case insensitive "true" or "false" string operand is required.

Syntax: NOT *expression*.

NOT operator

Operand	Output
Boolean	Boolean. True if operand is false. Otherwise, true.
String	If string is "true" or "false" (case insensitive), it is converted to the corresponding boolean value, and the opposite value is returned.
Other value	Undefined.

> operator

Returns a Boolean result. Returns true if the left operand is greater than the right operand. Both operands are converted to a Decimal, and then compared.

Syntax: *expression* > *expression*.

> operator

Left operand	Right operand	Output
Int/Decimal	Int/Decimal	Boolean. True if the left operand is greater than the right operand. Otherwise, false.
String/Int/ Decimal	String/Int/ Decimal	If all strings can be converted to Decimal, then Boolean. Returns true if the left operand is greater than the right operand. Otherwise, false.
Other value	Undefined.	Undefined.

>= operator

Returns a Boolean result. Returns true if the left operand is greater than or equal to the right operand. Both operands are converted to a Decimal, and then compared.

Syntax: *expression* >= *expression*.

>= operator

Left operand	Right operand	Output
Int/Decimal	Int/Decimal	Boolean. True if the left operand is greater than or equal to the right operand. Otherwise, false.
String/Int/ Decimal	String/Int/ Decimal	If all strings can be converted to Decimal, then Boolean. Returns true if the left operand is greater than or equal to the right operand. Otherwise, false.

Left operand	Right operand	Output
Other value	Undefined.	Undefined.

< operator

Returns a Boolean result. Returns true if the left operand is less than the right operand. Both operands are converted to a Decimal, and then compared.

Syntax: `expression < expression`.

< operator

Left operand	Right operand	Output
Int/Decimal	Int/Decimal	Boolean. True if the left operand is less than the right operand. Otherwise, false.
String/Int/ Decimal	String/Int/ Decimal	If all strings can be converted to Decimal, then Boolean. Returns true if the left operand is less than the right operand. Otherwise, false.
Other value	Undefined	Undefined

<= operator

Returns a Boolean result. Returns true if the left operand is less than or equal to the right operand. Both operands are converted to a Decimal, and then compared.

Syntax: `expression <= expression`.

<= operator

Left operand	Right operand	Output
Int/Decimal	Int/Decimal	Boolean. True if the left operand is less than or equal to the right operand. Otherwise, false.
String/Int/ Decimal	String/Int/ Decimal	If all strings can be converted to Decimal, then Boolean. Returns true if the left operand is less than or equal to the right operand. Otherwise, false.
Other value	Undefined	Undefined

<> operator

Returns a Boolean result. Returns true if both left and right operands are not equal. Otherwise, returns false.

Syntax: `expression <> expression`.

<> operator

Left operand	Right operand	Output
Int	Int	True if left operand is not equal to right operand. Otherwise, false.
Decimal	Decimal	True if left operand is not equal to right operand. Otherwise, false. Int is converted to Decimal before being compared.
String	String	True if left operand is not equal to right operand. Otherwise, false.
Array	Array	True if the items in each operand are not equal and not in the same order. Otherwise, false
Object	Object	True if the keys and values of each operand are not equal. Otherwise, false. The order of keys/values is unimportant.
Null	Null	False.
Any value	Undefined	Undefined.
Undefined	Any value	Undefined.
Mismatched type	Mismatched type	True.

= operator

Returns a Boolean result. Returns true if both left and right operands are equal. Otherwise, returns false.

Syntax: `expression = expression`.

= operator

Left operand	Right operand	Output
Int	Int	True if left operand is equal to right operand. Otherwise, false.
Decimal	Decimal	True if left operand is equal to right operand. Otherwise, false. Int is converted to Decimal before being compared.
String	String	True if left operand is equal to right operand. Otherwise, false.
Array	Array	True if the items in each operand are equal and in the same order. Otherwise, false.
Object	Object	True if the keys and values of each operand are equal. Otherwise, false. The order of keys/values is unimportant.
Any value	Undefined	Undefined.
Undefined	Any value	Undefined.
Mismatched type	Mismatched type	False.

+ operator

The "+" is an overloaded operator. It can be used for string concatenation or addition.

Syntax: `expression + expression`.

+ operator

Left operand	Right operand	Output
String	Any value	Converts the right operand to a string and concatenates it to the end of the left operand.
Any value	String	Converts the left operand to a string and concatenates the right operand to the end of the converted left operand.
Int	Int	Int value. Adds operands together.
Int/Decimal	Int/Decimal	Decimal value. Adds operands together.
Other value	Other value	Undefined.

- operator

Subtracts the right operand from the left operand.

Syntax: `expression - expression`.

- operator

Left operand	Right operand	Output
Int	Int	Int value. Subtracts right operand from left operand.
Int/Decimal	Int/Decimal	Decimal value. Subtracts right operand from left operand.
String/Int/ Decimal	String/Int/ Decimal	If all strings convert to decimals correctly, a Decimal value is returned. Subtracts right operand from left operand. Otherwise, returns Undefined.
Other value	Other value	Undefined.
Other value	Other value	Undefined.

* operator

Multiplies the left operand by the right operand.

Syntax: `expression * expression`.

* operator

Left operand	Right operand	Output
Int	Int	Int value. Multiplies the left operand by the right operand.
Int/Decimal	Int/Decimal	Decimal value. Multiplies the left operand by the right operand.

Left operand	Right operand	Output
String/Int/ Decimal	String/Int/ Decimal	If all strings convert to decimals correctly, a Decimal value is returned. Multiplies the left operand by the right operand. Otherwise, returns Undefined.
Other value	Other value	Undefined.

/ operator

Divides the left operand by the right operand.

Syntax: *expression* / *expression*.

/ operator

Left operand	Right operand	Output
Int	Int	Int value. Divides the left operand by the right operand.
Int/Decimal	Int/Decimal	Decimal value. Divides the left operand by the right operand.
String/Int/ Decimal	String/Int/ Decimal	If all strings convert to decimals correctly, a Decimal value is returned. Divides the left operand by the right operand. Otherwise, returns Undefined.
Other value	Other value	Undefined.

% operator

Returns the remainder from dividing the left operand by the right operand.

Syntax: *expression* % *expression*.

% operator

Left operand	Right operand	Output
Int	Int	Int value. Returns the remainder from dividing the left operand by the right operand.
String/Int/ Decimal	String/Int/ Decimal	If all strings convert to decimals correctly, a Decimal value is returned. Returns the remainder from dividing the left operand by the right operand. Otherwise, Undefined.
Other value	Other value	Undefined.

Functions

You can use the following built-in functions in the SELECT or WHERE clauses of your SQL expressions.

abs(Decimal)

Returns the absolute value of a number. Supported by SQL version 2015-10-08 and later.

Example: `abs(-5)` returns 5.

Argument type	Result
Int	Int, the absolute value of the argument.
Decimal	Decimal, the absolute value of the argument.
Boolean	Undefined.
String	Decimal. The result is the absolute value of the argument. If the string cannot be converted, the result is Undefined.
Array	Undefined.
Object	Undefined.
Null	Undefined.
Undefined	Undefined.

accountid()

Returns the ID of the account that owns this rule as a String. Supported by SQL version 2015-10-08 and later.

Example:

```
accountid() = "123456789012"
```

acos(Decimal)

Returns the inverse cosine of a number in radians. Decimal arguments are rounded to double precision before function application. Supported by SQL version 2015-10-08 and later.

Example: `acos(0) = 1.5707963267948966`

Argument type	Result
Int	Decimal (with double precision), the inverse cosine of the argument. Imaginary results are returned as Undefined.
Decimal	Decimal (with double precision), the inverse cosine of the argument. Imaginary results are returned as Undefined.
Boolean	Undefined.
String	Decimal, the inverse cosine of the argument. If the string cannot be converted, the result is Undefined. Imaginary results are returned as Undefined.
Array	Undefined.
Object	Undefined.
Null	Undefined.

Argument type	Result
Undefined	Undefined.

asin(Decimal)

Returns the inverse sine of a number in radians. Decimal arguments are rounded to double precision before function application. Supported by SQL version 2015-10-08 and later.

Example: `asin(0) = 0.0`

Argument type	Result
Int	Decimal (with double precision), the inverse sine of the argument. Imaginary results are returned as Undefined.
Decimal	Decimal (with double precision), the inverse sine of the argument. Imaginary results are returned as Undefined.
Boolean	Undefined.
String	Decimal (with double precision), the inverse sine of the argument. If the string cannot be converted, the result is Undefined. Imaginary results are returned as Undefined.
Array	Undefined.
Object	Undefined.
Null	Undefined.
Undefined	Undefined.

atan(Decimal)

Returns the inverse tangent of a number in radians. Decimal arguments are rounded to double precision before function application. Supported by SQL version 2015-10-08 and later.

Example: `atan(0) = 0.0`

Argument type	Result
Int	Decimal (with double precision), the inverse tangent of the argument. Imaginary results are returned as Undefined.
Decimal	Decimal (with double precision), the inverse tangent of the argument. Imaginary results are returned as Undefined.
Boolean	Undefined.
String	Decimal, the inverse tangent of the argument. If the string cannot be converted, the result is Undefined. Imaginary results are returned as Undefined.

Argument type	Result
Array	Undefined.
Object	Undefined.
Null	Undefined.
Undefined	Undefined.

atan2(Decimal, Decimal)

Returns the angle, in radians, between the positive x-axis and the (x, y) point defined in the two arguments. The angle is positive for counter-clockwise angles (upper half-plane, $y > 0$), and negative for clockwise angles (lower half-plane, $y < 0$). Decimal arguments are rounded to double precision before function application. Supported by SQL version 2015-10-08 and later.

Example: `atan2(1, 0) = 1.5707963267948966`

Argument type	Argument type	Result
Int/Decimal	Int/Decimal	Decimal (with double precision)
Int/Decimal/String	Int/Decimal/String	Decimal, the inverse string cannot be converted to a number
Other value	Other value	Undefined.

aws_lambda(functionArn, inputJson)

Calls the specified Lambda function passing `inputJson` to the Lambda function and returns the JSON generated by the Lambda function.

Arguments

Argument	Description
<code>functionArn</code>	The ARN of the Lambda function to call. The Lambda function must return JSON data.
<code>inputJson</code>	The JSON input passed to the Lambda function. To pass nested object queries and literals, you must use SQL version 2016-03-23.

You must grant AWS IoT `lambda:InvokeFunction` permissions to invoke the specified Lambda function. The following example shows how to grant the `lambda:InvokeFunction` permission using the AWS CLI:

```
aws lambda add-permission --function-name "function_name"
--region "region"
--principal iot.amazonaws.com
--source-arn arn:aws:iot:us-east-1:account_id:rule/rule_name
--source-account "account_id"
--statement-id "unique_id"
--action "lambda:InvokeFunction"
```

The following are the arguments for the **add-permission** command:

--function-name

Name of the Lambda function. You add a new permission to update the function's resource policy.

--region

The AWS Region of your account.

--principal

The principal who is getting the permission. This should be `iot.amazonaws.com` to allow AWS IoT permission to call a Lambda function.

--source-arn

The ARN of the rule. You can use the **get-topic-rule** AWS CLI command to get the ARN of a rule.

--source-account

The AWS account where the rule is defined.

--statement-id

A unique statement identifier.

--action

The Lambda action that you want to allow in this statement. To allow AWS IoT to invoke a Lambda function, specify `lambda:InvokeFunction`.

Important

If you add a permission for an AWS IoT principal without providing the `source-arn` or `source-account`, any AWS account that creates a rule with your Lambda action can trigger rules to invoke your Lambda function from AWS IoT. For more information, see [Lambda Permission Model](#).

Given a JSON message payload like:

```
{  
    "attribute1": 21,  
    "attribute2": "value"  
}
```

The `aws_lambda` function can be used to call Lambda function as follows.

```
SELECT  
aws_lambda("arn:aws:lambda:us-east-1:account_id:function:lambda_function",  
{"payload":attribute1}) as output FROM 'topic-filter'
```

If you want to pass the full MQTT message payload, you can specify the JSON payload using `*`, such as the following example.

```
SELECT  
aws_lambda("arn:aws:lambda:us-east-1:account_id:function:lambda_function", *) as output  
FROM 'topic-filter'
```

`payload.inner.element` selects data from message published on topic 'topic/subtopic'.

`some.value` selects data from the output that is generated by the Lambda function.

Note

The rules engine limits the execution duration of Lambda functions. Lambda function calls from rules should be completed within 2000 milliseconds.

bitand(Int, Int)

Performs a bitwise AND on the bit representations of the two `Int`(-converted) arguments. Supported by SQL version 2015-10-08 and later.

Example: `bitand(13, 5) = 5`

Argument type	Argument type	Result
<code>Int</code>	<code>Int</code>	<code>Int</code> , a bitwise AND of the two arguments.
<code>Int/Decimal</code>	<code>Int/Decimal</code>	<code>Int</code> , a bitwise AND of the two arguments. If one or both numbers are rounded, the result is <code>Undefined</code> .
<code>Int/Decimal/String</code>	<code>Int/Decimal/String</code>	<code>Int</code> , a bitwise AND of the two arguments. If one or both numbers are converted to decimal, the result is <code>Undefined</code> .
Other value	Other value	<code>Undefined</code> .

bitor(Int, Int)

Performs a bitwise OR of the bit representations of the two arguments. Supported by SQL version 2015-10-08 and later.

Example: `bitor(8, 5) = 13`

Argument type	Argument type	Result
<code>Int</code>	<code>Int</code>	<code>Int</code> , the bitwise OR of the two arguments.
<code>Int/Decimal</code>	<code>Int/Decimal</code>	<code>Int</code> , the bitwise OR of the two arguments. If one or both numbers are rounded, the result is <code>Undefined</code> .
<code>Int/Decimal/String</code>	<code>Int/Decimal/String</code>	<code>Int</code> , the bitwise OR of the two arguments. If one or both numbers are converted to decimal, the result is <code>Undefined</code> .
Other value	Other value	<code>Undefined</code> .

bitxor(Int, Int)

Performs a bitwise XOR on the bit representations of the two `Int`(-converted) arguments. Supported by SQL version 2015-10-08 and later.

Example: `bitxor(13, 5) = 8`

Argument type	Argument type	Result
Int	Int	Int, a bitwise XOR or
Int/Decimal	Int/Decimal	Int, a bitwise XOR or numbers are rounded
Int/Decimal/String	Int/Decimal/String	Int, a bitwise XOR or converted to decimal Int. If any conversion fails, the result is Undefined.
Other value	Other value	Undefined.

bitnot(Int)

Performs a bitwise NOT on the bit representations of the Int(-converted) argument. Supported by SQL version 2015-10-08 and later.

Example: `bitnot(13) = 2`

Argument type	Result
Int	Int, a bitwise NOT of the argument.
Decimal	Int, a bitwise NOT of the argument. The Decimal value is rounded down to the nearest Int.
String	Int, a bitwise NOT of the argument. Strings are converted to decimals and rounded down to the nearest Int. If any conversion fails, the result is Undefined.
Other value	Other value.

cast()

Converts a value from one data type to another. Cast behaves mostly like the standard conversions, with the addition of the ability to cast numbers to or from Booleans. If AWS IoT cannot determine how to cast one type to another, the result is Undefined. Supported by SQL version 2015-10-08 and later. Format: `cast(value as type)`.

Example:

```
cast(true as Int) = 1
```

The following keywords might appear after "as" when calling cast:

For SQL version 2015-10-08 and 2016-03-23

Keyword	Result
String	Casts value to String.
Nvarchar	Casts value to String.
Text	Casts value to String.
Ntext	Casts value to String.

Keyword	Result
varchar	Casts value to String.
Int	Casts value to Int.
Integer	Casts value to Int.
Double	Casts value to Decimal (with double precision).

Additionally, for SQL version 2016-03-23

Keyword	Result
Decimal	Casts value to Decimal.
Bool	Casts value to Boolean.
Boolean	Casts value to Boolean.

Casting rules:

Cast to decimal

Argument type	Result
Int	A Decimal with no decimal point.
Decimal	The source value. Note With SQL V2 (2016-03-23), numeric values that are whole numbers, such as 10.0, return an Int value (10) instead of the expected Decimal value (10.0). To reliably cast whole number numeric values as Decimal values, use SQL V1 (2015-10-08) for the rule query statement.
Boolean	true = 1.0, false = 0.0.
String	Tries to parse the string as a Decimal. AWS IoT attempts to parse strings matching the regex: ^-?\d+(\.\d+)?((?i)E-?\d+)?\$."0", "-1.2", "5E-12" are all examples of strings that are converted automatically to decimals.
Array	Undefined.
Object	Undefined.
Null	Undefined.
Undefined	Undefined.

Cast to int

Argument type	Result
Int	The source value.

Argument type	Result
Decimal	The source value, rounded down to the nearest Int.
Boolean	true = 1.0, false = 0.0.
String	Tries to parse the string as a Decimal. AWS IoT attempts to parse strings matching the regex: ^-?\d+(\.\d+)?((? <i>i</i>)E-?\d+)?\$". "0", "-1.2", "5E-12" are all examples of strings that are converted automatically to decimals. AWS IoT attempts to convert the string to a Decimal and round down to the nearest Int.
Array	Undefined.
Object	Undefined.
Null	Undefined.
Undefined	Undefined.

Cast to Boolean

Argument type	Result
Int	0 = False, any_nonzero_value = True.
Decimal	0 = False, any_nonzero_value = True.
Boolean	The source value.
String	"true" = True and "false" = False (case insensitive). Other string values = Undefined.
Array	Undefined.
Object	Undefined.
Null	Undefined.
Undefined	Undefined.

Cast to string

Argument type	Result
Int	A string representation of the Int, in standard notation.
Decimal	A string representing the Decimal value, possibly in scientific notation.
Boolean	"true" or "false", all lowercase.
String	"true"=True and "false"=False (case-insensitive). Other string values = Undefined.
Array	The array serialized to JSON. The result string is a comma-separated list enclosed in square brackets. String is quoted. Decimal, Int, and Boolean are not.

Argument type	Result
Object	The object serialized to JSON. The JSON string is a comma-separated list of key-value pairs and begins and ends with curly braces. String is quoted. Decimal, Int, Boolean, and Null are not.
Null	Undefined.
Undefined	Undefined.

ceil(Decimal)

Rounds the given Decimal up to the nearest Int. Supported by SQL version 2015-10-08 and later.

Examples:

```
ceil(1.2) = 2
ceil(-1.2) = -1
```

Argument type	Result
Int	Int, the argument value.
Decimal	Int, the Decimal value rounded up to the nearest Int.
String	Int. The string is converted to Decimal and rounded up to the nearest Int. If the string cannot be converted to a Decimal, the result is Undefined.
Other value	Undefined.

chr(String)

Returns the ASCII character that corresponds to the given Int argument. Supported by SQL version 2015-10-08 and later.

Examples:

```
chr(65) = "A".
chr(49) = "1".
```

Argument type	Result
Int	The character corresponding to the specified ASCII value. If the argument is not a valid ASCII value, the result is Undefined.
Decimal	The character corresponding to the specified ASCII value. The Decimal argument is rounded down to the nearest Int. If the argument is not a valid ASCII value, the result is Undefined.
Boolean	Undefined.

Argument type	Result
String	If the String can be converted to a Decimal, it is rounded down to the nearest Int. If the argument is not a valid ASCII value, the result is Undefined.
Array	Undefined.
Object	Undefined.
Null	Undefined.
Other value	Undefined.

clientid()

Returns the ID of the MQTT client sending the message, or n/a if the message wasn't sent over MQTT. Supported by SQL version 2015-10-08 and later.

Example:

```
clientid() = "123456789012"
```

concat()

Concatenates arrays or strings. This function accepts any number of arguments and returns a String or an Array. Supported by SQL version 2015-10-08 and later.

Examples:

```
concat() = Undefined.
```

```
concat(1) = "1".
```

```
concat([1, 2, 3], 4) = [1, 2, 3, 4].
```

```
concat([1, 2, 3], "hello") = [1, 2, 3, "hello"]
```

```
concat("con", "cat") = "concat"
```

```
concat(1, "hello") = "1hello"
```

```
concat("he", "is", "man") = "heisman"
```

```
concat([1, 2, 3], "hello", [4, 5, 6]) = [1, 2, 3, "hello", 4, 5, 6]
```

Number of arguments	Result
0	Undefined.
1	The argument is returned unmodified.
2+	If any argument is an Array, the result is a single array containing all of the arguments. If no arguments are arrays, and at least one argument is a String, the result is the concatenation of the String representations of all the arguments. Arguments are converted to strings using the standard conversions listed above. •

cos(Decimal)

Returns the cosine of a number in radians. Decimal arguments are rounded to double precision before function application. Supported by SQL version 2015-10-08 and later.

Example:

$\cos(0) = 1$.

Argument type	Result
Int	Decimal (with double precision), the cosine of the argument. Imaginary results are returned as Undefined.
Decimal	Decimal (with double precision), the cosine of the argument. Imaginary results are returned as Undefined.
Boolean	Undefined.
String	Decimal (with double precision), the cosine of the argument. If the string cannot be converted to a Decimal, the result is Undefined. Imaginary results are returned as Undefined.
Array	Undefined.
Object	Undefined.
Null	Undefined.
Undefined	Undefined.

cosh(Decimal)

Returns the hyperbolic cosine of a number in radians. Decimal arguments are rounded to double precision before function application. Supported by SQL version 2015-10-08 and later.

Example: $\cosh(2.3) = 5.037220649268761$.

Argument type	Result
Int	Decimal (with double precision), the hyperbolic cosine of the argument. Imaginary results are returned as Undefined.
Decimal	Decimal (with double precision), the hyperbolic cosine of the argument. Imaginary results are returned as Undefined.
Boolean	Undefined.
String	Decimal (with double precision), the hyperbolic cosine of the argument. If the string cannot be converted to a Decimal, the result is Undefined. Imaginary results are returned as Undefined.
Array	Undefined.

Argument type	Result
Object	Undefined.
Null	Undefined.
Undefined	Undefined.

decode(value, decodingScheme)

Use the `decode` function to decode an encoded value. If the decoded string is a JSON document, an addressable object is returned. Otherwise, the decoded string is returned as a string. The function returns `NULL` if the string cannot be decoded.

Supported by SQL version 2016-03-23 and later.

`value`

A string value or any of the valid expressions, as defined in [AWS IoT SQL reference \(p. 529\)](#), that return a string.

`decodingScheme`

A literal string representing the scheme used to decode the value. Currently, only '`base64`' is supported.

Example

In this example, the message payload includes an encoded value.

```
{
    encoded_temp: "eyAidGVtcGVyYXR1cmUiOiAzMyB9Cg=="
}
```

The `decode` function in this SQL statement, decodes the value in the message payload.

```
SELECT decode(encoded_temp, "base64").temperature AS temp from 'topic/subtopic'
```

Decoding the `encoded_temp` value results in the following valid JSON document, which allows the `SELECT` statement to read the temperature value.

```
{ "temperature": 33 }
```

The result of the `SELECT` statement in this example is shown here.

```
{ "temp": 33 }
```

If the decoded value was not a valid JSON document, the decoded value would be returned as a string.

encode(value, encodingScheme)

Use the `encode` function to encode the payload, which potentially might be non-JSON data, into its string representation based on the encoding scheme. Supported by SQL version 2016-03-23 and later.

value

Any of the valid expressions, as defined in [AWS IoT SQL reference \(p. 529\)](#). You can specify * to encode the entire payload, regardless of whether it's in JSON format. If you supply an expression, the result of the evaluation is converted to a string before it is encoded.

encodingScheme

A literal string representing the encoding scheme you want to use. Currently, only 'base64' is supported.

endswith(String, String)

Returns a Boolean indicating whether the first `String` argument ends with the second `String` argument. If either argument is `Null` or `Undefined`, the result is `Undefined`. Supported by SQL version 2015-10-08 and later.

Example: `endswith("cat", "at") = true`.

Argument type 1	Argument type 2	Result
<code>String</code>	<code>String</code>	True if the first argument ends in the second argument. Otherwise, false.
Other value	Other value	Both arguments are converted to strings using standard conversion rules. The result is true if the first argument ends in the second argument. Otherwise, false. If either argument is <code>Null</code> or <code>Undefined</code> , the result is <code>Undefined</code> .

exp(Decimal)

Returns e raised to the `Decimal` argument. `Decimal` arguments are rounded to double precision before function application. Supported by SQL version 2015-10-08 and later.

Example: `exp(1) = e`.

Argument type	Result
<code>Int</code>	<code>Decimal</code> (with double precision), e^{argument} .
<code>Decimal</code>	<code>Decimal</code> (with double precision), e^{argument} .
<code>String</code>	<code>Decimal</code> (with double precision), e^{argument} . If the <code>String</code> cannot be converted to a <code>Decimal</code> , the result is <code>Undefined</code> .
Other value	<code>Undefined</code> .

floor(Decimal)

Rounds the given `Decimal` down to the nearest `Int`. Supported by SQL version 2015-10-08 and later.

Examples:

`floor(1.2) = 1`

`floor(-1.2) = -2`

Argument type	Result
<code>Int</code>	<code>Int</code> , the argument value.
<code>Decimal</code>	<code>Int</code> , the <code>Decimal</code> value rounded down to the nearest <code>Int</code> .
<code>String</code>	<code>Int</code> . The string is converted to <code>Decimal</code> and rounded down to the nearest <code>Int</code> . If the string cannot be converted to a <code>Decimal</code> , the result is <code>Undefined</code> .
Other value	<code>Undefined</code> .

get

Extracts a value from a collection-like type (`Array`, `String`, `Object`). No conversion is applied to the first argument. Conversion applies as documented in the table to the second argument. Supported by SQL version 2015-10-08 and later.

Examples:

`get(["a", "b", "c"], 1) = "b"`

`get({"a": "b"}, "a") = "b"`

`get("abc", 1) = "a"`

Argument type 1	Argument type 2	Result
<code>Array</code>	<code>Any Type</code> (converted to <code>Int</code>)	The item at the 0-based index specified by the second argument. If the conversion is unsuccessful or the index is outside the bounds of the array (<code>array.length</code>), the result is <code>Undefined</code> .
<code>String</code>	<code>Any Type</code> (converted to <code>Int</code>)	The character at the 0-based index specified by the second argument. If the conversion is unsuccessful or the index is outside the bounds of the string (<code>string.length</code>), the result is <code>Undefined</code> .
<code>Object</code>	<code>String</code> (no conversion is applied)	The value stored in the object corresponding to the key specified by the second argument.
Other value	<code>Any value</code>	<code>Undefined</code> .

get_dynamodb(tableName, partitionKeyName, partitionKeyValue, sortKeyName, sortKeyValue, roleArn)

Retrieves data from a DynamoDB table. `get_dynamodb()` allows you to query a DynamoDB table while a rule is evaluated. You can filter or augment message payloads using data retrieved from DynamoDB. Supported by SQL version 2016-03-23 and later.

`get_dynamodb()` takes the following parameters:

`tableName`

The name of the DynamoDB table to query.

`partitionKeyName`

The name of the partition key. For more information, see [DynamoDB Keys](#).

`partitionKeyValue`

The value of the partition key used to identify a record. For more information, see [DynamoDB Keys](#).

`sortKeyName`

(Optional) The name of the sort key. This parameter is required only if the DynamoDB table queried uses a composite key. For more information, see [DynamoDB Keys](#).

`sortKeyValue`

(Optional) The value of the sort key. This parameter is required only if the DynamoDB table queried uses a composite key. For more information, see [DynamoDB Keys](#).

`roleArn`

The ARN of an IAM role that grants access to the DynamoDB table. The rules engine assumes this role to access the DynamoDB table on your behalf. Avoid using an overly permissive role. Grant the role only those permissions required by the rule. The following is an example policy that grants access to one DynamoDB table.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": "dynamodb:GetItem",  
            "Resource": "arn:aws:dynamodb:aws-region:account-id:table/table-name"  
        }  
    ]  
}
```

As an example of how you can use `get_dynamodb()`, say you have a DynamoDB table that contains device ID and location information for all of your devices connected to AWS IoT. The following SELECT statement uses the `get_dynamodb()` function to retrieve the location for the specified device ID:

```
SELECT *, get_dynamodb("InServiceDevices", "deviceId", id,  
"arn:aws:iam::12345678910:role/getdynamo").location AS location FROM 'some/  
topic'
```

Note

- You can call `get_dynamodb()` a maximum of one time per SQL statement. Calling `get_dynamodb()` multiple times in a single SQL statement causes the rule to terminate without invoking any actions.
- If `get_dynamodb()` returns more than 8 KB of data, the rule's action may not be invoked.

[get_secret\(secretId, secretType, key, roleArn\)](#)

Retrieves the value of the encrypted `SecretString` or `SecretBinary` field of the current version of a secret in [AWS Secrets Manager](#). For more information about creating and maintaining secrets, see [CreateSecret](#), [UpdateSecret](#), and [PutSecretValue](#).

`get_secret()` takes the following parameters:

`secretId`

String: The Amazon Resource Name (ARN) or the friendly name of the secret to retrieve.

`secretType`

String: The secret type. Valid values: `SecretString` | `SecretBinary`.

`SecretString`

- For secrets that you create as JSON objects by using the APIs, the AWS CLI, or the AWS Secrets Manager console:
 - If you specify a value for the `key` parameter, this function returns the value of the specified key.
 - If you don't specify a value for the `key` parameter, this function returns the entire JSON object.
- For secrets that you create as non-JSON objects by using the APIs or the AWS CLI:
 - If you specify a value for the `key` parameter, this function fails with an exception.
 - If you don't specify a value for the `key` parameter, this function returns the contents of the secret.

`SecretBinary`

- If you specify a value for the `key` parameter, this function fails with an exception.
- If you don't specify a value for the `key` parameter, this function returns the secret value as a base64-encoded UTF-8 string.

`key`

(Optional) String: The key name inside a JSON object stored in the `SecretString` field of a secret. Use this value when you want to retrieve only the value of a key stored in a secret instead of the entire JSON object.

If you specify a value for this parameter and the secret doesn't contain a JSON object inside its `SecretString` field, this function fails with an exception.

`roleArn`

String: A role ARN with `secretsmanager:GetSecretValue` and `secretsmanager:DescribeSecret` permissions.

Note

This function always returns the current version of the secret (the version with the `AWSCURRENT` tag). The AWS IoT rules engine caches each secret for up to 15 minutes. As a result, the rules engine can take up to 15 minutes to update a secret. This means that if you retrieve a secret up to 15 minutes after an update with AWS Secrets Manager, this function might return the older version.

This function is not metered and is free to use, but AWS Secrets Manager charges apply. Because of the secret caching mechanism, the rules engine occasionally calls AWS Secrets Manager. Because the rules engine is a fully distributed service, you might see multiple Secrets Manager API calls from the rules engine during the 15-minute caching window.

Examples:

You can use the `get_secret` function in an authentication header in an HTTPS rule action, as in the following API key authentication example.

```
"API_KEY": "${get_secret('API_KEY', 'SecretString', 'API_KEY_VALUE',  
'arn:aws:iam::12345678910:role/getsecret')}"
```

For more information about the HTTPS rule action, see [the section called “HTTP” \(p. 468\)](#).

get_thing_shadow(thingName, shadowName, roleARN)

Returns the specified shadow of the specified thing. Supported by SQL version 2016-03-23 and later.

thingName

String: The name of the thing whose shadow you want to retrieve.

shadowName

(Optional) String: The name of the shadow. This parameter is required only when referencing named shadows.

roleArn

String: A role ARN with `iot:GetThingShadow` permission.

Examples:

When used with a named shadow, provide the shadowName parameter.

```
SELECT * from 'topic/subtopic'  
WHERE  
    get_thing_shadow("MyThing", "MyThingShadow", "arn:aws:iam::123456789012:role/  
AllowsThingShadowAccess")  
    .state.reported.alarm = 'ON'
```

When used with an unnamed shadow, omit the shadowName parameter.

```
SELECT * from 'topic/subtopic'  
WHERE  
    get_thing_shadow("MyThing", "arn:aws:iam::123456789012:role/AllowsThingShadowAccess")  
    .state.reported.alarm = 'ON'
```

Hashing functions

AWS IoT provides the following hashing functions:

- md2
- md5
- sha1
- sha224
- sha256
- sha384
- sha512

All hash functions expect one string argument. The result is the hashed value of that string. Standard string conversions apply to non-string arguments. All hash functions are supported by SQL version 2015-10-08 and later.

Examples:

```
md2("hello") = "a9046c73e00331af68917d3804f70655"
```

```
md5("hello") = "5d41402abc4b2a76b9719d911017c592"
```

[indexof\(String, String\)](#)

Returns the first index (0-based) of the second argument as a substring in the first argument. Both arguments are expected as strings. Arguments that are not strings are subjected to standard string conversion rules. This function does not apply to arrays, only to strings. Supported by SQL version 2016-03-23 and later.

Examples:

```
indexof("abcd", "bc") = 1
```

[isNull\(\)](#)

Returns true if the argument is the Null value. Supported by SQL version 2015-10-08 and later.

Examples:

```
isNull(5) = false.
```

```
isNull(null) = true.
```

Argument type	Result
Int	false
Decimal	false
Boolean	false
String	false
Array	false
Object	false
Null	true
Undefined	false

[isUndefined\(\)](#)

Returns true if the argument is Undefined. Supported by SQL version 2016-03-23 and later.

Examples:

```
isUndefined(5) = false.
```

```
isUndefined(floor([1,2,3])) = true.
```

Argument type	Result
Int	false
Decimal	false

Argument type	Result
Boolean	false
String	false
Array	false
Object	false
Null	false
Undefined	true

length(String)

Returns the number of characters in the provided string. Standard conversion rules apply to non-String arguments. Supported by SQL version 2016-03-23 and later.

Examples:

`length("hi") = 2`

`length(false) = 5`

ln(Decimal)

Returns the natural logarithm of the argument. Decimal arguments are rounded to double precision before function application. Supported by SQL version 2015-10-08 and later.

Example: `ln(e) = 1.`

Argument type	Result
Int	Decimal (with double precision), the natural log of the argument.
Decimal	Decimal (with double precision), the natural log of the argument.
Boolean	Undefined.
String	Decimal (with double precision), the natural log of the argument. If the string cannot be converted to a Decimal, the result is Undefined.
Array	Undefined.
Object	Undefined.
Null	Undefined.
Undefined	Undefined.

log(Decimal)

Returns the base 10 logarithm of the argument. Decimal arguments are rounded to double precision before function application. Supported by SQL version 2015-10-08 and later.

Example: `log(100) = 2.0.`

Argument type	Result
Int	Decimal (with double precision), the base 10 log of the argument.
Decimal	Decimal (with double precision), the base 10 log of the argument.
Boolean	Undefined.
String	Decimal (with double precision), the base 10 log of the argument. If the String cannot be converted to a Decimal, the result is Undefined.
Array	Undefined.
Object	Undefined.
Null	Undefined.
Undefined	Undefined.

lower(String)

Returns the lowercase version of the given String. Non-string arguments are converted to strings using the standard conversion rules. Supported by SQL version 2015-10-08 and later.

Examples:

`lower("HELLO") = "hello".`

`lower(["HELLO"]) = ["hello"].`

lpad(String, Int)

Returns the String argument, padded on the left side with the number of spaces specified by the second argument. The Int argument must be between 0 and 1000. If the provided value is outside of this valid range, the argument is set to the nearest valid value (0 or 1000). Supported by SQL version 2015-10-08 and later.

Examples:

`lpad("hello", 2) = " hello".`

`lpad(1, 3) = " 1"`

Argument type 1	Argument type 2	Result
String	Int	String, the provided with a number of spaces.
String	Decimal	The Decimal argument. Int and the String specified number of spaces.

Argument type 1	Argument type 2	Result
String	String	The second argument is rounded down to the specified number of digits and padded with the specified character. If the second argument is not provided, the result is Undefined.
Other value	Int/Decimal/String	The first value is converted to standard conversions applied on that String. The result is Undefined.
Any value	Other value	Undefined.

[ltrim\(String\)](#)

Removes all leading white space (tabs and spaces) from the provided `String`. Supported by SQL version 2015-10-08 and later.

Example:

```
Ltrim(" h i ") = "hi".
```

Argument type	Result
Int	The <code>String</code> representation of the <code>Int</code> with all leading white space removed.
Decimal	The <code>String</code> representation of the <code>Decimal</code> with all leading white space removed.
Boolean	The <code>String</code> representation of the boolean ("true" or "false") with all leading white space removed.
String	The argument with all leading white space removed.
Array	The <code>String</code> representation of the <code>Array</code> (using standard conversion rules) with all leading white space removed.
Object	The <code>String</code> representation of the <code>Object</code> (using standard conversion rules) with all leading white space removed.
Null	<code>Undefined</code> .
Undefined	<code>Undefined</code> .

[machinelearning_predict\(modelId, roleArn, record\)](#)

Use the `machinelearning_predict` function to make predictions using the data from an MQTT message based on an Amazon Machine Learning (Amazon ML) model. Supported by SQL version 2015-10-08 and later. The arguments for the `machinelearning_predict` function are:

`modelId`

The ID of the model against which to run the prediction. The real-time endpoint of the model must be enabled.

roleArn

The IAM role that has a policy with `machinelearning:Predict` and `machinelearning:GetMLModel` permissions and allows access to the model against which the prediction is run.

record

The data to be passed into the Amazon ML Predict API. This should be represented as a single layer JSON object. If the record is a multi-level JSON object, the record is flattened by serializing its values. For example, the following JSON:

```
{ "key1": {"innerKey1": "value1"}, "key2": 0}
```

would become:

```
{ "key1": "{\"innerKey1\": \"value1\"}", "key2": 0}
```

The function returns a JSON object with the following fields:

predictedLabel

The classification of the input based on the model.

details

Contains the following attributes:

PredictiveModelType

The model type. Valid values are REGRESSION, BINARY, MULTICLASS.

Algorithm

The algorithm used by Amazon ML to make predictions. The value must be SGD.

predictedScores

Contains the raw classification score corresponding to each label.

predictedValue

The value predicted by Amazon ML.

mod(Decimal, Decimal)

Returns the remainder of the division of the first argument by the second argument. Equivalent to [remainder\(Decimal, Decimal\) \(p. 569\)](#). You can also use "%" as an infix operator for the same modulo functionality. Supported by SQL version 2015-10-08 and later.

Example: `mod(8, 3) = 2`.

Left operand	Right operand	Output
Int	Int	Int, the first argument converted to an integer.
Int/Decimal	Int/Decimal	Decimal, the first argument converted to a decimal.
String/Int/Decimal	String/Int/Decimal	If all strings convert to numbers, the result is a decimal. If any string does not convert to a number, the result is undefined.

Left operand	Right operand	Output
Other value	Other value	Undefined.

nanvl(AnyValue, AnyValue)

Returns the first argument if it is a valid `Decimal`. Otherwise, the second argument is returned.
Supported by SQL version 2015-10-08 and later.

Example: `Nanvl(8, 3) = 8.`

Argument type 1	Argument type 2	Output
Undefined	Any value	The second argument.
Null	Any value	The second argument.
<code>Decimal</code> (<code>NaN</code>)	Any value	The second argument.
<code>Decimal</code> (not <code>NaN</code>)	Any value	The first argument.
Other value	Any value	The first argument.

newuuid()

Returns a random 16-byte UUID. Supported by SQL version 2015-10-08 and later.

Example: `newuuid() = 123a4567-b89c-12d3-e456-789012345000`

numbytes(String)

Returns the number of bytes in the UTF-8 encoding of the provided string. Standard conversion rules apply to non-String arguments. Supported by SQL version 2016-03-23 and later.

Examples:

`numbytes("hi") = 2`

`numbytes("€") = 3`

parse_time(String, Long[, String])

Use the `parse_time` function to format a timestamp into a human-readable date/time format. Supported by SQL version 2016-03-23 and later. To convert a timestamp string into milliseconds, see [time_to_epoch\(String, String\) \(p. 577\)](#).

The `parse_time` function expects the following arguments:

pattern

(String) A date/time pattern that follows the [ISO 8601](#) standard format. Specifically, the function supports [Joda-Time formats](#).

timestamp

(Long) The time to be formatted in milliseconds since Unix epoch. See function [timestamp\(\) \(p. 578\)](#).

timezone

(String) The time zone of the formatted date/time. The default is "UTC". The function supports [Joda-Time time zones](#). This argument is optional.

Examples:

When this message is published to the topic 'A/B', the payload `{"ts": "1970.01.01 AD at 21:46:40 CST"}` is sent to the S3 bucket:

```
{
    "ruleArn": "arn:aws:iot:us-east-2:ACCOUNT_ID:rule/RULE_NAME",
    "topicRulePayload": {
        "sql": "SELECT parse_time('yyyy.MM.dd G \'at\' HH:mm:ss z", 100000000, \"America/Belize\" ) as ts FROM 'A/B'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "s3": {
                    "roleArn": "arn:aws:iam::ACCOUNT_ID:rule:role/ROLE_NAME",
                    "bucketName": "BUCKET_NAME",
                    "key": "KEY_NAME"
                }
            }
        ],
        "ruleName": "RULE_NAME"
    }
}
```

When this message is published to the topic 'A/B', a payload similar to `{"ts": "2017.06.09 AD at 17:19:46 UTC"}` (but with the current date/time) is sent to the S3 bucket:

```
{
    "ruleArn": "arn:aws:iot:us-east-2:ACCOUNT_ID:rule/RULE_NAME",
    "topicRulePayload": {
        "sql": "SELECT parse_time('yyyy.MM.dd G \'at\' HH:mm:ss z", timestamp() ) as ts FROM 'A/B'",
        "awsIotSqlVersion": "2016-03-23",
        "ruleDisabled": false,
        "actions": [
            {
                "s3": {
                    "roleArn": "arn:aws:iam::ACCOUNT_ID:rule:role/ROLE_NAME",
                    "bucketName": "BUCKET_NAME",
                    "key": "KEY_NAME"
                }
            }
        ],
        "ruleName": "RULE_NAME"
    }
}
```

`parse_time()` can also be used as a substitution template. For example, when this message is published to the topic 'A/B', the payload is sent to the S3 bucket with key = "2017":

```
{
    "ruleArn": "arn:aws:iot:us-east-2:ACCOUNT_ID:rule/RULE_NAME",
    "topicRulePayload": {
        "sql": "SELECT * FROM 'A/B'" ,
```

```

    "awsIotSqlVersion": "2016-03-23",
    "ruleDisabled": false,
    "actions": [
        {
            "s3": {
                "roleArn": "arn:aws:iam::ACCOUNT_ID:rule:role/ROLE_NAME",
                "bucketName": BUCKET_NAME,
                "key": "${parse_time("yyyy", timestamp(), "UTC")}"
            }
        }
    ],
    "ruleName": "RULE_NAME"
}
}

```

power(Decimal, Decimal)

Returns the first argument raised to the second argument. Decimal arguments are rounded to double precision before function application. Supported by SQL version 2015-10-08 and later. Supported by SQL version 2015-10-08 and later.

Example: `power(2, 5) = 32.0.`

Argument type 1	Argument type 2	Output
Int/Decimal	Int/Decimal	A Decimal (with double precision) raised to the second argument.
Int/Decimal/String	Int/Decimal/String	A Decimal (with double precision) raised to the second argument. String values are converted to decimal and converted to Decimal.
Other value	Other value	Undefined.

principal()

Returns the principal that the device uses for authentication, based on how the triggering message was published. The following table describes the principal returned for each publishing method and protocol.

How the message is published	Protocol	Credential type
MQTT client	MQTT	X.509 device certificate thumbprint
AWS IoT console MQTT client	MQTT	IAM user or role
AWS CLI	HTTP	IAM user or role
AWS IoT Device SDK	MQTT	X.509 device certificate thumbprint
AWS IoT Device SDK	MQTT over WebSocket	IAM user or role

The following examples show the different types of values that `principal()` can return:

- X.509 certificate thumbprint:
`ba67293af50bf2506f5f93469686da660c7c844e7b3950bfb16813e0d31e9373`
- IAM role ID and session name: `ABCD1EFG3HIJK2LMNOP5:my-session-name`

- Returns a user ID: ABCD1EFG3HIJK2LMNOPS

rand()

Returns a pseudorandom, uniformly distributed double between 0.0 and 1.0. Supported by SQL version 2015-10-08 and later.

Example:

```
rand() = 0.8231909191640703
```

regexp_matches(String, String)

Returns true if the string (first argument) contains a match for the regular expression (second argument).

Example:

```
regexp_matches("aaaa", "a{2,}") = true.
```

```
regexp_matches("aaaa", "b") = false.
```

First argument:

Argument type	Result
Int	The String representation of the Int.
Decimal	The String representation of the Decimal.
Boolean	The String representation of the boolean ("true" or "false").
String	The String.
Array	The String representation of the Array (using standard conversion rules).
Object	The String representation of the Object (using standard conversion rules).
Null	Undefined.
Undefined	Undefined.

Second argument:

Must be a valid regex expression. Non-string types are converted to String using the standard conversion rules. Depending on the type, the resultant string might not be a valid regular expression. If the (converted) argument is not valid regex, the result is Undefined.

regexp_replace(String, String, String)

Replaces all occurrences of the second argument (regular expression) in the first argument with the third argument. Reference capture groups with "\$". Supported by SQL version 2015-10-08 and later.

Example:

```
regexp_replace("abcd", "bc", "x") = "axd".
```

```
regexp_replace("abcd", "b(.*)d", "$1") = "ac".
```

First argument:

Argument type	Result
Int	The String representation of the Int.
Decimal	The String representation of the Decimal.
Boolean	The String representation of the boolean ("true" or "false").
String	The source value.
Array	The String representation of the Array (using standard conversion rules).
Object	The String representation of the Object (using standard conversion rules).
Null	Undefined.
Undefined	Undefined.

Second argument:

Must be a valid regex expression. Non-string types are converted to String using the standard conversion rules. Depending on the type, the resultant string might not be a valid regular expression. If the (converted) argument is not a valid regex expression, the result is Undefined.

Third argument:

Must be a valid regex replacement string. (Can reference capture groups.) Non-string types are converted to String using the standard conversion rules. If the (converted) argument is not a valid regex replacement string, the result is Undefined.

regexp_substr(String, String)

Finds the first match of the second parameter (regex) in the first parameter. Reference capture groups with "\$". Supported by SQL version 2015-10-08 and later.

Example:

```
regexp_substr("hihihello", "hi") = "hi"
regexp_substr("hihihello", "(hi)*") = "hihi"
```

First argument:

Argument type	Result
Int	The String representation of the Int.
Decimal	The String representation of the Decimal.
Boolean	The String representation of the boolean ("true" or "false").
String	The String argument.
Array	The String representation of the Array (using standard conversion rules).

Argument type	Result
Object	The String representation of the Object (using standard conversion rules).
Null	Undefined.
Undefined	Undefined.

Second argument:

Must be a valid regex expression. Non-string types are converted to String using the standard conversion rules. Depending on the type, the resultant string might not be a valid regular expression. If the (converted) argument is not a valid regex expression, the result is Undefined.

remainder(Decimal, Decimal)

Returns the remainder of the division of the first argument by the second argument. Equivalent to [mod\(Decimal, Decimal\) \(p. 563\)](#). You can also use "%" as an infix operator for the same modulo functionality. Supported by SQL version 2015-10-08 and later.

Example: `remainder(8, 3) = 2.`

Left operand	Right operand	Output
Int	Int	Int, the first argument
Int/Decimal	Int/Decimal	Decimal, the first argument
String/Int/Decimal	String/Int/Decimal	If all strings convert to Int, the result is Int. Otherwise, it is Undefined.
Other value	Other value	Undefined.

replace(String, String, String)

Replaces all occurrences of the second argument in the first argument with the third argument. Supported by SQL version 2015-10-08 and later.

Example:

```
replace("abcd", "bc", "x") = "axd".
replace("abcdabcd", "b", "x") = "axcdaxcd".
```

All arguments

Argument type	Result
Int	The String representation of the Int.
Decimal	The String representation of the Decimal.
Boolean	The String representation of the boolean ("true" or "false").
String	The source value.

Argument type	Result
Array	The String representation of the Array (using standard conversion rules).
Object	The String representation of the Object (using standard conversion rules).
Null	Undefined.
Undefined	Undefined.

rpad(String, Int)

Returns the string argument, padded on the right side with the number of spaces specified in the second argument. The Int argument must be between 0 and 1000. If the provided value is outside of this valid range, the argument is set to the nearest valid value (0 or 1000). Supported by SQL version 2015-10-08 and later.

Examples:

```
rpad("hello", 2) = "hello  ".
rpad(1, 3) = "1   ".
```

Argument type 1	Argument type 2	Result
String	Int	The String is padded on the right side with a number of spaces equal to the provided Int.
String	Decimal	The Decimal argument is rounded down to the nearest Int and the string is padded

Argument type 1	Argument type 2	Result
		on the right side with a number of spaces equal to the provided Int.
String	String	The second argument is converted to a Decimal, which is rounded down to the nearest Int. The String is padded on the right side with a number of spaces equal to the Int value.

Argument type 1	Argument type 2	Result
Other value	Int/Decimal/String	The first value is converted to a String using the standard conversions, and the rpad function is applied on that String. If it cannot be converted, the result is Undefined.
Any value	Other value	Undefined.

round(Decimal)

Rounds the given Decimal to the nearest Int. If the Decimal is equidistant from two Int values (for example, 0.5), the Decimal is rounded up. Supported by SQL version 2015-10-08 and later.

Example: Round(1.2) = 1.

Round(1.5) = 2.

Round(1.7) = 2.

Round(-1.1) = -1.

Round(-1.5) = -2.

Argument type	Result
Int	The argument.
Decimal	Decimal is rounded down to the nearest Int.
String	Decimal is rounded down to the nearest Int. If the string cannot be converted to a Decimal, the result is Undefined.

Argument type	Result
Other value	Undefined.

rtrim(String)

Removes all trailing white space (tabs and spaces) from the provided `String`. Supported by SQL version 2015-10-08 and later.

Examples:

`rtrim(" h i ") = " h i"`

Argument type	Result
<code>Int</code>	The <code>String</code> representation of the <code>Int</code> .
<code>Decimal</code>	The <code>String</code> representation of the <code>Decimal</code> .
<code>Boolean</code>	The <code>String</code> representation of the boolean ("true" or "false").
<code>Array</code>	The <code>String</code> representation of the <code>Array</code> (using standard conversion rules).
<code>Object</code>	The <code>String</code> representation of the <code>Object</code> (using standard conversion rules).
<code>Null</code>	Undefined.
<code>Undefined</code>	Undefined

sign(Decimal)

Returns the sign of the given number. When the sign of the argument is positive, 1 is returned. When the sign of the argument is negative, -1 is returned. If the argument is 0, 0 is returned. Supported by SQL version 2015-10-08 and later.

Examples:

`sign(-7) = -1.`

`sign(0) = 0.`

`sign(13) = 1.`

Argument type	Result
<code>Int</code>	<code>Int</code> , the sign of the <code>Int</code> value.
<code>Decimal</code>	<code>Int</code> , the sign of the <code>Decimal</code> value.
<code>String</code>	<code>Int</code> , the sign of the <code>Decimal</code> value. The string is converted to a <code>Decimal</code> value, and the sign of the <code>Decimal</code> value is returned. If the <code>String</code> cannot be converted to a <code>Decimal</code> , the result is <code>Undefined</code> . Supported by SQL version 2015-10-08 and later.

Argument type	Result
Other value	Undefined.

sin(Decimal)

Returns the sine of a number in radians. `Decimal` arguments are rounded to double precision before function application. Supported by SQL version 2015-10-08 and later.

Example: `sin(0) = 0.0`

Argument type	Result
<code>Int</code>	<code>Decimal</code> (with double precision), the sine of the argument.
<code>Decimal</code>	<code>Decimal</code> (with double precision), the sine of the argument.
<code>Boolean</code>	Undefined.
<code>String</code>	<code>Decimal</code> (with double precision), the sine of the argument. If the string cannot be converted to a <code>Decimal</code> , the result is Undefined.
<code>Array</code>	Undefined.
<code>Object</code>	Undefined.
<code>Null</code>	Undefined.
<code>Undefined</code>	Undefined.

sinh(Decimal)

Returns the hyperbolic sine of a number. `Decimal` values are rounded to double precision before function application. The result is a `Decimal` value of double precision. Supported by SQL version 2015-10-08 and later.

Example: `sinh(2.3) = 4.936961805545957`

Argument type	Result
<code>Int</code>	<code>Decimal</code> (with double precision), the hyperbolic sine of the argument.
<code>Decimal</code>	<code>Decimal</code> (with double precision), the hyperbolic sine of the argument.
<code>Boolean</code>	Undefined.
<code>String</code>	<code>Decimal</code> (with double precision), the hyperbolic sine of the argument. If the string cannot be converted to a <code>Decimal</code> , the result is Undefined.
<code>Array</code>	Undefined.

Argument type	Result
Object	Undefined.
Null	Undefined.
Undefined	Undefined.

substring(String, Int[, Int])

Expects a `String` followed by one or two `Int` values. For a `String` and a single `Int` argument, this function returns the substring of the provided `String` from the provided `Int` index (0-based, inclusive) to the end of the `String`. For a `String` and two `Int` arguments, this function returns the substring of the provided `String` from the first `Int` index argument (0-based, inclusive) to the second `Int` index argument (0-based, exclusive). Indices that are less than zero are set to zero. Indices that are greater than the `String` length are set to the `String` length. For the three argument version, if the first index is greater than (or equal to) the second index, the result is the empty `String`.

If the arguments provided are not (`String, Int`), or (`String, Int, Int`), the standard conversions are applied to the arguments to attempt to convert them into the correct types. If the types cannot be converted, the result of the function is `Undefined`. Supported by SQL version 2015-10-08 and later.

Examples:

```
substring("012345", 0) = "012345".
substring("012345", 2) = "2345".
substring("012345", 2.745) = "2345".
substring(123, 2) = "3".
substring("012345", -1) = "012345".
substring(true, 1.2) = "true".
substring(false, -2.411E247) = "false".
substring("012345", 1, 3) = "12".
substring("012345", -50, 50) = "012345".
substring("012345", 3, 1) = "".
```

sql_version()

Returns the SQL version specified in this rule. Supported by SQL version 2015-10-08 and later.

Example:

```
sql_version() = "2016-03-23"
```

sqrt(Decimal)

Returns the square root of a number. `Decimal` arguments are rounded to double precision before function application. Supported by SQL version 2015-10-08 and later.

Example: `sqrt(9) = 3.0.`

Argument type	Result
Int	The square root of the argument.
Decimal	The square root of the argument.
Boolean	Undefined.
String	The square root of the argument. If the string cannot be converted to a Decimal, the result is Undefined.
Array	Undefined.
Object	Undefined.
Null	Undefined.
Undefined	Undefined.

startswith(String, String)

Returns Boolean, whether the first string argument starts with the second string argument. If either argument is Null or Undefined, the result is Undefined. Supported by SQL version 2015-10-08 and later.

Example:

`startswith("ranger", "ran") = true`

Argument type 1	Argument type 2	Result
String	String	Whether the first string starts with the second string
Other value	Other value	Both arguments are converted to strings. If either argument is Null or Undefined, the result is Undefined. Standard conversion rules apply.

tan(Decimal)

Returns the tangent of a number in radians. Decimal values are rounded to double precision before function application. Supported by SQL version 2015-10-08 and later.

Example: `tan(3) = -0.1425465430742778`

Argument type	Result
Int	Decimal (with double precision), the tangent of the argument.
Decimal	Decimal (with double precision), the tangent of the argument.

Argument type	Result
Boolean	Undefined.
String	Decimal (with double precision), the tangent of the argument. If the string cannot be converted to a Decimal, the result is Undefined.
Array	Undefined.
Object	Undefined.
Null	Undefined.
Undefined	Undefined.

tanh(Decimal)

Returns the hyperbolic tangent of a number in radians. Decimal values are rounded to double precision before function application. Supported by SQL version 2015-10-08 and later.

Example: `tanh(2.3) = 0.9800963962661914`

Argument type	Result
Int	Decimal (with double precision), the hyperbolic tangent of the argument.
Decimal	Decimal (with double precision), the hyperbolic tangent of the argument.
Boolean	Undefined.
String	Decimal (with double precision), the hyperbolic tangent of the argument. If the string cannot be converted to a Decimal, the result is Undefined.
Array	Undefined.
Object	Undefined.
Null	Undefined.
Undefined	Undefined.

time_to_epoch(String, String)

Use the `time_to_epoch` function to convert a timestamp string into a number of milliseconds in Unix epoch time. Supported by SQL version 2016-03-23 and later. To convert milliseconds to a formatted timestamp string, see [parse_time\(String, Long\[, String\]\) \(p. 564\)](#).

The `time_to_epoch` function expects the following arguments:

`timestamp`

(String) The timestamp string to be converted to milliseconds since Unix epoch. If the timestamp string doesn't specify a timezone, the function uses the UTC timezone.

pattern

(String) A date/time pattern that follows the [ISO 8601](#) standard format. Specifically, the function supports [JDK11 Time Formats](#).

Examples:

```
time_to_epoch("2020-04-03 09:45:18 UTC+01:00", "yyyy-MM-dd HH:mm:ss VV") = 1585903518000
```

```
time_to_epoch("18 December 2015", "dd MMMM yyyy") = 1450396800000
```

```
time_to_epoch("2007-12-03 10:15:30.592 America/Los_Angeles", "yyyy-MM-dd HH:mm:ss.SSS z") = 1196705730592
```

timestamp()

Returns the current timestamp in milliseconds from 00:00:00 Coordinated Universal Time (UTC), Thursday, 1 January 1970, as observed by the AWS IoT rules engine. Supported by SQL version 2015-10-08 and later.

Example: `timestamp() = 1481825251155`

topic(Decimal)

Returns the topic to which the message that triggered the rule was sent. If no parameter is specified, the entire topic is returned. The `Decimal` parameter is used to specify a specific topic segment, with `1` designating the first segment. For the topic `foo/bar/baz`, `topic(1)` returns `foo`, `topic(2)` returns `bar`, and so on. Supported by SQL version 2015-10-08 and later.

Examples:

```
topic() = "things/myThings/thingOne"
```

```
topic(1) = "things"
```

When [Basic Ingest \(p. 528\)](#) is used, the initial prefix of the topic (`$aws/rules/rule-name`) is not available to the `topic()` function. For example, given the topic:

```
$aws/rules/BuildingManager/Buildings/Building5/Floor2/Room201/Lights
```

```
topic() = "Buildings/Building5/Floor2/Room201/Lights"
```

```
topic(3) = "Floor2"
```

traceid()

Returns the trace ID (UUID) of the MQTT message, or `Undefined` if the message wasn't sent over MQTT. Supported by SQL version 2015-10-08 and later.

Example:

```
traceid() = "12345678-1234-1234-1234-123456789012"
```

transform(String, Object, Array)

Returns an array of objects that contains the result of the specified transformation of the `Object` parameter on the `Array` parameter.

Supported by SQL version 2016-03-23 and later.

String

The transformation mode to use. Refer to the following table for the supported transformation modes and how they create the `Result` from the `Object` and `Array` parameters.

Object

An object that contains the attributes to apply to each element of the `Array`.

Array

An array of objects into which the attributes of `Object` are applied.

Each object in this `Array` corresponds to an object in the function's response. Each object in the function's response contains the attributes present in the original object and the attributes provided by `Object` as determined by the transformation mode specified in `String`.

String parameter	Object parameter	Array parameter	Result
enrichArray	Object	Array of objects	An Array of objects in which each object contains the attributes of an element from the <code>Array</code> parameter and the attributes of the <code>Object</code> parameter.
Any other value	Any value	Any value	Undefined

Note

The array returned by this function is limited to 128 KiB.

Transform function example 1

This example shows how the `transform()` function produces a single array of objects from a data object and an array.

In this example, the following message is published to the MQTT topic A/B.

```
{
  "attributes": {
    "data1": 1,
    "data2": 2
  },
  "values": [
    {
      "a": 3
    },
    {
      "b": 4
    },
    {
      "c": 5
    }
  ]
}
```

This SQL statement for a topic rule action uses the `transform()` function with a `String` value of `enrichArray`. In this example, `Object` is the `attributes` property from the message payload and `Array` is the `values` array, which contains three objects.

```
select value transform("enrichArray", attributes, values) from 'A/B'
```

Upon receiving the message payload, the SQL statement evaluates to the following response.

```
[  
  {  
    "a": 3,  
    "data1": 1,  
    "data2": 2  
  },  
  {  
    "b": 4,  
    "data1": 1,  
    "data2": 2  
  },  
  {  
    "c": 5,  
    "data1": 1,  
    "data2": 2  
  }  
]
```

Transform function example 2

This example shows how the **transform()** function can use literal values to include and rename individual attributes from the message payload.

In this example, the following message is published to the MQTT topic A/B. This is the same message that was used in [the section called “Transform function example 1” \(p. 579\)](#).

```
{  
  "attributes": {  
    "data1": 1,  
    "data2": 2  
  },  
  "values": [  
    {  
      "a": 3  
    },  
    {  
      "b": 4  
    },  
    {  
      "c": 5  
    }  
  ]  
}
```

This SQL statement for a topic rule action uses the **transform()** function with a **String** value of **enrichArray**. The **Object** in the **transform()** function has a single attribute named **key** with the value of **attributes.data1** in the message payload and **Array** is the **values** array, which contains the same three objects used in the previous example.

```
select value transform("enrichArray", {"key": attributes.data1}, values) from 'A/B'
```

Upon receiving the message payload, this SQL statement evaluates to the following response. Notice how the **data1** property is named **key** in the response.

```
[
```

```
{
  "a": 3,
  "key": 1
},
{
  "b": 4,
  "key": 1
},
{
  "c": 5,
  "key": 1
}
]
```

Transform function example 3

This example shows how the **transform()** function can be used in nested SELECT clauses to select multiple attributes and create new objects for subsequent processing.

In this example, the following message is published to the MQTT topic A/B.

```
{
  "data1": "example",
  "data2": {
    "a": "first attribute",
    "b": "second attribute",
    "c": [
      {
        "x": {
          "someInt": 5,
          "someString": "hello"
        },
        "y": true
      },
      {
        "x": {
          "someInt": 10,
          "someString": "world"
        },
        "y": false
      }
    ]
  }
}
```

The Object for this transform function is the object returned by the SELECT statement, which contains the a and b elements of the message's data2 object. The Array parameter consists of the two objects from the data2.c array in the original message.

```
select value transform('enrichArray', (select a, b from data2), (select value c from data2)) from 'A/B'
```

With the preceding message, the SQL statement evaluates to the following response.

```
[
  {
    "x": {
      "someInt": 5,
      "someString": "hello"
    },
    "y": true,
```

```

        "a": "first attribute",
        "b": "second attribute"
    },
{
    "x": {
        "someInt": 10,
        "someString": "world"
    },
    "y": false,
    "a": "first attribute",
    "b": "second attribute"
}
]
```

The array returned in this response could be used with topic rule actions that support `batchMode`.

trim(String)

Removes all leading and trailing white space from the provided `String`. Supported by SQL version 2015-10-08 and later.

Example:

```
trim(" hi ") = "hi"
```

Argument type	Result
<code>Int</code>	The <code>String</code> representation of the <code>Int</code> with all leading and trailing white space removed.
<code>Decimal</code>	The <code>String</code> representation of the <code>Decimal</code> with all leading and trailing white space removed.
<code>Boolean</code>	The <code>String</code> representation of the <code>Boolean</code> ("true" or "false") with all leading and trailing white space removed.
<code>String</code>	The <code>String</code> with all leading and trailing white space removed.
<code>Array</code>	The <code>String</code> representation of the <code>Array</code> using standard conversion rules.
<code>Object</code>	The <code>String</code> representation of the <code>Object</code> using standard conversion rules.
<code>Null</code>	<code>Undefined</code> .
<code>Undefined</code>	<code>Undefined</code> .

trunc(Decimal, Int)

Truncates the first argument to the number of `Decimal` places specified by the second argument. If the second argument is less than zero, it is set to zero. If the second argument is greater than 34, it is set to 34. Trailing zeroes are stripped from the result. Supported by SQL version 2015-10-08 and later.

Examples:

```
trunc(2.3, 0) = 2.
```

```
trunc(2.3123, 2) = 2.31.
```

`trunc(2.888, 2) = 2.88.`

`trunc(2.00, 5) = 2.`

Argument type 1	Argument type 2	Result
Int	Int	The source value.
Int/Decimal	Int/Decimal	The first argument is converted to a Decimal by the second argument. If the second argument is Int, it is rounded down.
Int/Decimal/String	Int/Decimal	The first argument is converted to a Decimal by the second argument. If the second argument is Int, it is rounded down. If the conversion fails, the result is Undefined.
Other value		Undefined.

upper(String)

Returns the uppercase version of the given String. Non-String arguments are converted to String using the standard conversion rules. Supported by SQL version 2015-10-08 and later.

Examples:

`upper("hello") = "HELLO"`

`upper(["hello"]) = ["HELLO"]"`

Literals

You can directly specify literal objects in the SELECT and WHERE clauses of your rule SQL, which can be useful for passing information.

Note

Literals are available only when using SQL version 2016-03-23 or later.

JSON object syntax is used (key-value pairs, comma-separated, where keys are strings and values are JSON values, wrapped in curly brackets {}). For example:

Incoming payload published on topic `topic/subtopic: {"lat_long": [47.606, -122.332]}`

SQL statement: `SELECT {'latitude': get(lat_long, 0), 'longitude': get(lat_long, 1)} as lat_long FROM 'topic/subtopic'`

The resulting outgoing payload would be: `{"lat_long": {"latitude": 47.606, "longitude": -122.332}}`.

You can also directly specify arrays in the SELECT and WHERE clauses of your rule SQL, which allows you to group information. JSON syntax is used (wrap comma-separated items in square brackets [] to create an array literal). For example:

Incoming payload published on topic `topic/subtopic: {"lat": 47.696, "long": -122.332}`

SQL statement: `SELECT [lat, long] as lat_long FROM 'topic/subtopic'`

The resulting output payload would be: { "lat_long": [47.606, -122.332] }.

Case statements

Case statements can be used for branching execution, like a switch statement.

Syntax:

```
CASE v WHEN t[1] THEN r[1]
    WHEN t[2] THEN r[2] ...
    WHEN t[n] THEN r[n]
ELSE r[e] END
```

The expression `v` is evaluated and matched for equality against the `t[i]` value of each `WHEN` clause. If a match is found, the corresponding `r[i]` expression becomes the result of the `CASE` statement. The `WHEN` clauses are evaluated in order so that if there is more than one matching clause, the result of the first matching clause becomes the result of the `CASE` statement. If there are no matches, `r[e]` of the `ELSE` clause is the result. If there is no match and no `ELSE` clause, the result is `Undefined`.

`CASE` statements require at least one `WHEN` clause. An `ELSE` clause is optional.

For example:

Incoming payload published on topic `topic/subtopic`:

```
{
    "color": "yellow"
}
```

SQL statement:

```
SELECT CASE color
        WHEN 'green' THEN 'go'
        WHEN 'yellow' THEN 'caution'
        WHEN 'red' THEN 'stop'
        ELSE 'you are not at a stop light' END as instructions
FROM 'topic/subtopic'
```

The resulting output payload would be:

```
{
    "instructions": "caution"
}
```

Note

If `v` is `Undefined`, the result of the case statement is `Undefined`.

JSON extensions

You can use the following extensions to ANSI SQL syntax to make it easier to work with nested JSON objects.

`".` Operator

This operator accesses members in embedded JSON objects and functions identically to ANSI SQL and JavaScript. For example:

```
SELECT foo.bar AS bar.baz FROM 'topic/subtopic'
```

selects the value of the `bar` property in the `foo` object from the following message payload sent to the `topic/subtopic` topic.

```
{
  "foo": {
    "bar": "RED",
    "bar1": "GREEN",
    "bar2": "BLUE"
  }
}
```

If a JSON property name includes a hyphen character or numeric characters, the simple 'dot' notation will not work. Instead, you must use the [get function \(p. 555\)](#) to extract the property's value.

In this example the following message is sent to the `iot/rules` topic.

```
{
  "mydata": {
    "item2": {
      "0": {
        "my-key": "myValue"
      }
    }
  }
}
```

Normally, the value of `my-key` would be identified as in this query.

```
SELECT * from iot/rules WHERE mydata.item2.0.my-key= "myValue"
```

However, because the property name `my-key` contains a hyphen and `item2` contains a numeric character, the [get function \(p. 555\)](#) must be used as the following query shows.

```
SELECT * from 'iot/rules' WHERE get(get(get(mydata,"item2"),"0"),"my-key") = "myValue"
```

* Operator

This functions in the same way as the `*` wildcard in ANSI SQL. It's used in the `SELECT` clause only and creates a new JSON object containing the message data. If the message payload is not in JSON format, `*` returns the entire message payload as raw bytes. For example:

```
SELECT * FROM 'topic/subtopic'
```

Applying a Function to an Attribute Value

The following is an example JSON payload that might be published by a device:

```
{
  "deviceid" : "iot123",
  "temp" : 54.98,
  "humidity" : 32.43,
  "coords" : {
    "latitude" : 47.615694,
    "longitude" : -122.3359976
}
```

```
}
```

The following example applies a function to an attribute value in a JSON payload:

```
SELECT temp, md5(deviceid) AS hashed_id FROM topic/#
```

The result of this query is the following JSON object:

```
{
    "temp": 54.98,
    "hashed_id": "e37f81fb397e595c4aeb5645b8cbbbd1"
}
```

Substitution templates

You can use a substitution template to augment the JSON data returned when a rule is triggered and AWS IoT performs an action. The syntax for a substitution template is `#{expression}`, where *expression* can be any expression supported by AWS IoT in SELECT clauses, WHERE clauses, and [AWS IoT rule actions \(p. 450\)](#). This expression can be plugged into an action field on a rule, allowing you to dynamically configure an action. In effect, this feature substitutes a piece of information in an action. This includes functions, operators, and information present in the original message payload.

Important

Because an expression in a substitution template is evaluated separately from the "SELECT ..." statement, you cannot reference an alias created using the AS clause. You can reference only information present in the original payload, [functions \(p. 541\)](#), and [operators \(p. 536\)](#).

For more information about supported expressions, see [AWS IoT SQL reference \(p. 529\)](#).

The following rule actions support substitution templates. Each action supports different fields that can be substituted.

- [Apache Kafka \(p. 452\)](#)
- [CloudWatch alarms \(p. 459\)](#)
- [CloudWatch Logs \(p. 460\)](#)
- [CloudWatch metrics \(p. 461\)](#)
- [DynamoDB \(p. 463\)](#)
- [DynamoDBv2 \(p. 465\)](#)
- [Elasticsearch \(p. 466\)](#)
- [HTTP \(p. 468\)](#)
- [IoT Analytics \(p. 495\)](#)
- [IoT Events \(p. 496\)](#)
- [IoT SiteWise \(p. 498\)](#)
- [Kinesis Data Streams \(p. 503\)](#)
- [Kinesis Data Firehose \(p. 502\)](#)
- [Lambda \(p. 505\)](#)
- [OpenSearch \(p. 507\)](#)
- [Republish \(p. 509\)](#)
- [S3 \(p. 510\)](#)
- [SNS \(p. 512\)](#)

- [SQS \(p. 514\)](#)
- [Step Functions \(p. 515\)](#)
- [Timestream \(p. 516\)](#)

Substitution templates appear in the action parameters within a rule:

```
{
    "sql": "SELECT *, timestamp() AS timestamp FROM 'my/iot/topic'",
    "ruleDisabled": false,
    "actions": [
        {
            "republish": {
                "topic": "${topic()}/republish",
                "roleArn": "arn:aws:iam::123456789012:role/my-iot-role"
            }
        }
    ]
}
```

If this rule is triggered by the following JSON published to `my/iot/topic`:

```
{
    "deviceid": "iot123",
    "temp": 54.98,
    "humidity": 32.43,
    "coords": {
        "latitude": 47.615694,
        "longitude": -122.3359976
    }
}
```

Then this rule publishes the following JSON to `my/iot/topic/republish`, which AWS IoT substitutes from `${topic()}/republish`:

```
{
    "deviceid": "iot123",
    "temp": 54.98,
    "humidity": 32.43,
    "coords": {
        "latitude": 47.615694,
        "longitude": -122.3359976
    },
    "timestamp": 1579637878451
}
```

Nested object queries

You can use nested SELECT clauses to query for attributes within arrays and inner JSON objects. Supported by SQL version 2016-03-23 and later.

Consider the following MQTT message:

```
{
    "e": [
        {
            "n": "temperature", "u": "Cel", "t": 1234, "v": 22.5 },
        {
            "n": "light", "u": "lm", "t": 1235, "v": 135 },
        {
            "n": "acidity", "u": "pH", "t": 1235, "v": 7 }
    ]
}
```

Example

You can convert values to a new array with the following rule.

```
SELECT (SELECT VALUE n FROM e) as sensors FROM 'my/topic'
```

The rule generates the following output.

```
{  
    "sensors": [  
        "temperature",  
        "light",  
        "acidity"  
    ]  
}
```

Example

Using the same MQTT message, you can also query a specific value within a nested object with the following rule.

```
SELECT (SELECT v FROM e WHERE n = 'temperature') as temperature FROM 'my/topic'
```

The rule generates the following output.

```
{  
    "temperature": [  
        {  
            "v": 22.5  
        }  
    ]  
}
```

Example

You can also flatten the output with a more complicated rule.

```
SELECT get((SELECT v FROM e WHERE n = 'temperature'), 0).v as temperature FROM 'topic'
```

The rule generates the following output.

```
{  
    "temperature": 22.5  
}
```

Working with binary payloads

When the message payload should be handled as raw binary data, rather than a JSON object, use the * operator to refer to it in a SELECT clause. This works for non-JSON payloads with some rule actions, such as the [S3 action](#).

Binary payload examples

When you use * to refer to the message payload as raw binary data, you can add data to the rule. If you have an empty or a JSON payload, the resulting payload can have data added using the rule. The following shows examples of supported SELECT clauses.

- You can use the following `SELECT` clauses with only a `*` for binary payloads.

```
• SELECT * FROM 'topic/subtopic'  
• SELECT * FROM 'topic/subtopic' WHERE timestamp() % 12 = 0
```

- You can also add data and use the following `SELECT` clauses.

```
• SELECT *, principal() as principal, timestamp() as time FROM 'topic/subtopic'  
• SELECT encode(*, 'base64') AS data, timestamp() AS ts FROM 'topic/subtopic'
```

- You can also use these `SELECT` clauses with binary payloads.

- The following refers to `device_type` in the `WHERE` clause.

```
SELECT * FROM 'topic/subtopic' WHERE device_type = 'thermostat'
```

- The following is also supported.

```
{  
    "sql": "SELECT * FROM 'topic/subtopic'"  
    "actions": [{  
        "republish": {  
            "topic": "device/${device_id}"  
        }  
    }]  
}
```

The following rule actions don't support binary payloads so you must decode them.

- Some rule actions don't support binary payload input, such as a [Lambda action](#), so you must decode binary payloads. The Lambda rule action can receive binary data, if it's base64 encoded and in a JSON payload. You can do this by changing the rule to the following.

```
SELECT encode(*, 'base64') AS data FROM 'my_topic'
```

- The SQL statement doesn't support string as input. To convert a string input to JSON, you can run the following command.

```
SELECT decode(encode(*, 'base64'), 'base64') AS payload FROM 'topic'
```

SQL versions

The AWS IoT rules engine uses an SQL-like syntax to select data from MQTT messages. The SQL statements are interpreted based on an SQL version specified with the `awsIotSqlVersion` property in a JSON document that describes the rule. For more information about the structure of JSON rule documents, see [Creating a Rule \(p. 446\)](#). The `awsIotSqlVersion` property lets you specify which version of the AWS IoT SQL rules engine that you want to use. When a new version is deployed, you can continue to use an earlier version or change your rule to use the new version. Your current rules continue to use the version with which they were created.

The following JSON example shows you how to specify the SQL version using the `awsIotSqlVersion` property.

```
{
```

```
    "sql": "expression",
    "ruleDisabled": false,
    "awsIotSqlVersion": "2016-03-23",
    "actions": [
        {
            "republish": {
                "topic": "my-mqtt-topic",
                "roleArn": "arn:aws:iam::123456789012:role/my-iot-role"
            }
        }
    ]
}
```

AWS IoT currently supports the following SQL versions:

- 2015-10-08 – The original SQL version built on 2015-10-08.
- 2016-03-23 – The SQL version built on 2016-03-23.
- beta – The most recent beta SQL version. If you use this version, it might introduce breaking changes to your rules.

What's new in the 2016-03-23 SQL rules engine version

- Fixes for selecting nested JSON objects.
- Fixes for array queries.
- Intra-object query support. For more information, see [Nested object queries \(p. 587\)](#).
- Support to output an array as a top-level object.
- Addition of the `encode(value, encodingScheme)` function, which can be applied on JSON and non-JSON format data. For more information, see the [encode function \(p. 553\)](#).

Output an Array as a top-level object

This feature allows a rule to return an array as a top-level object. For example, given the following MQTT message:

```
{
    "a": {"b": "c"},
    "arr": [1, 2, 3, 4]
}
```

And the following rule:

```
SELECT VALUE arr FROM 'topic'
```

The rule generates the following output.

```
[1, 2, 3, 4]
```

AWS IoT Device Shadow service

The AWS IoT Device Shadow service adds shadows to AWS IoT thing objects. Shadows can make a device's state available to apps and other services whether the device is connected to AWS IoT or not. AWS IoT thing objects can have multiple named shadows so that your IoT solution has more options for connecting your devices to other apps and services.

AWS IoT thing objects do not have any named shadows until they are created explicitly; however, an unnamed, classic shadow is created for a thing when the thing is created. Shadows can be created, updated, and deleted by using the AWS IoT console. Devices, other web clients, and services can create, update, and delete shadows by using MQTT and the [reserved MQTT topics \(p. 106\)](#), HTTP using the [Device Shadow REST API \(p. 615\)](#), and the [AWS CLI for AWS IoT](#). Because shadows are stored by AWS in the cloud, they can collect and report device state data from apps and other cloud services whether the device is connected or not.

Topics

- [Using shadows \(p. 591\)](#)
- [Using shadows in devices \(p. 594\)](#)
- [Using shadows in apps and services \(p. 597\)](#)
- [Simulating Device Shadow service communications \(p. 599\)](#)
- [Interacting with shadows \(p. 609\)](#)
- [Device Shadow REST API \(p. 615\)](#)
- [Device Shadow MQTT topics \(p. 619\)](#)
- [Device Shadow service documents \(p. 627\)](#)
- [Device Shadow error messages \(p. 636\)](#)

Using shadows

Shadows provide a reliable data store for devices, apps, and other cloud services to share data. They enable devices, apps, and other cloud services to connect and disconnect without losing a device's state.

While devices, apps, and other cloud services are connected to AWS IoT, they can access and control the current state of a device through its shadows. For example, an app can request a change in a device's state by updating a shadow. AWS IoT publishes a message that indicates the change to the device. The device receives this message, updates its state to match, and publishes a message with its updated state. The Device Shadow service reflects this updated state in the corresponding shadow. The app can subscribe to the shadow's update or it can query the shadow for its current state.

When a device goes offline, an app can still communicate with AWS IoT and the device's shadows. When the device reconnects, it receives the current state of its shadows so that it can update its state to match that of its shadows, and then publish a message with its updated state. Likewise, when an app goes offline and the device state changes while it's offline, the device keeps the shadow updated so the app can query the shadows for its current state when it reconnects.

Choosing to use named or unnamed shadows

The Device Shadow service supports named and unnamed, classic shadows, as have been used in the past. A thing object can have multiple named shadows, and no more than one unnamed, classic shadow. A thing object can have both named and unnamed shadows at the same time; however, the API used to access each is slightly different, so it might be more efficient to decide which type of shadow would work best for your solution and use that type only. For more information about the API to access the shadows, see [Shadow topics \(p. 106\)](#).

With named shadows, you can create different views of a thing object's state. For example, you could divide a thing object with many properties into shadows with logical groups of properties, each identified by its shadow name. You could also limit access to properties by grouping them into different shadows and using policies to control access. Named shadows, however, do not support [fleet indexing \(p. 732\)](#).

The classic, unnamed shadows are simpler, but somewhat more limited than the named shadows. Each AWS IoT thing object can have only one unnamed shadow. If you expect your IoT solution to have a limited need for shadow data, this might be how you want to get started using shadows. However, if you think you might want to add additional shadows in the future, consider using named shadows from the start.

Accessing shadows

Every shadow has a reserved [MQTT topic \(p. 106\)](#) and [HTTP URL \(p. 615\)](#) that supports the get, update, and delete actions on the shadow.

Shadows use [JSON shadow documents \(p. 627\)](#) to store and retrieve data. A shadow's document contains a state property that describes these aspects of the device's state:

- **desired**

Apps specify the desired states of device properties by updating the **desired** object.

- **reported**

Devices report their current state in the **reported** object.

- **delta**

AWS IoT reports differences between the desired and the reported state in the **delta** object.

The data stored in a shadow is determined by the state property of the update action's message body. Subsequent update actions can modify the values of an existing data object, and also add and delete keys and other elements in the shadow's state object. For more information about accessing shadows, see [Using shadows in devices \(p. 594\)](#) and [Using shadows in apps and services \(p. 597\)](#).

Important

Permission to make update requests should be limited to trusted apps and devices. This prevents the shadow's state property from being changed unexpectedly; otherwise, the devices and apps that use the shadow should be designed to expect the keys in the state property to change.

Using shadows in devices, apps, and other cloud services

Using shadows in devices, apps, and other cloud services requires consistency and coordination between all of these. The AWS IoT Device Shadow service stores the shadow state, sends messages when the shadow state changes, and responds to messages that change its state. The devices, apps, and other cloud services in your IoT solution must manage their state and keep it consistent with the device shadow's state.

The shadow state data is dynamic and can be altered by the devices, apps, and other cloud services with permission to access the shadow. For this reason, it is important to consider how each device, app, and other cloud service will interact with the shadow. For example:

- *Devices* should write only to the **reported** property of the shadow state when communicating state data to the shadow.

- *Apps and other cloud services* should write only to the `desired` property when communicating state change requests to the device through the shadow.

Important

The data contained in a shadow data object is independent from that of other shadows and other thing object properties, such as a thing's attributes and the content of MQTT messages that a thing object's device might publish. A device can, however, report the same data in different MQTT topics and shadows if necessary.

A device that supports multiple shadows must maintain the consistency of the data that it reports in the different shadows.

Message order

There is no guarantee that messages from the AWS IoT service will arrive at the device in any specific order. The following scenario shows what happens in this case.

Initial state document:

```
{  
  "state": {  
    "reported": {  
      "color": "blue"  
    }  
  },  
  "version": 9,  
  "timestamp": 123456776  
}
```

Update 1:

```
{  
  "state": {  
    "desired": {  
      "color": "RED"  
    }  
  },  
  "version": 10,  
  "timestamp": 123456777  
}
```

Update 2:

```
{  
  "state": {  
    "desired": {  
      "color": "GREEN"  
    }  
  },  
  "version": 11,  
  "timestamp": 123456778  
}
```

Final state document:

```
{  
  "state": {  
    "reported": {  
      "color": "GREEN"  
    }  
  }  
}
```

```

    },
    "version": 12,
    "timestamp": 123456779
}

```

This results in two delta messages:

```
{
  "state": {
    "color": "RED"
  },
  "version": 11,
  "timestamp": 123456778
}
```

```
{
  "state": {
    "color": "GREEN"
  },
  "version": 12,
  "timestamp": 123456779
}
```

The device might receive these messages out of order. Because the state in these messages is cumulative, a device can safely discard any messages that contain a version number older than the one it is tracking. If the device receives the delta for version 12 before version 11, it can safely discard the version 11 message.

Trim shadow messages

To reduce the size of shadow messages sent to your device, define a rule that selects only the fields your device needs then republishes the message on an MQTT topic to which your device is listening.

The rule is specified in JSON and should look like the following:

```
{
  "sql": "SELECT state, version FROM '$aws/things/+shadow/update/delta'",
  "ruleDisabled": false,
  "actions": [
    {
      "republish": {
        "topic": "${topic(3)}/delta",
        "roleArn": "arn:aws:iam:123456789012:role/my-iot-role"
      }
    }
  ]
}
```

The SELECT statement determines which fields from the message will be republished to the specified topic. A "+" wild card is used to match all shadow names. The rule specifies that all matching messages should be republished to the specified topic. In this case, the "topic()" function is used to specify the topic on which to republish. topic(3) evaluates to the thing name in the original topic. For more information about creating rules, see [Rules for AWS IoT \(p. 443\)](#).

Using shadows in devices

This section describes device communications with shadows using MQTT messages, the preferred method for devices to communicate with the AWS IoT Device Shadow service.

Shadow communications emulate a request/response model using the publish/subscribe communication model of MQTT. Every shadow action consists of a request topic, a successful response topic (accepted), and an error response topic (rejected).

If you want apps and services to be able to determine whether a device is connected, see [Detecting a device is connected \(p. 598\)](#).

Important

Because MQTT uses a publish/subscribe communication model, you should subscribe to the response topics *before* you publish a request topic. If you don't, you might not receive the response to the request that you publish.

If you use an [AWS IoT Device SDK \(p. 1127\)](#) to call the Device Shadow service APIs, this is handled for you.

The examples in this section use an abbreviated form of the topic where the *ShadowTopicPrefix* can refer to either a named or an unnamed shadow, as described in this table.

Shadows can be named or unnamed (classic). The topics used by each differ only in the topic prefix. This table shows the topic prefix used by each shadow type.

<i>ShadowTopicPrefix</i> value	Shadow type
\$aws/things/ <i>thingName</i> /shadow	Unnamed (classic) shadow
\$aws/things/ <i>thingName</i> /shadow/ name/ <i>shadowName</i>	Named shadow

Important

Make sure that your app's or service's use of the shadows is consistent and supported by the corresponding implementations in your devices. For example, consider how shadows are created, updated, and deleted. Also consider how updates are handled in the device and the apps or services that access the device through a shadow. Your design should be clear about how the device's state is updated and reported and how your apps and services interact with the device and its shadows.

To create a complete topic, select the *ShadowTopicPrefix* for the type of shadow to which you want to refer, replace *thingName*, and *shadowName* if applicable, with their corresponding values, and then append that with the topic stub as shown in the following table. Remember that topics are case sensitive.

See [Shadow topics \(p. 106\)](#) for more information about the reserved topics for shadows.

Initializing the device on first connection to AWS IoT

After a device registers with AWS IoT, it should subscribe to these MQTT messages for the shadows that it supports.

Topic	Meaning	Action a device should take when this topic is received
<i>ShadowTopicPrefix</i> / delete/accepted	The delete request was accepted and AWS IoT deleted the shadow.	The actions necessary to accommodate the deleted shadow, such as stop publishing updates.

Topic	Meaning	Action a device should take when this topic is received
<i>ShadowTopicPrefix/</i> delete/rejected	The delete request was rejected by AWS IoT and the shadow was not deleted. The message body contains the error information.	Respond to the error message in the message body.
<i>ShadowTopicPrefix/get/</i> accepted	The get request was accepted by AWS IoT, and the message body contains the current shadow document.	The actions necessary to process the state document in the message body.
<i>ShadowTopicPrefix/get/</i> rejected	The get request was rejected by AWS IoT, and the message body contains the error information.	Respond to the error message in the message body.
<i>ShadowTopicPrefix/update/</i> accepted	The update request was accepted by AWS IoT, and the message body contains the current shadow document.	Confirm the updated data in the message body matches the device state.
<i>ShadowTopicPrefix/</i> update/rejected	The update request was rejected by AWS IoT, and the message body contains the error information.	Respond to the error message in the message body.
<i>ShadowTopicPrefix/</i> update/delta	The shadow document was updated by a request to AWS IoT, and the message body contains the changes requested.	Update the device's state to match the desired state in the message body.
<i>ShadowTopicPrefix/</i> update/documents	An update to the shadow was recently completed, and the message body contains the current shadow document.	Confirm the updated state in the message body matches the device's state.

After subscribing to the messages in the preceding table for each shadow, the device should test to see if the shadows that it supports have already been created by publishing a /get topic to each shadow. If a /get/accepted message is received, the message body contains the shadow document, which the device can use to initialize its state. If a /get/rejected message is received, the shadow should be created by publishing an /update message with the current device state.

Processing messages while the device is connected to AWS IoT

While a device is connected to AWS IoT, it can receive **/update/delta** messages and should keep the device state matched to the changes in its shadows by:

1. Reading all **/update/delta** messages received and synchronizing the device state to match.
2. Publishing an **/update** message with a **reported** message body that has the device's current state, whenever the device's state changes.

While a device is connected, it should publish these messages when indicated.

Indication	Topic	Payload
The device's state has changed.	<code>ShadowTopicPrefix/update</code>	A shadow document with the reported property.
The device might not be synchronized with the shadow.	<code>ShadowTopicPrefix/get</code>	(empty)
An action on the device indicates that a shadow will no longer be supported by the device, such as when the device is being removed or replaced	<code>ShadowTopicPrefix/delete</code>	(empty)

Processing messages when the device reconnects to AWS IoT

When a device with one or more shadows connects to AWS IoT, it should synchronize its state with that of all the shadows that it supports by:

1. Reading all `/update/delta` messages received and synchronizing the device state to match.
2. Publishing an `/update` message with a `reported` message body that has the device's current state.

Using shadows in apps and services

This section describes how an app or service interacts with the AWS IoT Device Shadow service. This example assumes the app or service is interacting only with the shadow and, through the shadow, the device. This example doesn't include any management actions, such as creating or deleting shadows.

This example uses the AWS IoT Device Shadow service's REST API to interact with shadows. Unlike the example used in [Using shadows in devices \(p. 594\)](#), which uses a publish/subscribe communications model, this example uses the request/response communications model of the REST API. This means the app or service must make a request before it can receive a response from AWS IoT. A disadvantage of this model, however, is that it does not support notifications. If your app or service requires timely notifications of device state changes, consider the MQTT or MQTT over WSS protocols, which support the publish/subscribe communication model, as described in [Using shadows in devices \(p. 594\)](#).

Important

Make sure that your app's or service's use of the shadows is consistent with and supported by the corresponding implementations in your devices. Consider, for example, how shadows are created, updated, and deleted, and how updates are handled in the device and the apps or services that access the shadow. Your design should clearly specify how the device's state is updated and reported, and how your apps and services interact with the device and its shadows.

The REST API's URL for a named shadow is:

```
https://endpoint/things/thingName/shadow?name=shadowName
```

and for an unnamed shadow:

```
https://endpoint/things/thingName/shadow
```

where:

endpoint

The endpoint returned by the CLI command:

```
aws iot describe-endpoint --endpoint-type IOT:Data-ATS
```

thingName

The name of the thing object to which the shadow belongs

shadowName

The name of the named shadow. This parameter is not used with unnamed shadows.

Initializing the app or service on connection to AWS IoT

When the app first connects to AWS IoT, it should send an HTTP GET request to the URLs of the shadows it uses to get the current state of the shadows it's using. This allows it to sync the app or service to the shadow.

Processing state changes while the app or service is connected to AWS IoT

While the app or service is connected to AWS IoT, it can query the current state periodically by sending an HTTP GET request on the URLs of the shadows it uses.

When an end user interacts with the app or service to change the state of the device, the app or service can send an HTTP POST request to the URLs of the shadows it uses to update the desired state of the shadow. This request returns the change that was accepted, but you might have to poll the shadow by making HTTP GET requests until the device has updated the shadow with its new state.

Detecting a device is connected

To determine if a device is currently connected, include a `connected` property in the shadow document and use an MQTT Last Will and Testament (LWT) message to set the `connected` property to `false` if a device is disconnected due to an error.

Note

MQTT LWT messages sent to AWS IoT reserved topics (topics that begin with \$) are ignored by the AWS IoT Device Shadow service. However, they are processed by subscribed clients and by the AWS IoT rules engine, so you will need to create an LWT message that is sent to a non-reserved topic and a rule that republishes the MQTT LWT message as a shadow update message to the shadow's reserved update topic, `ShadowTopicPrefix/update`.

To send the Device Shadow service an LWT message

1. Create a rule that republishes the MQTT LWT message on the reserved topic. The following example is a rule that listens for a messages on the `my/things/myLightBulb/update` topic and republishes it to `$aws/things/myLightBulb/shadow/update`.

```
{  
  "rule": {  
    "ruleDisabled": false,  
    "topic": "my/things/myLightBulb/update"  
  },  
  "rule": {  
    "ruleDisabled": false,  
    "topic": "$aws/things/myLightBulb/shadow/update"  
  }  
}
```

```

    "sql": "SELECT * FROM 'my/things/myLightBulb/update'",
    "description": "Turn my/things/ into $aws/things/",
    "actions": [
        {
            "republish": {
                "topic": "$$aws/things/myLightBulb/shadow/update",
                "roleArn": "arn:aws:iam:123456789012:role/aws_iot_republish"
            }
        }
    ]
}

```

- When the device connects to AWS IoT, it registers an LWT message to a non-reserved topic for the republish rule to recognize. In this example, that topic is `my/things/myLightBulb/update` and it sets the `connected` property to `false`.

```

{
    "state": {
        "reported": {
            "connected": "false"
        }
    }
}

```

- After connecting, the device publishes a message on its shadow update topic, `$aws/things/myLightBulb/shadow/update`, to report its current state, which includes setting its `connected` property to `true`.

```

{
    "state": {
        "reported": {
            "connected": "true"
        }
    }
}

```

- Before the device disconnects gracefully, it publishes a message on its shadow update topic, `$aws/things/myLightBulb/shadow/update`, to report its latest state, which include setting its `connected` property to `false`.

```

{
    "state": {
        "reported": {
            "connected": "false"
        }
    }
}

```

- If the device disconnects due to an error, the AWS IoT message broker publishes the device's LWT message on behalf of the device. The republish rule detects this message and publishes the shadow update message to update the `connected` property of the device shadow.

Simulating Device Shadow service communications

This topic demonstrates how the Device Shadow service acts as an intermediary and allows devices and apps to use a shadow to update, store, and retrieve a device's state.

To demonstrate the interaction described in this topic, and to explore it further, you'll need an AWS account and a system on which you can run the AWS CLI. If you don't have these, you can still see the interaction in the code examples.

In this example, the AWS IoT console represents the device. The AWS CLI represents the app or service that accesses the device by way of the shadow. The AWS CLI interface is very similar to the API that an app might use to communicate with AWS IoT. The device in this example is a smart light bulb and the app displays the light bulb's state and can change the light bulb's state.

Setting up the simulation

These procedures initialize the simulation by opening the [AWS IoT console](#), which simulates your device, and the command line window that simulates your app.

To set up your simulation environment

1. Create an AWS account or, if you already have one to use for this simulation, you can skip this step.
You'll need an AWS account to run the examples from this topic on your own. If you don't have an AWS account, create one, as described in [Set up your AWS account \(p. 17\)](#).
2. Open the [AWS IoT console](#), and in the left menu, choose **Test** to open the **MQTT client**.
3. In another window, open a terminal window on a system that has the AWS CLI installed on it.

You should have two windows open: one with the AWS IoT console on the **Test** page, and one with a command line prompt.

Initialize the device

In this simulation, we'll be working with a thing object named, *mySimulatedThing*, and its shadow named, *simShadow1*.

Create thing object and its named shadow

To create a thing object, in the **AWS IoT Console**:

1. Choose **Manage** and then choose **Things**.
2. Click the **Create** button if things are listed otherwise click **Register a single thing>** to create a single AWS IoT thing.
3. Enter the name *mySimulatedThing*, leave other settings to default, and then click **Next**.
4. Use one-click certificate creation to generate the certificates that will authenticate the device's connection to AWS IoT. Click **Activate** to activate the certificate.
5. You can attach the policy **My_IoT_Policy** that would give the device permission to publish and subscribe to the MQTT reserved topics. For more detailed steps about how to create an AWS IoT thing and how to create this policy, see [Create a thing object \(p. 37\)](#).

By default, every AWS IoT thing object has a single, classic shadow. You can create a shadow by publishing an update request to the topic `$aws/things/mySimulatedThing/shadow/name/simShadow1/update` as described below. Alternatively, in the **AWS IoT Console**, choose your thing object in the list of things displayed and then choose **Shadows**. Choose **Add a shadow**, enter the name *simShadow1*, and then choose **Create** to add the named shadow.

Subscribe and publish to reserved MQTT topics

In the console, subscribe to the reserved MQTT shadow topics. These topics are the responses to the get, update, and delete actions so that your device will be ready to receive the responses after it publishes an action.

To subscribe to an MQTT topic in the MQTT client

1. In the **MQTT client**, choose **Subscribe to topic**.
2. Enter the get, update, and delete topics to subscribe to. Copy one topic at a time from the following list, paste it in the **Topic filter** field, and then click **Subscribe**. You should see the topics appear under **Subscriptions**.
 - \$aws/things/mySimulatedThing/shadow/name/simShadow1/delete/accepted
 - \$aws/things/mySimulatedThing/shadow/name/simShadow1/delete/rejected
 - \$aws/things/mySimulatedThing/shadow/name/simShadow1/get/accepted
 - \$aws/things/mySimulatedThing/shadow/name/simShadow1/get/rejected
 - \$aws/things/mySimulatedThing/shadow/name/simShadow1/update/accepted
 - \$aws/things/mySimulatedThing/shadow/name/simShadow1/update/rejected
 - \$aws/things/mySimulatedThing/shadow/name/simShadow1/update/delta
 - \$aws/things/mySimulatedThing/shadow/name/simShadow1/update/documents

At this point, your simulated device is ready to receive the topics as they are published by AWS IoT.

To publish to an MQTT topic in the MQTT client

After a device has initialized itself and subscribed to the response topics, it should query for the shadows it supports. This simulation supports only one shadow, the shadow that supports a thing object named, *mySimulatedThing*, named, *simShadow1*.

To get the current shadow state from the MQTT client

1. In the **MQTT client**, choose **Publish to a topic**.
2. Under **Publish**, enter the following topic and delete any content from the message body window below where you entered the topic to get. You can then choose **Publish to topic** to publish the request. \$aws/things/mySimulatedThing/shadow/name/simShadow1/get.

If you haven't created the named shadow, *simShadow1*, you receive a message in the \$aws/things/mySimulatedThing/shadow/name/simShadow1/get/rejected topic and the code is 404, such as in this example as the shadow has not been created, so we'll create it next.

```
{  
  "code": 404,  
  "message": "No shadow exists with name: 'simShadow1'"  
}
```

To create a shadow with the current status of the device

1. In the **MQTT client**, choose **Publish to a topic** and enter this topic:

```
$aws/things/mySimulatedThing/shadow/name/simShadow1/update
```

2. In the message body window below where you entered the topic, enter this shadow document to show the device is reporting its ID and its current color in RGB values. Choose **Publish** to publish the request.

```
{  
  "state": {  
    "reported": {  
      "ID": "SmartLamp21",  
      "color": {  
        "r": 255,  
        "g": 0,  
        "b": 0  
      }  
    }  
  }  
}
```

```
        "ColorRGB": [
            128,
            128,
            128
        ]
    },
    "clientToken": "426bfd96-e720-46d3-95cd-014e3ef12bb6"
}
```

If you receive a message in the topic:

- `$aws/things/mySimulatedThing/shadow/name/simShadow1/update/accepted`: It means that the shadow was created and the message body contains the current shadow document.
- `$aws/things/mySimulatedThing/shadow/name/simShadow1/update/rejected`: Review the error in the message body.
- `$aws/things/mySimulatedThing/shadow/name/simShadow1/get/accepted`: The shadow already exists and the message body has the current shadow state, such as in this example. With this, you could set your device or confirm that it matches the shadow state.

```
{
    "state": {
        "reported": {
            "ID": "SmartLamp21",
            "ColorRGB": [
                128,
                128,
                128
            ]
        }
    },
    "metadata": {
        "reported": {
            "ID": {
                "timestamp": 1591140517
            },
            "ColorRGB": [
                {
                    "timestamp": 1591140517
                },
                {
                    "timestamp": 1591140517
                },
                {
                    "timestamp": 1591140517
                }
            ]
        }
    },
    "version": 3,
    "timestamp": 1591140517,
    "clientToken": "426bfd96-e720-46d3-95cd-014e3ef12bb6"
}
```

Send an update from the app

This section uses the AWS CLI to demonstrate how an app can interact with a shadow.

To get the current state of the shadow using the AWS CLI

From the command line, enter this command.

```
aws iot-data get-thing-shadow --thing-name mySimulatedThing --shadow-name simShadow1 /dev/stdout
```

On Windows platforms, you can use con instead of /dev/stdout.

```
aws iot-data get-thing-shadow --thing-name mySimulatedThing --shadow-name simShadow1 con
```

Because the shadow exists and had been initialized by the device to reflect its current state, it should return the following shadow document.

```
{
  "state": {
    "reported": {
      "ID": "SmartLamp21",
      "ColorRGB": [
        128,
        128,
        128
      ]
    }
  },
  "metadata": {
    "reported": {
      "ID": {
        "timestamp": 1591140517
      },
      "ColorRGB": [
        {
          "timestamp": 1591140517
        },
        {
          "timestamp": 1591140517
        },
        {
          "timestamp": 1591140517
        }
      ]
    },
    "version": 3,
    "timestamp": 1591141111
  }
}
```

The app can use this response to initialize its representation of the device state.

If the app updates the state, such as when an end user changes the color of our smart light bulb to yellow, the app would send an **update-thing-shadow** command. This command corresponds to the UpdateThingShadow REST API.

To update a shadow from an app

From the command line, enter this command.

AWS CLI v2.x

```
aws iot-data update-thing-shadow --thing-name mySimulatedThing --shadow-name simShadow1 \
  --cli-binary-format raw-in-base64-out \
```

```
--payload '{"state":{"desired":{"ColorRGB":[255,255,0]}}, "clientToken": "21b21b21-bfd2-4279-8c65-e2f697ff4fab"}' /dev/stdout
```

AWS CLI v1.x

```
aws iot-data update-thing-shadow --thing-name mySimulatedThing --shadow-name simShadow1
 \
 --payload '{"state":{"desired":{"ColorRGB":[255,255,0]}}, "clientToken": "21b21b21-bfd2-4279-8c65-e2f697ff4fab"}' /dev/stdout
```

If successful, this command should return the following shadow document.

```
{
  "state": {
    "desired": {
      "ColorRGB": [
        255,
        255,
        0
      ]
    }
  },
  "metadata": {
    "desired": {
      "ColorRGB": [
        {
          "timestamp": 1591141596
        },
        {
          "timestamp": 1591141596
        },
        {
          "timestamp": 1591141596
        }
      ]
    }
  },
  "version": 4,
  "timestamp": 1591141596,
  "clientToken": "21b21b21-bfd2-4279-8c65-e2f697ff4fab"
}
```

Respond to update in device

Returning to the **MQTT client** in the AWS console, you should see the messages that AWS IoT published to reflect the update command issued in the previous section.

To view the update messages in the MQTT client

In the **MQTT client**, choose **\$aws/things/mySimulatedThing/shadow/name/simShadow1/update/delta** in the **Subscriptions** column. If the topic name is truncated, you can pause on it to see the full topic. In the topic log of this topic, you should see a /delta message similar to this one.

```
{
  "version": 4,
  "timestamp": 1591141596,
  "state": {
    "ColorRGB": [
      255,
      255,
```

```

        0
    ],
},
"metadata": {
    "ColorRGB": [
        {
            "timestamp": 1591141596
        },
        {
            "timestamp": 1591141596
        },
        {
            "timestamp": 1591141596
        }
    ]
},
"clientToken": "21b21b21-bfd2-4279-8c65-e2f697ff4fab"
}

```

Your device would process the contents of this message to set the device state to match the desired state in the message.

After the device updates the state to match the desired state in the message, it must send the new reported state back to AWS IoT by publishing an update message. This procedure simulates this in the **MQTT client**.

To update the shadow from the device

1. In the **MQTT client**, choose **Publish to a topic**.
2. In the message body window , in the topic field above the message body window, enter the shadow's topic followed by the /update action: `$aws/things/mySimulatedThing/shadow/name/simShadow1/update` and in the message body, enter this updated shadow document, which describes the current state of the device. Click **Publish** to publish the updated device state.

```
{
    "state": {
        "reported": {
            "ColorRGB": [255,255,0]
        }
    },
    "clientToken": "a4dc2227-9213-4c6a-a6a5-053304f60258"
}
```

If the message was successfully received by AWS IoT, you should see a new response in the `$aws/things/mySimulatedThing/shadow/name/simShadow1/update/accepted` message log in the **MQTT client** with the current state of the shadow, such as this example.

```
{
    "state": {
        "reported": {
            "ColorRGB": [
                255,
                255,
                0
            ]
        }
    },
    "metadata": {
        "reported": {
            "ColorRGB": [
                {

```

```

        "timestamp": 1591142747
    },
    {
        "timestamp": 1591142747
    },
    {
        "timestamp": 1591142747
    }
]
}
},
"version": 5,
"timestamp": 1591142747,
"clientToken": "a4dc2227-9213-4c6a-a6a5-053304f60258"
}
```

A successful update to the reported state of the device also causes AWS IoT to send a comprehensive description of the shadow state in a message to the topic, such as this message body that resulted from the shadow update performed by the device in the preceding procedure.

```
{
  "previous": {
    "state": {
      "desired": {
        "ColorRGB": [
          255,
          255,
          0
        ]
      },
      "reported": {
        "ID": "SmartLamp21",
        "ColorRGB": [
          128,
          128,
          128
        ]
      }
    },
    "metadata": {
      "desired": {
        "ColorRGB": [
          {
            "timestamp": 1591141596
          },
          {
            "timestamp": 1591141596
          },
          {
            "timestamp": 1591141596
          }
        ]
      },
      "reported": {
        "ID": {
          "timestamp": 1591140517
        },
        "ColorRGB": [
          {
            "timestamp": 1591140517
          },
          {
            "timestamp": 1591140517
          }
        ]
      }
    }
  }
}
```

```

        },
        {
          "timestamp": 1591140517
        }
      ]
    }
  },
  "version": 4
},
"current": {
  "state": {
    "desired": {
      "ColorRGB": [
        255,
        255,
        0
      ]
    },
    "reported": {
      "ID": "SmartLamp21",
      "ColorRGB": [
        255,
        255,
        0
      ]
    }
  },
  "metadata": {
    "desired": {
      "ColorRGB": [
        {
          "timestamp": 1591141596
        },
        {
          "timestamp": 1591141596
        },
        {
          "timestamp": 1591141596
        }
      ]
    },
    "reported": {
      "ID": {
        "timestamp": 1591140517
      },
      "ColorRGB": [
        {
          "timestamp": 1591142747
        },
        {
          "timestamp": 1591142747
        },
        {
          "timestamp": 1591142747
        }
      ]
    }
  },
  "version": 5
},
"timestamp": 1591142747,
"clientToken": "a4dc2227-9213-4c6a-a6a5-053304f60258"
}

```

Observe the update in the app

The app can now query the shadow for the current state as reported by the device.

To get the current state of the shadow using the AWS CLI

1. From the command line, enter this command.

```
aws iot-data get-thing-shadow --thing-name mySimulatedThing --shadow-name simShadow1 /  
dev/stdout
```

On Windows platforms, you can use `con` instead of `/dev/stdout`.

```
aws iot-data get-thing-shadow --thing-name mySimulatedThing --shadow-name simShadow1  
con
```

2. Because the shadow has just been updated by the device to reflect its current state, it should return the following shadow document.

```
{  
    "state": {  
        "desired": {  
            "ColorRGB": [  
                255,  
                255,  
                0  
            ]  
        },  
        "reported": {  
            "ID": "SmartLamp21",  
            "ColorRGB": [  
                255,  
                255,  
                0  
            ]  
        }  
    },  
    "metadata": {  
        "desired": {  
            "ColorRGB": [  
                {  
                    "timestamp": 1591141596  
                },  
                {  
                    "timestamp": 1591141596  
                },  
                {  
                    "timestamp": 1591141596  
                }  
            ]  
        },  
        "reported": {  
            "ID": {  
                "timestamp": 1591140517  
            },  
            "ColorRGB": [  
                {  
                    "timestamp": 1591142747  
                },  
                {  
                    "timestamp": 1591142747  
                },  
                {  
                    "timestamp": 1591142747  
                }  
            ]  
        }  
    }  
}
```

```
{  
    "timestamp": 1591142747  
}  
]  
}  
,  
"version": 5,  
"timestamp": 1591143269  
}
```

Going beyond the simulation

Experiment with the interaction between the AWS CLI (representing the app) and the console (representing the device) to model your IoT solution.

Interacting with shadows

This topic describes the messages associated with each of the three methods that AWS IoT provides for working with shadows. These methods include the following:

UPDATE

Creates a shadow if it doesn't exist, or updates the contents of an existing shadow with the state information provided in the message body. AWS IoT records a timestamp with each update to indicate when the state was last updated. When the shadow's state changes, AWS IoT sends `/delta` messages to all MQTT subscribers with the difference between the desired and the reported states. Devices or apps that receive a `/delta` message can perform actions based on the difference. For example, a device can update its state to the desired state, or an app can update its UI to reflect the device's state change.

GET

Retrieves a current shadow document that contains the complete state of the shadow, including metadata.

DELETE

Deletes the device shadow and its content.

You can't restore a deleted device shadow document, but you can create a new device shadow with the name of a deleted device shadow document. If you create a device shadow document that has the same name as one that was deleted within the past 48 hours, the version number of the new device shadow document will follow that of the deleted one. If a device shadow document has been deleted for more than 48 hours, the version number of a new device shadow document with the same name will be 0.

Protocol support

AWS IoT supports [MQTT](#) and a REST API over HTTPS protocols to interact with shadows. AWS IoT provides a set of reserved request and response topics for MQTT publish and subscribe actions. Devices and apps should subscribe to the response topics before publishing a request topic for information about how AWS IoT handled the request. For more information, see [Device Shadow MQTT topics \(p. 619\)](#) and [Device Shadow REST API \(p. 615\)](#).

Requesting and reporting state

When designing your IoT solution using AWS IoT and shadows, you should determine the apps or devices that will request changes and those that will implement them. Typically, a device implements and reports changes back to the shadow and apps and services respond to and request changes in the shadow. Your solution could be different, but the examples in this topic assume that the client app or service requests changes in the shadow and the device performs the changes and reports them back to the shadow.

Updating a shadow

Your app or service can update a shadow's state by using the [UpdateThingShadow \(p. 617\)](#) API or by publishing to the [/update \(p. 622\)](#) topic. Updates affect only the fields specified in the request.

Updating a shadow when a client requests a state change

When a client requests a state change in a shadow by using the MQTT protocol

1. The client should have a current shadow document so that it can identify the properties to change. See the /get action for how to obtain the current shadow document.
2. The client subscribes to these MQTT topics:
 - \$aws/things/*thingName*/shadow/name/*shadowName*/update/accepted
 - \$aws/things/*thingName*/shadow/name/*shadowName*/update/rejected
 - \$aws/things/*thingName*/shadow/name/*shadowName*/update/delta
 - \$aws/things/*thingName*/shadow/name/*shadowName*/update/documents
3. The client publishes a \$aws/things/*thingName*/shadow/name/*shadowName*/update request topic with a state document that contains the desired state of the shadow. Only the properties to change need to be included in the document. This is an example of a document with the desired state.

```
{  
  "state": {  
    "desired": {  
      "color": {  
        "r": 10  
      },  
      "engine": "ON"  
    }  
  }  
}
```

4. If the update request is valid, AWS IoT updates the desired state in the shadow and publishes messages on these topics:
 - \$aws/things/*thingName*/shadow/name/*shadowName*/update/accepted
 - \$aws/things/*thingName*/shadow/name/*shadowName*/update/delta

The /update/accepted message contains an [/accepted response state document \(p. 629\)](#) shadow document, and the /update/delta message contains a [/delta response state document \(p. 629\)](#) shadow document.

5. If the update request is not valid, AWS IoT publishes a message with the \$aws/things/*thingName*/shadow/name/*shadowName*/update/rejected topic with an [Error response document \(p. 631\)](#) shadow document that describes the error.

When a client requests a state change in a shadow by using the API

1. The client calls the [UpdateThingShadow \(p. 617\)](#) API with a [Request state document \(p. 628\)](#) state document as its message body.
2. If the request was valid, AWS IoT returns an HTTP success response code and an [/accepted response state document \(p. 629\)](#) shadow document as its response message body.
AWS IoT will also publish an MQTT message to the `$aws/things/thingName/shadow/name/shadowName/update/delta` topic with a [/delta response state document \(p. 629\)](#) shadow document for any devices or clients that subscribe to it.
3. If the request was not valid, AWS IoT returns an HTTP error response code an [Error response document \(p. 631\)](#) as its response message body.

When the device receives the /desired state on the /update/delta topic, it makes the desired changes in the device. It then sends a message to the /update topic to report its current state to the shadow.

Updating a shadow when a device reports its current state

When a device reports its current state to the shadow by using the MQTT protocol

1. The device should subscribe to these MQTT topics before updating the shadow:
 - `$aws/things/thingName/shadow/name/shadowName/update/accepted`
 - `$aws/things/thingName/shadow/name/shadowName/update/rejected`
 - `$aws/things/thingName/shadow/name/shadowName/update/delta`
 - `$aws/things/thingName/shadow/name/shadowName/update/documents`
2. The device reports its current state by publishing a message to the `$aws/things/thingName/shadow/name/shadowName/update` topic that reports the current state, such as in this example.

```
{  
    "state": {  
        "reported" : {  
            "color" : { "r" : 10 },  
            "engine" : "ON"  
        }  
    }  
}
```

3. If AWS IoT accepts the update, it publishes a message to the `$aws/things/thingName/shadow/name/shadowName/update/accepted` topics with an [/accepted response state document \(p. 629\)](#) shadow document.
4. If the update request is not valid, AWS IoT publishes a message with the `$aws/things/thingName/shadow/name/shadowName/update/rejected` topic with an [Error response document \(p. 631\)](#) shadow document that describes the error.

When a device reports its current state to the shadow by using the API

1. The device calls the [UpdateThingShadow \(p. 617\)](#) API with a [Request state document \(p. 628\)](#) state document as its message body.
2. If the request was valid, AWS IoT updates the shadow and returns an HTTP success response code with an [/accepted response state document \(p. 629\)](#) shadow document as its response message body.

AWS IoT will also publish an MQTT message to the `$aws/things/thingName/shadow/name/shadowName/update/delta` topic with a [/delta response state document \(p. 629\)](#) shadow document for any devices or clients that subscribe to it.

3. If the request was not valid, AWS IoT returns an HTTP error response code an [Error response document \(p. 631\)](#) as its response message body.

Optimistic locking

You can use the state document version to ensure you are updating the most recent version of a device's shadow document. When you supply a version with an update request, the service rejects the request with an HTTP 409 conflict response code if the current version of the state document does not match the version supplied.

For example:

Initial document:

```
{  
  "state": {  
    "desired": {  
      "colors": [  
        "RED",  
        "GREEN",  
        "BLUE"  
      ]  
    }  
  },  
  "version": 10  
}
```

Update: (version doesn't match; this request will be rejected)

```
{  
  "state": {  
    "desired": {  
      "colors": [  
        "BLUE"  
      ]  
    }  
  },  
  "version": 9  
}
```

Result:

```
{  
  "code": 409,  
  "message": "Version conflict",  
  "clientToken": "426bfd96-e720-46d3-95cd-014e3ef12bb6"  
}
```

Update: (version matches; this request will be accepted)

```
{  
  "state": {  
    "desired": {  
      "colors": [  

```

```
        "BLUE"
    ]
},
"version": 10
}
```

Final state:

```
{
  "state": {
    "desired": {
      "colors": [
        "BLUE"
      ]
    }
  },
  "version": 11
}
```

Retrieving a shadow document

You can retrieve a shadow document by using the [GetThingShadow \(p. 616\)](#) API or by subscribing and publishing to the [/get \(p. 620\)](#) topic. This retrieves a complete shadow document, including any delta between the desired and reported states. The procedure for this task is the same whether the device or a client is making the request.

To retrieve a shadow document by using the MQTT protocol

1. The device or client should subscribe to these MQTT topics before updating the shadow:
 - \$aws/things/*thingName*/shadow/name/*shadowName*/get/accepted
 - \$aws/things/*thingName*/shadow/name/*shadowName*/get/rejected
2. The device or client publishes a message to the \$aws/things/*thingName*/shadow/name/*shadowName*/get topic with an empty message body.
3. If the request is successful, AWS IoT publishes a message to the \$aws/things/*thingName*/shadow/name/*shadowName*/get/accepted topic with a [/accepted response state document \(p. 629\)](#) in the message body.
4. If the request was not valid, AWS IoT publishes a message to the \$aws/things/*thingName*/shadow/name/*shadowName*/get/rejected topic with an [Error response document \(p. 631\)](#) in the message body.

To retrieve a shadow document by using a REST API

1. The device or client call the [GetThingShadow \(p. 616\)](#) API with an empty message body.
2. If the request is valid, AWS IoT returns an HTTP success response code with an [/accepted response state document \(p. 629\)](#) shadow document as its response message body.
3. If the request is not valid, AWS IoT returns an HTTP error response code an [Error response document \(p. 631\)](#) as its response message body.

Deleting shadow data

There are two ways to delete shadow data: you can delete specific properties in the shadow document and you can delete the shadow completely.

- To delete specific properties from a shadow, update the shadow; however set the value of the properties that you want to delete to `null`. Fields with a value of `null` are removed from the shadow document.
- To delete the entire shadow, use the [DeleteThingShadow \(p. 618\)](#) API or publish to the [/delete \(p. 626\)](#) topic.

Note that deleting a shadow does not reset its version number to 0.

Deleting a property from a shadow document

To delete a property from a shadow by using the MQTT protocol

1. The device or client should have a current shadow document so that it can identify the properties to change. See [Retrieving a shadow document \(p. 613\)](#) for information on how to obtain the current shadow document.
2. The device or client subscribes to these MQTT topics:
 - `$aws/things/thingName/shadow/name/shadowName/update/accepted`
 - `$aws/things/thingName/shadow/name/shadowName/update/rejected`
3. The device or client publishes a `$aws/things/thingName/shadow/name/shadowName/update` request topic with a state document that assigns `null` values to the properties of the shadow to delete. Only the properties to change need to be included in the document. This is an example of a document that deletes the `engine` property.

```
{  
  "state": {  
    "desired": {  
      "engine": null  
    }  
  }  
}
```

4. If the update request is valid, AWS IoT deletes the specified properties in the shadow and publishes a message with the `$aws/things/thingName/shadow/name/shadowName/update/accepted` topic with an [/accepted response state document \(p. 629\)](#) shadow document in the message body.
5. If the update request is not valid, AWS IoT publishes a message with the `$aws/things/thingName/shadow/name/shadowName/update/rejected` topic with an [Error response document \(p. 631\)](#) shadow document that describes the error.

To delete a property from a shadow by using the REST API

1. The device or client calls the [UpdateThingShadow \(p. 617\)](#) API with a [Request state document \(p. 628\)](#) that assigns `null` values to the properties of the shadow to delete. Include only the properties that you want to delete in the document. This is an example of a document that deletes the `engine` property.

```
{  
  "state": {  
    "desired": {  
      "engine": null  
    }  
  }  
}
```

2. If the request was valid, AWS IoT returns an HTTP success response code and an [/accepted response state document \(p. 629\)](#) shadow document as its response message body.

3. If the request was not valid, AWS IoT returns an HTTP error response code an [Error response document \(p. 631\)](#) as its response message body.

Deleting a shadow

Note

Setting the device's shadow state to `null` does not delete the shadow. The shadow version will be incremented on the next update.

Deleting a device's shadow does not delete the thing object. Deleting a thing object does not delete the corresponding device's shadow.

Deleting a shadow does not reset its version number to 0.

To delete a shadow by using the MQTT protocol

1. The device or client subscribes to these MQTT topics:
 - `$aws/things/thingName/shadow/name/shadowName/delete/accepted`
 - `$aws/things/thingName/shadow/name/shadowName/delete/rejected`
2. The device or client publishes a `$aws/things/thingName/shadow/name/shadowName/delete` with an empty message buffer.
3. If the delete request is valid, AWS IoT deletes the shadow and publishes a messages with the `$aws/things/thingName/shadow/name/shadowName/delete/accepted` topic and an abbreviated [/accepted response state document \(p. 629\)](#) shadow document in the message body. This is an example of the accepted delete message:

```
{  
    "version": 4,  
    "timestamp": 1591057529  
}
```

4. If the update request is not valid, AWS IoT publishes a message with the `$aws/things/thingName/shadow/name/shadowName/delete/rejected` topic with an [Error response document \(p. 631\)](#) shadow document that describes the error.

To delete a shadow by using the REST API

1. The device or client calls the [DeleteThingShadow \(p. 618\)](#) API with an empty message buffer.
2. If the request was valid, AWS IoT returns an HTTP success response code and an [/accepted response state document \(p. 629\)](#) and an abbreviated [/accepted response state document \(p. 629\)](#) shadow document in the message body. This is an example of the accepted delete message:

```
{  
    "version": 4,  
    "timestamp": 1591057529  
}
```

3. If the request was not valid, AWS IoT returns an HTTP error response code an [Error response document \(p. 631\)](#) as its response message body.

Device Shadow REST API

A shadow exposes the following URI for updating state information:

```
https://account-specific-prefix-ats.iot.region.amazonaws.com/things/thingName/shadow
```

The endpoint is specific to your AWS account. To find your endpoint, you can:

- Use the [describe-endpoint](#) command from the AWS CLI.
- Use the AWS IoT console settings. In **Settings**, the endpoint is listed under **Custom endpoint**.
- Use the AWS IoT console thing details page. In the console:
 1. Open **Manage** and under **Manage**, choose **Things**.
 2. In the list of things, choose the thing for which you want to get the endpoint URI.
 3. Choose the **Device Shadows** tab and choose your shadow. You can view the endpoint URI in the **Device Shadow URL** section of the **Device Shadow details** page.

The format of the endpoint is as follows:

```
identifier.iot.region.amazonaws.com
```

The shadow REST API follows the same HTTPS protocols/port mappings as described in [Device communication protocols \(p. 77\)](#).

API actions

- [GetThingShadow \(p. 616\)](#)
- [UpdateThingShadow \(p. 617\)](#)
- [DeleteThingShadow \(p. 618\)](#)
- [ListNamedShadowsForThing \(p. 618\)](#)

You can also use the API to create a named shadow by providing `name=shadowName` as part of the query parameter of the API.

GetThingShadow

Gets the shadow for the specified thing.

The response state document includes the delta between the desired and reported states.

Request

The request includes the standard HTTP headers plus the following URI:

```
HTTP GET https://endpoint/things/thingName/shadow?name=shadowName
Request body: (none)
```

The `name` query parameter is not required for unnamed (classic) shadows.

Response

Upon success, the response includes the standard HTTP headers plus the following code and body:

```
HTTP 200
Response Body: response state document
```

For more information, see [Example Response State Document \(p. 628\)](#).

Authorization

Retrieving a shadow requires a policy that allows the caller to perform the `iot:GetThingShadow` action. The Device Shadow service accepts two forms of authentication: Signature Version 4 with IAM credentials or TLS mutual authentication with a client certificate.

The following is an example policy that allows a caller to retrieve a device's shadow:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": "iot:GetThingShadow",  
            "Resource": [  
                "arn:aws:iot:region:account:thing/thing"  
            ]  
        }  
    ]  
}
```

UpdateThingShadow

Updates the shadow for the specified thing.

Updates affect only the fields specified in the request state document. Any field with a value of `null` is removed from the device's shadow.

Request

The request includes the standard HTTP headers plus the following URI and body:

```
HTTP POST https://endpoint/things/thingName/shadow?name=shadowName  
Request body: request state document
```

The `name` query parameter is not required for unnamed (classic) shadows.

For more information, see [Example Request State Document \(p. 628\)](#).

Response

Upon success, the response includes the standard HTTP headers plus the following code and body:

```
HTTP 200  
Response body: response state document
```

For more information, see [Example Response State Document \(p. 628\)](#).

Authorization

Updating a shadow requires a policy that allows the caller to perform the `iot:UpdateThingShadow` action. The Device Shadow service accepts two forms of authentication: Signature Version 4 with IAM credentials or TLS mutual authentication with a client certificate.

The following is an example policy that allows a caller to update a device's shadow:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": "iot:UpdateThingShadow",  
            "Resource": [  
                "arn:aws:iot:region:account:thing/thing"  
            ]  
        }  
    ]  
}
```

```
    "Action": "iot:UpdateThingShadow",
    "Resource": [
        "arn:aws:iot:region:account:thing/thing"
    ]
}
}
```

DeleteThingShadow

Deletes the shadow for the specified thing.

Request

The request includes the standard HTTP headers plus the following URI:

```
HTTP DELETE https://endpoint/things/thingName/shadow?name=shadowName
Request body: (none)
```

The name query parameter is not required for unnamed (classic) shadows.

Response

Upon success, the response includes the standard HTTP headers plus the following code and body:

```
HTTP 200
Response body: Empty response state document
```

Note that deleting a shadow does not reset its version number to 0.

Authorization

Deleting a device's shadow requires a policy that allows the caller to perform the iot:DeleteThingShadow action. The Device Shadow service accepts two forms of authentication: Signature Version 4 with IAM credentials or TLS mutual authentication with a client certificate.

The following is an example policy that allows a caller to delete a device's shadow:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iot:DeleteThingShadow",
            "Resource": [
                "arn:aws:iot:region:account:thing/thing"
            ]
        }
    ]
}
```

ListNamedShadowsForThing

Lists the shadows for the specified thing.

Request

The request includes the standard HTTP headers plus the following URI:

```
HTTP GET /api/things/shadow/ListNamedShadowsForThing/thingName?  
nextToken=nextToken&pageSize=pageSize  
Request body: (none)
```

nextToken

The token to retrieve the next set of results.

This value is returned on paged results and is used in the call that returns the next page.

pageSize

The number of shadow names to return in each call. See also `nextToken`.

thingName

The name of the thing for which to list the named shadows.

Response

Upon success, the response includes the standard HTTP headers plus the following response code and a [Shadow name list response document \(p. 632\)](#).

Note

The unnamed (classic) shadow does not appear in this list. The response is an empty list if you only have a classic shadow or if the `thingName` you specify doesn't exist.

```
HTTP 200  
Response body: Shadow name list document
```

Authorization

Listing a device's shadow requires a policy that allows the caller to perform the `iot:ListNamedShadowsForThing` action. The Device Shadow service accepts two forms of authentication: Signature Version 4 with IAM credentials or TLS mutual authentication with a client certificate.

The following is an example policy that allows a caller to list a thing's named shadows:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": "iot:ListNamedShadowsForThing",  
            "Resource": [  
                "arn:aws:iot:region:account:thing/thing"  
            ]  
        }  
    ]  
}
```

Device Shadow MQTT topics

The Device Shadow service uses reserved MQTT topics to enable devices and apps to get, update, or delete the state information for a device (shadow).

Publishing and subscribing on shadow topics requires topic-based authorization. AWS IoT reserves the right to add new topics to the existing topic structure. For this reason, we recommend that you avoid

wild card subscriptions to shadow topics. For example, avoid subscribing to topic filters like \$aws/things/thingName/shadow/# because the number of topics that match this topic filter might increase as AWS IoT introduces new shadow topics. For examples of the messages published on these topics see [Interacting with shadows \(p. 609\)](#).

Shadows can be named or unnamed (classic). The topics used by each differ only in the topic prefix. This table shows the topic prefix used by each shadow type.

ShadowTopicPrefix value	Shadow type
\$aws/things/ <i>thingName</i> /shadow	Unnamed (classic) shadow
\$aws/things/ <i>thingName</i> /shadow/name/ <i>shadowName</i>	Named shadow

To create a complete topic, select the *ShadowTopicPrefix* for the type of shadow to which you want to refer, replace *thingName*, and *shadowName* if applicable, with their corresponding values, and then append that with the topic stub as shown in the following sections.

The following are the MQTT topics used for interacting with shadows.

Topics

- [/get \(p. 620\)](#)
- [/get/accepted \(p. 621\)](#)
- [/get/rejected \(p. 621\)](#)
- [/update \(p. 622\)](#)
- [/update/delta \(p. 623\)](#)
- [/update/accepted \(p. 624\)](#)
- [/update/documents \(p. 624\)](#)
- [/update/rejected \(p. 625\)](#)
- [/delete \(p. 626\)](#)
- [/delete/accepted \(p. 626\)](#)
- [/delete/rejected \(p. 627\)](#)

/get

Publish an empty message to this topic to get the device's shadow:

```
ShadowTopicPrefix/get
```

AWS IoT responds by publishing to either [/get/accepted \(p. 621\)](#) or [/get/rejected \(p. 621\)](#).

Example policy

The following is an example of the required policy:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:Publish"
      ],
      "Resource": [
        "arn:aws:iot:region:account-id:topic/topic-name"
      ]
    }
  ]
}
```

```
        "iot:Publish"
    ],
    "Resource": [
        "arn:aws:iot:region:account:topic/$aws/things/thingName/shadow/get"
    ]
}
}
```

/get/accepted

AWS IoT publishes a response shadow document to this topic when returning the device's shadow:

```
ShadowTopicPrefix/get/accepted
```

For more information, see [Response state documents \(p. 628\)](#).

Example policy

The following is an example of the required policy:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
                "arn:aws:iot:region:account:topicfilter/$aws/things/thingName/shadow/get/accepted"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Receive"
            ],
            "Resource": [
                "arn:aws:iot:region:account:topic/$aws/things/thingName/shadow/get/accepted"
            ]
        }
    ]
}
```

/get/rejected

AWS IoT publishes an error response document to this topic when it can't return the device's shadow:

```
ShadowTopicPrefix/get/rejected
```

For more information, see [Error response document \(p. 631\)](#).

Example policy

The following is an example of the required policy:

```
{
```

```
"Version": "2012-10-17",
"Statement": [
  {
    "Effect": "Allow",
    "Action": [
      "iot:Subscribe"
    ],
    "Resource": [
      "arn:aws:iot:region:account:topicfilter/$aws/things/thingName/shadow/get/rejected"
    ]
  },
  {
    "Action": [
      "iot:Receive"
    ],
    "Resource": [
      "arn:aws:iot:region:account:topic/$aws/things/thingName/shadow/get/rejected"
    ]
  }
]
```

/update

Publish a request state document to this topic to update the device's shadow:

```
ShadowTopicPrefix/update
```

The message body contains a [partial request state document \(p. 628\)](#).

A client attempting to update the state of a device would send a JSON request state document with the `desired` property such as this:

```
{
  "state": {
    "desired": {
      "color": "red",
      "power": "on"
    }
  }
}
```

A device updating its shadow would send a JSON request state document with the `reported` property, such as this:

```
{
  "state": {
    "reported": {
      "color": "red",
      "power": "on"
    }
  }
}
```

AWS IoT responds by publishing to either [/update/accepted \(p. 624\)](#) or [/update/rejected \(p. 625\)](#).

Example policy

The following is an example of the required policy:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:Publish"  
            ],  
            "Resource": [  
                "arn:aws:iot:region:account:topic/$aws/things/thingName/shadow/update"  
            ]  
        }  
    ]  
}
```

/update/delta

AWS IoT publishes a response state document to this topic when it accepts a change for the device's shadow, and the request state document contains different values for desired and reported states:

```
ShadowTopicPrefix/update/delta
```

The message buffer contains a [/delta response state document \(p. 629\)](#).

Message body details

- A message published on update/delta includes only the desired attributes that differ between the desired and reported sections. It contains all of these attributes, regardless of whether these attributes were contained in the current update message or were already stored in AWS IoT. Attributes that do not differ between the desired and reported sections are not included.
- If an attribute is in the reported section but has no equivalent in the desired section, it is not included.
- If an attribute is in the desired section but has no equivalent in the reported section, it is included.
- If an attribute is deleted from the reported section but still exists in the desired section, it is included.

Example policy

The following is an example of the required policy:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:Subscribe"  
            ],  
            "Resource": [  
                "arn:aws:iot:region:account:topicfilter/$aws/things/thingName/shadow/update/delta"  
            ]  
        },  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:Receive"  
            ],  
            "Resource": [  
                "arn:aws:iot:region:account:topicfilter/$aws/things/thingName/shadow/update/delta"  
            ]  
        }  
    ]  
}
```

```
    "Resource": [
        "arn:aws:iot:region:account:topic/$aws/things/thingName/shadow/update/delta"
    ]
}
}
```

/update/accepted

AWS IoT publishes a response state document to this topic when it accepts a change for the device's shadow:

```
ShadowTopicPrefix/update/accepted
```

The message buffer contains a [/accepted response state document \(p. 629\)](#).

Example policy

The following is an example of the required policy:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
                "arn:aws:iot:region:account:topicfilter/$aws/things/thingName/shadow/update/
accepted"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Receive"
            ],
            "Resource": [
                "arn:aws:iot:region:account:topic/$aws/things/thingName/shadow/update/accepted"
            ]
        }
    ]
}
```

/update/documents

AWS IoT publishes a state document to this topic whenever an update to the shadow is successfully performed:

```
ShadowTopicPrefix/update/documents
```

The message body contains a [/documents response state document \(p. 629\)](#).

Example policy

The following is an example of the required policy:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:Subscribe"  
            ],  
            "Resource": [  
                "arn:aws:iot:region:account:topicfilter/$aws/things/thingName/shadow/update/documents"  
            ]  
        },  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:Receive"  
            ],  
            "Resource": [  
                "arn:aws:iot:region:account:topic/$aws/things/thingName/shadow/update/documents"  
            ]  
        }  
    ]  
}
```

/update/rejected

AWS IoT publishes an error response document to this topic when it rejects a change for the device's shadow:

```
ShadowTopicPrefix/update/rejected
```

The message body contains an [Error response document \(p. 631\)](#).

Example policy

The following is an example of the required policy:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:Subscribe"  
            ],  
            "Resource": [  
                "arn:aws:iot:region:account:topicfilter/$aws/things/thingName/shadow/update/rejected"  
            ]  
        },  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:Receive"  
            ],  
            "Resource": [  
                "arn:aws:iot:region:account:topic/$aws/things/thingName/shadow/update/rejected"  
            ]  
        }  
    ]  
}
```

```
    ]  
}
```

/delete

To delete a device's shadow, publish an empty message to the delete topic:

```
ShadowTopicPrefix/shadow/delete
```

The content of the message is ignored.

Note that deleting a shadow does not reset its version number to 0.

AWS IoT responds by publishing to either [/delete/accepted \(p. 626\)](#) or [/delete/rejected \(p. 627\)](#).

Example policy

The following is an example of the required policy:

```
{  
  "Version": "2012-10-17",  
  "Statement": [  
    {  
      "Effect": "Allow",  
      "Action": [  
        "iot:Publish"  
      ],  
      "Resource": [  
        "arn:aws:iot:region:account:topic/$aws/things/thingName/shadow/delete"  
      ]  
    }  
  ]  
}
```

/delete/accepted

AWS IoT publishes a message to this topic when a device's shadow is deleted:

```
ShadowTopicPrefix/delete/accepted
```

Example policy

The following is an example of the required policy:

```
{  
  "Version": "2012-10-17",  
  "Statement": [  
    {  
      "Effect": "Allow",  
      "Action": [  
        "iot:Subscribe"  
      ],  
      "Resource": [  
        "arn:aws:iot:region:account:topicfilter/$aws/things/thingName/shadow/delete/  
accepted"  
      ]  
    }  
  ]  
}
```

```
},
{
  "Effect": "Allow",
  "Action": [
    "iot:Receive"
  ],
  "Resource": [
    "arn:aws:iot:region:account:topic/$aws/things/thingName/shadow/delete/accepted"
  ]
}
]
```

/delete/rejected

AWS IoT publishes an error response document to this topic when it can't delete the device's shadow:

```
ShadowTopicPrefix/delete/rejected
```

The message body contains an [Error response document \(p. 631\)](#).

Example policy

The following is an example of the required policy:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:Subscribe"
      ],
      "Resource": [
        "arn:aws:iot:region:account:topicfilter/$aws/things/thingName/shadow/delete/rejected"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iot:Receive"
      ],
      "Resource": [
        "arn:aws:iot:region:account:topic/$aws/things/thingName/shadow/delete/rejected"
      ]
    }
  ]
}
```

Device Shadow service documents

The Device Shadow service respects all rules of the JSON specification. Values, objects, and arrays are stored in the device's shadow document.

Contents

- [Shadow document examples \(p. 628\)](#)

- Document properties (p. 632)
- Delta state (p. 633)
- Versioning shadow documents (p. 634)
- Client tokens in shadow documents (p. 634)
- Empty shadow document properties (p. 635)
- Array values in shadow documents (p. 635)

Shadow document examples

The Device Shadow service uses these documents in UPDATE, GET, and DELETE operations using the [REST API \(p. 615\)](#) or [MQTT Pub/Sub Messages \(p. 619\)](#).

Examples

- Request state document (p. 628)
- Response state documents (p. 628)
- Error response document (p. 631)
- Shadow name list response document (p. 632)

Request state document

A request state document has the following format:

```
{  
    "state": {  
        "desired": {  
            "attribute1": integer2,  
            "attribute2": "string2",  
            ...  
            "attributeN": boolean2  
        },  
        "reported": {  
            "attribute1": integer1,  
            "attribute2": "string1",  
            ...  
            "attributeN": boolean1  
        }  
    },  
    "clientToken": "token",  
    "version": version  
}
```

- **state** — Updates affect only the fields specified. Typically, you'll use either the `desired` or the `reported` property, but not both in the same request.
 - `desired` — The state properties and values requested to be updated in the device.
 - `reported` — The state properties and values reported by the device.
- `clientToken` — If used, you can match the request and corresponding response by the client token.
- `version` — If used, the Device Shadow service processes the update only if the specified version matches the latest version it has.

Response state documents

Response state documents have the following format depending on the response type.

/accepted response state document

```
{  
    "state": {  
        "desired": {  
            "attribute1": integer2,  
            "attribute2": "string2",  
            ...  
            "attributeN": boolean2  
        }  
    },  
    "metadata": {  
        "desired": {  
            "attribute1": {  
                "timestamp": timestamp  
            },  
            "attribute2": {  
                "timestamp": timestamp  
            },  
            ...  
            "attributeN": {  
                "timestamp": timestamp  
            }  
        }  
    },  
    "timestamp": timestamp,  
    "clientToken": "token",  
    "version": version  
}
```

/delta response state document

```
{  
    "state": {  
        "attribute1": integer2,  
        "attribute2": "string2",  
        ...  
        "attributeN": boolean2  
    }  
},  
    "metadata": {  
        "attribute1": {  
            "timestamp": timestamp  
        },  
        "attribute2": {  
            "timestamp": timestamp  
        },  
        ...  
        "attributeN": {  
            "timestamp": timestamp  
        }  
    },  
    "timestamp": timestamp,  
    "clientToken": "token",  
    "version": version  
}
```

/documents response state document

```
{  
    "previous" : {  
        "state": {
```

```

    "desired": {
        "attribute1": integer2,
        "attribute2": "string2",
        ...
        "attributeN": boolean2
    },
    "reported": {
        "attribute1": integer1,
        "attribute2": "string1",
        ...
        "attributeN": boolean1
    }
},
"metadata": {
    "desired": {
        "attribute1": {
            "timestamp": timestamp
        },
        "attribute2": {
            "timestamp": timestamp
        },
        ...
        "attributeN": {
            "timestamp": timestamp
        }
    },
    "reported": {
        "attribute1": {
            "timestamp": timestamp
        },
        "attribute2": {
            "timestamp": timestamp
        },
        ...
        "attributeN": {
            "timestamp": timestamp
        }
    }
},
"version": version-1
},
"current": {
    "state": {
        "desired": {
            "attribute1": integer2,
            "attribute2": "string2",
            ...
            "attributeN": boolean2
        },
        "reported": {
            "attribute1": integer2,
            "attribute2": "string2",
            ...
            "attributeN": boolean2
        }
    },
    "metadata": {
        "desired": {
            "attribute1": {
                "timestamp": timestamp
            },
            "attribute2": {
                "timestamp": timestamp
            },
            ...
            "attributeN": {
                "timestamp": timestamp
            }
        }
    }
}

```

```

        "timestamp": timestamp
    }
},
"reported": {
    "attribute1": {
        "timestamp": timestamp
    },
    "attribute2": {
        "timestamp": timestamp
    },
    ...
    "attributeN": {
        "timestamp": timestamp
    }
}
},
"version": version
},
"timestamp": timestamp,
"clientToken": "token"
}

```

Response state document properties

- **previous** — After a successful update, contains the state of the object before the update.
- **current** — After a successful update, contains the state of the object after the update.
- **state**
 - **reported** — Present only if a thing reported any data in the `reported` section and contains only fields that were in the request state document.
 - **desired** — Present only if a device reported any data in the `desired` section and contains only fields that were in the request state document.
 - **delta** — Present only if the desired data differs from the shadow's current `reported` data.
- **metadata** — Contains the timestamps for each attribute in the `desired` and `reported` sections so that you can determine when the state was updated.
- **timestamp** — The Epoch date and time the response was generated by AWS IoT.
- **clientToken** — Present only if a client token was used when publishing valid JSON to the `/update` topic.
- **version** — The current version of the document for the device's shadow shared in AWS IoT. It is increased by one over the previous version of the document.

Error response document

An error response document has the following format:

```
{
    "code": error-code,
    "message": "error-message",
    "timestamp": timestamp,
    "clientToken": "token"
}
```

- **code** — An HTTP response code that indicates the type of error.
- **message** — A text message that provides additional information.
- **timestamp** — The date and time the response was generated by AWS IoT. This property is not present in all error response documents.

- `clientToken` — Present only if a client token was used in the published message.

For more information, see [Device Shadow error messages \(p. 636\)](#).

Shadow name list response document

A shadow name list response document has the following format:

```
{  
    "results": [  
        "shadowName-1",  
        "shadowName-2",  
        "shadowName-3",  
        "shadowName-n"  
    ],  
    "nextToken": "nextToken",  
    "timestamp": timestamp  
}
```

- `results` — The array of shadow names.
- `nextToken` — The token value to use in paged requests to get the next page in the sequence. This property is not present when there are no more shadow names to return.
- `timestamp` — The date and time the response was generated by AWS IoT.

Document properties

A device's shadow document has the following properties:

`state`

`desired`

The desired state of the device. Apps can write to this portion of the document to update the state of a device directly without having to connect to it.

`reported`

The reported state of the device. Devices write to this portion of the document to report their new state. Apps read this portion of the document to determine the device's last-reported state.

`metadata`

Information about the data stored in the `state` section of the document. This includes timestamps, in Epoch time, for each attribute in the `state` section, which enables you to determine when they were updated.

Note

Metadata do not contribute to the document size for service limits or pricing. For more information, see [AWS IoT Service Limits](#).

`timestamp`

Indicates when the message was sent by AWS IoT. By using the timestamp in the message and the timestamps for individual attributes in the `desired` or `reported` section, a device can determine a property's age, even if the device doesn't have an internal clock.

`clientToken`

A string unique to the device that enables you to associate responses with requests in an MQTT environment.

version

The document version. Every time the document is updated, this version number is incremented. Used to ensure the version of the document being updated is the most recent.

For more information, see [Shadow document examples \(p. 628\)](#).

Delta state

Delta state is a virtual type of state that contains the difference between the desired and reported states. Fields in the desired section that are not in the reported section are included in the delta. Fields that are in the reported section and not in the desired section are not included in the delta. The delta contains metadata, and its values are equal to the metadata in the desired field. For example:

```
{  
  "state": {  
    "desired": {  
      "color": "RED",  
      "state": "STOP"  
    },  
    "reported": {  
      "color": "GREEN",  
      "engine": "ON"  
    },  
    "delta": {  
      "color": "RED",  
      "state": "STOP"  
    }  
  },  
  "metadata": {  
    "desired": {  
      "color": {  
        "timestamp": 12345  
      },  
      "state": {  
        "timestamp": 12345  
      }  
    },  
    "reported": {  
      "color": {  
        "timestamp": 12345  
      },  
      "engine": {  
        "timestamp": 12345  
      }  
    },  
    "delta": {  
      "color": {  
        "timestamp": 12345  
      },  
      "state": {  
        "timestamp": 12345  
      }  
    }  
  },  
  "version": 17,  
  "timestamp": 123456789  
}
```

When nested objects differ, the delta contains the path all the way to the root.

```
{
  "state": {
    "desired": {
      "lights": {
        "color": {
          "r": 255,
          "g": 255,
          "b": 255
        }
      }
    },
    "reported": {
      "lights": {
        "color": {
          "r": 255,
          "g": 0,
          "b": 255
        }
      }
    },
    "delta": {
      "lights": {
        "color": {
          "g": 255
        }
      }
    }
  },
  "version": 18,
  "timestamp": 123456789
}
```

The Device Shadow service calculates the delta by iterating through each field in the desired state and comparing it to the reported state.

Arrays are treated like values. If an array in the desired section doesn't match the array in the reported section, then the entire desired array is copied into the delta.

Versioning shadow documents

The Device Shadow service supports versioning on every update message, both request and response. This means that with every update of a shadow, the version of the JSON document is incremented. This ensures two things:

- A client can receive an error if it attempts to overwrite a shadow using an older version number. The client is informed it must resync before it can update a device's shadow.
- A client can decide not to act on a received message if the message has a lower version than the version stored by the client.

A client can bypass version matching by not including a version in the shadow document.

Client tokens in shadow documents

You can use a client token with MQTT-based messaging to verify the same client token is contained in a request and request response. This ensures the response and request are associated.

Note

The client token can be no longer than 64 bytes. A client token that is longer than 64 bytes causes a 400 (Bad Request) response and an *Invalid clientToken* error message.

Empty shadow document properties

The `reported` and `desired` properties in a shadow document can be empty or omitted when they don't apply to the current shadow state. For example, a shadow document contains a `desired` property only if it has a desired state. The following is a valid example of a state document with no `desired` property:

```
{  
    "reported" : { "temp": 55 }  
}
```

The `reported` property can also be empty, such as if the shadow has not been updated by the device:

```
{  
    "desired" : { "color" : "RED" }  
}
```

If an update causes the `desired` or `reported` properties to become null, it is removed from the document. The following shows how to remove the `desired` property by setting it to null. You might do this when a device updates its state, for example.

```
{  
    "state": {  
        "reported": {  
            "color": "red"  
        },  
        "desired": null  
    }  
}
```

A shadow document can also have neither `desired` or `reported` properties, making the shadow document empty. This is an example of an empty, yet valid shadow document.

```
{  
}
```

Array values in shadow documents

Shadows support arrays, but treat them as normal values in that an update to an array replaces the whole array. It is not possible to update part of an array.

Initial state:

```
{  
    "desired" : { "colors" : [ "RED", "GREEN", "BLUE" ] }  
}
```

Update:

```
{  
    "desired" : { "colors" : [ "RED" ] }  
}
```

Final state:

```
{
    "desired" : { "colors" : [ "RED" ] }
}
```

Arrays can't have null values. For example, the following array is not valid and will be rejected.

```
{
    "desired" : {
        "colors" : [ null, "RED", "GREEN" ]
    }
}
```

Device Shadow error messages

The Device Shadow service publishes a message on the error topic (over MQTT) when an attempt to change the state document fails. This message is only emitted as a response to a publish request on one of the reserved \$aws topics. If the client updates the document using the REST API, then it receives the HTTP error code as part of its response, and no MQTT error messages are emitted.

HTTP error code	Error messages
400 (Bad Request)	<ul style="list-style-type: none"> • Invalid JSON • Missing required node: state • State node must be an object • Desired node must be an object • Reported node must be an object • Invalid version • Invalid clientToken <p>Note A client token that is longer than 64 bytes will cause this response.</p> <ul style="list-style-type: none"> • JSON contains too many levels of nesting; maximum is 6 • State contains an invalid node
401 (Unauthorized)	<ul style="list-style-type: none"> • Unauthorized
403 (Forbidden)	<ul style="list-style-type: none"> • Forbidden
404 (Not Found)	<ul style="list-style-type: none"> • Thing not found • No shadow exists with name: <i>shadowName</i>
409 (Conflict)	<ul style="list-style-type: none"> • Version conflict
413 (Payload Too Large)	<ul style="list-style-type: none"> • The payload exceeds the maximum size allowed
415 (Unsupported Media Type)	<ul style="list-style-type: none"> • Unsupported documented encoding; supported encoding is UTF-8
429 (Too Many Requests)	<ul style="list-style-type: none"> • The Device Shadow service will generate this error message when there are more than 10 in-flight requests on a single connection.
500 (Internal Server Error)	<ul style="list-style-type: none"> • Internal service failure

Jobs

Use AWS IoT Jobs to define a set of remote operations that can be sent to and run on one or more devices connected to AWS IoT. For example, you can define a job that instructs a set of devices to download and install applications, run firmware updates, reboot, rotate certificates, or perform remote troubleshooting operations.

To create jobs, first define a *job document* that contains a list of instructions describing operations that the device must perform remotely. To perform these operations, specify a list of *targets*, which are individual things, [thing groups \(p. 258\)](#), or both. The job document and targets together constitute a *deployment*.

Each deployment can have additional configurations:

- **Rollout:** This configuration defines how many devices receive the job document every minute.
- **Abort:** If a certain number of devices don't receive the job document, use this configuration to cancel the job and avoid sending a bad update to an entire fleet.
- **Timeout:** If a response isn't received from your job targets within a certain duration, the job can fail. You can keep track of the job that's running on these devices.

AWS IoT Jobs sends a message to inform the targets that a job is available. The target starts the *execution* of the job by downloading the job document, performing the operations it specifies, and reporting its progress to AWS IoT. You can track the progress of a job for a specific target and for all targets of the job by running commands that are provided by AWS IoT Jobs. When a job has started, an *In progress* status is reported. The devices then report incremental updates while displaying this status until the job has succeeded, failed, or timed out.

Jobs key concepts

Job

A job is a remote operation that is sent to and run on one or more devices connected to AWS IoT. For example, you can define a job that instructs a set of devices to download and install an application or run firmware updates, reboot, rotate certificates, or perform remote troubleshooting operations.

Job document

To create a job, you must first create a job document that is a description of the remote operations to be performed by the devices.

Job documents are UTF-8 encoded JSON documents and should contain information that your devices need to perform a job. A job document contains one or more URLs where the device can download an update or some other data. The job document can be stored in an Amazon S3 bucket, or be included inline with the command that creates the job.

Tip

For job document examples, see the [jobs-agent.js](#) example in the AWS IoT SDK for JavaScript.

Target

When you create a job, you specify a list of targets that are the devices that should perform the operations. The targets can be things or [thing groups \(p. 258\)](#) or both. The AWS IoT Jobs service sends a message to each target to inform it that a job is available.

Job execution

A job execution is an instance of a job on a target device. The target starts an execution of a job by downloading the job document. It then performs the operations specified in the document, and reports its progress to AWS IoT. An execution number is a unique identifier of a job execution on a specific target. The Jobs service provides commands to track the progress of a job execution on a target and the progress of a job across all targets.

Snapshot job

By default, a job is sent to all targets that you specify when you create the job. After those targets complete the job (or report that they're unable to do so), the job is complete.

Continuous job

A continuous job is sent to all targets that you specify when you create the job. It continues to run and is sent to any new devices (things) that are added to the target group. For example, a continuous job can be used to onboard or upgrade devices as they're added to a group. You can make a job continuous by setting an optional parameter when you create the job.

Rollouts

You can specify how quickly targets are notified of a pending job deployment. This allows you to create a staged rollout to better manage updates, reboots, and other operations.

You can create a rollout configuration by using either a static rollout rate or an exponential rollout rate. You can set a static rollout rate to specify the maximum number of job targets to inform per minute.

For examples of setting rollout rates and for more information about configuring job rollouts, see [Job rollout and abort configuration](#).

Abort

You can create a set of conditions to cancel rollouts when criteria that you specify have been met. For more information, see [Job rollout and abort configuration](#).

Presigned URLs

For secure, time-limited access to data that's not included in the job document, you can use presigned Amazon S3 URLs. Place your data in an Amazon S3 bucket and add a placeholder link to the data in the job document. When AWS IoT Jobs receives a request for the job document, it parses the job document by looking for the placeholder links, and then replaces the links with presigned Amazon S3 URLs.

The placeholder link is of the following form:

```
${aws:iot:s3-presigned-url:https://s3.amazonaws.com/bucket/key}
```

where **bucket** is your bucket name and **key** is the object in the bucket to which you are linking.

In the Beijing and Ningxia Regions, presigned URLs work only if the resource owner has an ICP (Internet Content Provider) license. For more information, see [Amazon Simple Storage Service](#) in the [Getting Started with AWS Services in China](#) documentation.

Timeouts

Job timeouts make it possible to be notified whenever a job deployment gets stuck in the IN_PROGRESS state for an unexpectedly long period of time. There are two types of timers: in-progress timers and step timers. When the job is IN_PROGRESS, you can monitor and track the progress of your job deployment.

Rollouts and abort configurations are specific to your job whereas the timeout configuration is specific to a job deployment. For more information, see [Job executions timeout configuration \(p. 678\)](#).

Managing jobs

Use jobs to notify devices of a software or firmware update. You can use the [AWS IoT console](#), the [Job management and control API \(p. 679\)](#), the [AWS Command Line Interface](#), or the [AWS SDKs](#) to create and manage jobs.

Code signing for jobs

When sending code to devices, for devices to detect whether the code has been modified in transit, we recommend that you sign the code file by using the AWS CLI. For instructions, see [???](#) (p. 641).

For more information, see [What Is Code Signing for AWS IoT?](#).

Job document

Before you create a job, you must create a job document. If you're using code signing for AWS IoT, you must upload your job document to a versioned Amazon S3 bucket. For more information about creating an Amazon S3 bucket and uploading files to it, see [Getting Started with Amazon Simple Storage Service](#) in the [Amazon S3 Getting Started Guide](#).

Tip

For job document examples, see the [jobs-agent.js](#) example in the AWS IoT SDK for JavaScript.

Presigned URLs

Your job document can contain a presigned Amazon S3 URL that points to your code file (or other file). Presigned Amazon S3 URLs are valid only for a limited amount of time and are generated when a device requests a job document. Because the presigned URL isn't created when you're creating the job document, use a placeholder URL in your job document instead. A placeholder URL looks like the following:

```
 ${aws:iot:s3-presigned-url:https://s3.region.amazonaws.com/<bucket>/<code file>}
```

where:

- **bucket** is the Amazon S3 bucket that contains the code file.
- **code file** is the Amazon S3 key of the code file.

When a device requests the job document, AWS IoT generates the presigned URL and replaces the placeholder URL with the presigned URL. Your job document is then sent to the device.

IAM role to grant permission to download files from S3

When you create a job that uses presigned Amazon S3 URLs, you must provide an IAM role that grants permission to download files from the Amazon S3 bucket where the data or updates are stored. The role must also grant permission for AWS IoT to assume the role.

You can specify an optional timeout for the presigned URL. For more information, see [CreateJob](#).

Grant AWS IoT Jobs permission to assume your role

1. Go to the [Roles hub of the IAM console](#) and choose your role.
2. On the **Trust Relationships** tab, choose **Edit Trust Relationship** and replace the policy document with the following JSON. Choose **Update Trust Policy**.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "",  
            "Effect": "Allow",  
            "Principal": {  
                "Service": [  
                    "iot.amazonaws.com"  
                ]  
            },  
            "Action": "sts:AssumeRole"  
        }  
    ]  
}
```

3. If your job uses a job document that's an Amazon S3 object, choose **Permissions** and with the following JSON, add a policy that grants permission to download files from your Amazon S3 bucket:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": "s3:GetObject",  
            "Resource": "arn:aws:s3:::your_S3_bucket/*"  
        }  
    ]  
}
```

Topics

- [Create and manage jobs by using the AWS Management Console \(p. 640\)](#)
- [Create and manage jobs by using the AWS CLI \(p. 641\)](#)

Create and manage jobs by using the AWS Management Console

To create a job

1. Choose your job type

1. Go to the [Job hub of the AWS IoT console](#) and choose **Create job**.
2. Depending on the device that you're using, you can create a custom job, a FreeRTOS OTA update job, or an AWS IoT Greengrass job. For this example, choose **Create a custom job**.

2. Enter job properties

Enter a unique, alphanumeric job name, optional description, and tags, and then choose **Next**.

Note

We recommend that you don't use personally identifiable information in your job IDs and description.

3. Choose your targets

Choose as job targets the things or thing groups you want to run in this job.

4. Specify your job document

You can either upload your JSON job file to an S3 bucket and then use that file as the job document, or choose your job file from a template.

If you're using a template, you can choose from a custom job template or an AWS managed template. If you're creating a job for performing frequently used remote actions such as rebooting your device, you can use an AWS managed template. These templates have already been preconfigured for use. For more information, see [Create a custom job template \(p. 660\)](#) and [Create custom job templates from managed templates \(p. 658\)](#).

5. Choose your job type

On the **Job configuration** page, choose the job type as continuous or a snapshot job. A snapshot job is complete when it finishes its run on the target devices and groups. A continuous job applies to thing groups and runs on any device that you subsequently add to a specified target group.

6. Specify additional configurations (optional)

Continue to add any additional configurations for your job and then review and create your job. For information about the additional configurations, see:

- [Job rollout and abort configuration \(p. 676\)](#)
- [Job executions timeout configuration \(p. 678\)](#)

After you create the job, the console generates a JSON signature and places it in your job document. You can use the [AWS IoT console](#) to view the status, cancel, or delete a job. To manage jobs, go to the [Job hub of the console](#).

Create and manage jobs by using the AWS CLI

This section describes how to create and manage jobs.

Create jobs

To create an AWS IoT job, use the **CreateJob** command. The job is queued for execution on the targets (things or thing groups) that you specify. To create an AWS IoT job, you need a job document that can be included in the body of the request or as a link to an Amazon S3 document. If the job includes downloading files using presigned Amazon S3 URLs, you need an IAM role Amazon Resource Name (ARN) that has permission to download the file and grants permission to the AWS IoT Jobs service to assume the role.

Code signing with jobs

If you're using code signing for AWS IoT, you must start a code signing job and include the output in your job document. Use the **start-signing-job** command to create a code signing job. **start-signing-job** returns a job ID. To get the Amazon S3 location where the signature is stored, use the **describe-signing-job** command. You can then download the signature from Amazon S3. For more information about code signing jobs, see [Code signing for AWS IoT](#).

Your job document must contain a presigned URL placeholder for your code file and the JSON signature output placed in an Amazon S3 bucket using the **start-signing-job** command, enclosed in a **codesign** element:

```
{  
    "presign": "${aws:iot:s3-presigned-url:https://s3.region.amazonaws.com/bucket/image}",  
    "codesign": {  
        "rawPayloadSize": <image-file-size>,  
        "signature": <signature>,
```

```

    "signatureAlgorithm": <signature-algorithm>,
    "payloadLocation": {
        "s3": {
            "bucketName": <my-s3-bucket>,
            "key": <my-code-file>,
            "version": <code-file-version-id>
        }
    }
}

```

Create a job with a job document

The following command shows how to create a job using a job document (`job-document.json`) stored in an Amazon S3 bucket (`jobBucket`), and a role with permission to download files from Amazon S3 (`S3DownloadRole`).

```

aws iot create-job \
--job-id 010 \
--targets arn:aws:iot:us-east-1:123456789012:thing/thingOne \
--document-source https://s3.amazonaws.com/my-s3-bucket/job-document.json \
--timeout-config inProgressTimeoutInMinutes=100 \
--job-executions-rollout-config "{\"exponentialRate\": { \"baseRatePerMinute\": 50, \
\"incrementFactor\": 2, \"rateIncreaseCriteria\": { \"numberOfNotifiedThings\": 1000, \
\"numberOfSucceededThings\": 1000 }}, \"maximumPerMinute\": 1000} \
--abort-config \"{ \"criteriaList\": [ { \"action\": \"CANCEL\", \"failureType\": \
\"FAILED\", \"minNumberOfExecutedThings\": 100, \"thresholdPercentage\": 20 }, { \"action\": \
\"CANCEL\", \"failureType\": \"TIMED_OUT\", \"minNumberOfExecutedThings\": 200, \
\"thresholdPercentage\": 50 } ] }\" \
--presigned-url-config "{\"roleArn\": \"arn:aws:iam::123456789012:role/ \
S3DownloadRole\", \"expiresInSec\": 3600}"

```

The job is run on `thingOne`.

The optional `timeout-config` parameter specifies the amount of time each device has to finish its execution of the job. The timer starts when the job execution status is set to `IN_PROGRESS`. If the job execution status isn't set to another terminal state before the time expires, it's set to `TIMED_OUT`.

The in-progress timer can't be updated and applies to all job executions for the job. Whenever a job execution remains in the `IN_PROGRESS` state for longer than this interval, it fails and switches to the terminal `TIMED_OUT` status. AWS IoT also publishes an MQTT notification.

For more information about creating configurations for job rollouts and aborts, see [Job Rollout and Abort Configuration](#).

Note

Job documents that are specified as Amazon S3 files are retrieved at the time you create the job. If you change the contents of the Amazon S3 file that you used as the source of your job document after you've created the job document, then what is sent to the job targets doesn't change.

Update a job

To update a job, use the `UpdateJob` command. You can update the `description`, `presignedUrlConfig`, `jobExecutionsRolloutConfig`, `abortConfig`, and `timeoutConfig` fields of a job.

```

aws iot update-job \
--job-id 010 \
--description "updated description" \
--timeout-config inProgressTimeoutInMinutes=100 \

```

```
--job-executions-rollout-config "{ \"exponentialRate\": { \"baseRatePerMinute\": 50, \"incrementFactor\": 2, \"rateIncreaseCriteria\": { \"numberOfNotifiedThings\": 1000, \"numberOfSucceededThings\": 1000 }, \"maximumPerMinute\": 1000 } }" \
--abort-config "{ \"criteriaList\": [ { \"action\": \"CANCEL\", \"failureType\": \"FAILED\", \"minNumberOfExecutedThings\": 100, \"thresholdPercentage\": 20 }, { \"action\": \"CANCEL\", \"failureType\": \"TIMED_OUT\", \"minNumberOfExecutedThings\": 200, \"thresholdPercentage\": 50 } ] }" \
--presigned-url-config "{\"roleArn\":\"arn:aws:iam::123456789012:role/S3DownloadRole\", \"expiresInSec\":3600}"
```

For more information, see [Job Rollout and Abort Configuration](#).

Cancel a job

To cancel a job, you use the **CancelJob** command. Canceling a job stops AWS IoT from rolling out any new job executions for the job. It also cancels any job executions that are in a `QUEUED` state. AWS IoT keeps any job executions in a terminal state untouched because the device has already completed the job. If the status of a job execution is `IN_PROGRESS`, it also remains untouched unless you use the optional `--force` parameter.

The following command shows how to cancel a job with ID 010.

```
aws iot cancel-job --job-id 010
```

The command displays the following output:

```
{
  "jobArn": "string",
  "jobId": "string",
  "description": "string"
}
```

When you cancel a job, job executions that are in a `QUEUED` state are canceled. Job executions that are in an `IN_PROGRESS` state are canceled, but only if you specify the optional `--force` parameter. Job executions in a terminal state aren't canceled.

Warning

Canceling a job that's in the `IN_PROGRESS` state (by setting the `--force` parameter) cancels any job executions that are in progress and causes the device that's running the job to be unable to update the job execution status. Use caution and make sure that each device executing a canceled job can recover to a valid state.

The status of a canceled job or of one of its job executions is eventually consistent. AWS IoT stops scheduling new job executions and `QUEUED` job executions for that job to devices as soon as possible. Changing the status of a job execution to `CANCELED` might take some time, depending on the number of devices and other factors.

If a job is canceled because it has met the criteria defined by an `AbortConfig` object, the service adds auto-populated values for the `comment` and `reasonCode` fields. You can create your own values for `reasonCode` when the job cancellation is user-driven.

Cancel a job execution

To cancel a job execution on a device, you use the **CancelJobExecution** command. It cancels a job execution that's in a `QUEUED` state. If you want to cancel a job execution that's in progress, you must use the `--force` parameter.

The following command shows how to cancel the job execution from job 010 running on `myThing`.

```
aws iot cancel-job-execution --job-id 010 --thing-name myThing
```

The command displays no output.

A job execution that's in a `QUEUED` state is canceled. A job execution that's in an `IN_PROGRESS` state is canceled, but only if you specify the optional `--force` parameter. Job executions in a terminal state can't be canceled.

Warning

When you cancel a job execution that's in the `IN_PROGRESS` state, the device can't update the job execution status. Use caution and make sure that the device can recover to a valid state.

If the job execution is in a terminal state or if the job execution is in an `IN_PROGRESS` state and the `--force` parameter isn't set to `true`, this command causes an `InvalidStateTransitionException`.

The status of a canceled job execution is eventually consistent. Changing the status of a job execution to `CANCELED` might take some time, depending on various factors.

Delete a job

To delete a job and its job executions, use the **DeleteJob** command. By default, you can only delete a job that's in a terminal state (`SUCCEEDED` or `CANCELED`). Otherwise, an exception occurs. You can delete a job in the `IN_PROGRESS` state, however, if the `force` parameter is set to `true`.

To delete a job, run the following command:

```
aws iot delete-job --job-id 010 --force|--no-force
```

The command displays no output.

Warning

When you delete a job that's in the `IN_PROGRESS` state, the device that's deploying the job can't access job information or update the job execution status. Use caution and make sure that each device deploying a job that has been deleted can recover to a valid state.

It can take some time to delete a job, depending on the number of job executions created for the job and other factors. While the job is being deleted, `DELETION_IN_PROGRESS` appears as the status of the job. An error results if you attempt to delete or cancel a job with a status that's already `DELETION_IN_PROGRESS`.

Only 10 jobs can have a status of `DELETION_IN_PROGRESS` at the same time. Otherwise, a `LimitExceededException` occurs.

Get a job document

To retrieve a job document for a job, use the **GetJobDocument** command. A job document is a description of the remote operations to be performed by the devices.

To get a job document, run the following command:

```
aws iot get-job-document --job-id 010
```

The command returns the job document for the specified job:

```
{
    "document": "{\n\t\"operation\":\"install\", \n\t"url\":\"http://amazon.com/\nfirmWareUpdate-01\", \n\t"data\":\"\${aws:iot:s3-presigned-url:https://s3.amazonaws.com/job-\ntest-bucket/datafile}\n\""
}
```

Note

When you use this command to retrieve a job document, placeholder URLs aren't replaced by presigned Amazon S3 URLs. When a device calls the [GetPendingJobExecutions](#) API operation, the placeholder URLs are replaced by presigned Amazon S3 URLs in the job document.

List jobs

To get a list of all jobs in your AWS account, use the **ListJobs** command. Job data and job execution data are retained for a [limited time](#). Run the following command to list all jobs in your AWS account:

```
aws iot list-jobs
```

The command returns all jobs in your account, sorted by the job status:

```
{  
    "jobs": [  
        {  
            "status": "IN_PROGRESS",  
            "lastUpdatedAt": 1486687079.743,  
            "jobArn": "arn:aws:iot:us-east-1:123456789012:job/013",  
            "createdAt": 1486687079.743,  
            "targetSelection": "SNAPSHOT",  
            "jobId": "013"  
        },  
        {  
            "status": "SUCCEEDED",  
            "lastUpdatedAt": 1486685868.444,  
            "jobArn": "arn:aws:iot:us-east-1:123456789012:job/012",  
            "createdAt": 1486685868.444,  
            "completedAt": 148668789.690,  
            "targetSelection": "SNAPSHOT",  
            "jobId": "012"  
        },  
        {  
            "status": "CANCELED",  
            "lastUpdatedAt": 1486678850.575,  
            "jobArn": "arn:aws:iot:us-east-1:123456789012:job/011",  
            "createdAt": 1486678850.575,  
            "targetSelection": "SNAPSHOT",  
            "jobId": "011"  
        }  
    ]  
}
```

Describe a job

To get the status of a job, run the **DescribeJob** command. The following command shows how to describe a job:

```
$ aws iot describe-job --job-id 010
```

The command returns the status of the specified job. For example:

```
{  
    "documentSource": "https://s3.amazonaws.com/job-test-bucket/job-document.json",  
    "job": {  
        "status": "IN_PROGRESS",  
        "jobArn": "arn:aws:iot:us-east-1:123456789012:job/010",  
        "lastUpdatedAt": 1486687079.743,  
        "createdAt": 1486687079.743,  
        "targetSelection": "SNAPSHOT",  
        "jobId": "010"  
    }  
}
```

```

    "targets": [
        "arn:aws:iot:us-east-1:123456789012:thing/myThing"
    ],
    "jobProcessDetails": {
        "numberOfCanceledThings": 0,
        "numberOfFailedThings": 0,
        "numberOfInProgressThings": 0,
        "numberOfQueuedThings": 0,
        "numberOfRejectedThings": 0,
        "numberOfRemovedThings": 0,
        "numberOfSucceededThings": 0,
        "numberOfTimedOutThings": 0,
        "processingTargets": [
            "arn:aws:iot:us-east-1:123456789012:thing/thingOne",
            "arn:aws:iot:us-east-1:123456789012:thinggroup/thinggroupOne",
            "arn:aws:iot:us-east-1:123456789012:thing/thingTwo",
            "arn:aws:iot:us-east-1:123456789012:thinggroup/thinggroupTwo"
        ]
    },
    "presignedUrlConfig": {
        "expiresInSec": 60,
        "roleArn": "arn:aws:iam::123456789012:role/S3DownloadRole"
    },
    "jobId": "010",
    "lastUpdatedAt": 1486593195.006,
    "createdAt": 1486593195.006,
    "targetSelection": "SNAPSHOT",
    "jobExecutionsRolloutConfig": {
        "exponentialRate": {
            "baseRatePerMinute": integer,
            "incrementFactor": integer,
            "rateIncreaseCriteria": {
                "numberOfNotifiedThings": integer, // Set one or the other
                "numberOfSucceededThings": integer // of these two values.
            },
            "maximumPerMinute": integer
        }
    },
    "abortConfig": {
        "criteriaList": [
            {
                "action": "string",
                "failureType": "string",
                "minNumberOfExecutedThings": integer,
                "thresholdPercentage": integer
            }
        ]
    },
    "timeoutConfig": {
        "inProgressTimeoutInMinutes": number
    }
}
}

```

List executions for a job

A job running on a specific device is represented by a job execution object. Run the **ListJobExecutionsForJob** command to list all job executions for a job. The following shows how to list the executions for a job:

```
aws iot list-job-executions-for-job --job-id 010
```

The command returns a list of job executions:

```
{  
    "executionSummaries": [  
        {  
            "thingArn": "arn:aws:iot:us-east-1:123456789012:thing/thingOne",  
            "jobExecutionSummary": {  
                "status": "QUEUED",  
                "lastUpdatedAt": 1486593196.378,  
                "queuedAt": 1486593196.378,  
                "executionNumber": 1234567890  
            }  
        },  
        {  
            "thingArn": "arn:aws:iot:us-east-1:123456789012:thing/thingTwo",  
            "jobExecutionSummary": {  
                "status": "IN_PROGRESS",  
                "lastUpdatedAt": 1486593345.659,  
                "queuedAt": 1486593196.378,  
                "startedAt": 1486593345.659,  
                "executionNumber": 4567890123  
            }  
        }  
    ]  
}
```

List job executions for a thing

Run the **ListJobExecutionsForThing** command to list all job executions running on a thing. The following shows how to list job executions for a thing:

```
aws iot list-job-executions-for-thing --thing-name thingOne
```

The command returns a list of job executions that are running or have run on the specified thing:

```
{  
    "executionSummaries": [  
        {  
            "jobExecutionSummary": {  
                "status": "QUEUED",  
                "lastUpdatedAt": 1486687082.071,  
                "queuedAt": 1486687082.071,  
                "executionNumber": 9876543210  
            },  
            "jobId": "013"  
        },  
        {  
            "jobExecutionSummary": {  
                "status": "IN_PROGRESS",  
                "startAt": 1486685870.729,  
                "lastUpdatedAt": 1486685870.729,  
                "queuedAt": 1486685870.729,  
                "executionNumber": 1357924680  
            },  
            "jobId": "012"  
        },  
        {  
            "jobExecutionSummary": {  
                "status": "SUCCEEDED",  
                "startAt": 1486678853.415,  
                "lastUpdatedAt": 1486678853.415,  
                "queuedAt": 1486678853.415,  
                "executionNumber": 4357680912  
            },  
            "jobId": "011"  
        }  
    ]  
}
```

```
        "jobId": "011"
    },
{
    "jobExecutionSummary": {
        "status": "CANCELED",
        "startAt": 1486593196.378,
        "lastUpdatedAt": 1486593196.378,
        "queuedAt": 1486593196.378,
        "executionNumber": 2143174250
    },
    "jobId": "010"
}
]
```

Describe job execution

Run the **DescribeJobExecution** command to get the status of a job execution. You must specify a job ID and thing name and, optionally, an execution number to identify the job execution. The following shows how to describe a job execution:

```
aws iot describe-job-execution --job-id 017 --thing-name thingOne
```

The command returns the [JobExecution](#). For example:

```
{
    "execution": {
        "jobId": "017",
        "executionNumber": 4516820379,
        "thingArn": "arn:aws:iot:us-east-1:123456789012:thing/thingOne",
        "versionNumber": 123,
        "createdAt": 1489084805.285,
        "lastUpdatedAt": 1489086279.937,
        "startedAt": 1489086279.937,
        "status": "IN_PROGRESS",
        "approximateSecondsBeforeTimedOut": 100,
        "statusDetails": {
            "status": "IN_PROGRESS",
            "detailsMap": {
                "percentComplete": "10"
            }
        }
    }
}
```

Delete job execution

Run the **DeleteJobExecution** command to delete a job execution. You must specify a job ID, a thing name, and an execution number to identify the job execution. The following shows how to delete a job execution:

```
aws iot delete-job-execution --job-id 017 --thing-name thingOne --execution-number
1234567890 --force|--no-force
```

The command displays no output.

By default, the status of the job execution must be `QUEUED` or in a terminal state (`SUCCEEDED`, `FAILED`, `REJECTED`, `TIMED_OUT`, `REMOVED` or `CANCELED`). Otherwise, an error occurs. To delete a job execution with a status of `IN_PROGRESS`, you can set the `force` parameter to `true`.

Warning

When you delete a job execution with a status of `IN_PROGRESS`, the device that is executing the job cannot access job information or update the job execution status. Use caution and make sure that the device can recover to a valid state.

Job templates

Use job templates to preconfigure jobs that you can deploy to multiple sets of target devices. For frequently performed remote actions that you want to deploy to your devices, such as rebooting or installing an application, you can use templates to define standard configurations. When you want to perform operations such as deploying security patches and bug fixes, you can also create templates from existing jobs.

When creating a job template, you can specify the following additional configurations and resources.

- Job properties
- Job documents and targets
- Rollout and cancel criteria
- Timeout criteria

Custom and AWS managed templates

Depending on the remote action that you want to perform, you can either create a custom job template or use an AWS managed template. Use custom job templates to provide your own custom job document and create reusable jobs to deploy to your devices. AWS managed templates are job templates provided by AWS IoT Jobs for commonly performed actions. These templates have a predefined job document for some remote actions so you don't have to create your own job document. Managed templates help you to create reusable jobs for faster execution to your devices.

Topics

- [Deploy common remote operations by using AWS managed templates \(p. 649\)](#)
- [Create custom job templates \(p. 660\)](#)

Deploy common remote operations by using AWS managed templates

AWS managed templates are job templates provided by AWS for frequently performed remote actions such as rebooting, downloading a file, or installing an application on your devices. These templates have a predefined job document for each remote action so you don't have to create your own job document.

You can choose from a set of predefined configurations and create jobs using these templates without writing any additional code. Using managed templates, you can view the job document deployed to your fleets. You can create a job using these templates and further create a custom job template that you can reuse for your remote operations.

What do managed templates contain?

Each AWS managed template contains:

- A job document that specifies the name of the operation and its parameters. For example, if you use a **Download file** template, the operation name is *Download file* and the parameters can be:

- The URL of the file you want to download to your device, which can be an internet resource, or a public or pre-signed S3 URL.
- A local file path on the device to store the downloaded file.

For more information about the job documents and its parameters, see [Managed template remote actions and job documents \(p. 650\)](#).

- The environment to run the commands in the job document.

Prerequisites

For your devices to run the remote actions specified by the managed template job document, you must:

- **Install the specific software on your device**
- **Use the AWS IoT Device Client**

We recommend that you install and run the AWS IoT Device Client on your devices because it supports using all managed templates directly from the console by default.

The Device Client is an open-source software written in C++ that you can compile and install on your embedded Linux-based IoT devices. The Device Client has a *base client* and discrete *client-side features*. The base client establishes connectivity with AWS IoT over MQTT protocol and can connect with the different client-side features.

To perform remote operations on your devices, use the *client-side Jobs feature* of the Device Client. This feature contains a parser to receive the job document and job handlers that implement the remote actions specified in the job document. For more information about the Device Client and its features, see [AWS IoT Device Client](#).

When running on devices, the Device Client receives the job document and has a platform-specific implementation that it uses to run commands in the document. For more information about setting up the Device Client and using the Jobs feature, see [AWS IoT tutorials \(p. 116\)](#).

- **Use your own device software and job handlers**

Alternatively, you can write your own code for the devices by using the AWS IoT Device SDK and its library of handlers that supports the remote operations. To deploy and run jobs, make sure that the device agent libraries have been installed correctly and are running on these devices.

You can also choose to use your own handlers that can support the remote operations. For information about how you can create these handlers, see [Sample job handlers in the AWS IoT Device Client GitHub repository](#).

- **Use a supported environment**

For each managed template, you'll find information about the environment that you can use to run the remote actions. We recommend that you use the template with a supported Linux environment as specified in the template. You can use the AWS IoT Device Client to run the manage template remote actions because it supports common microprocessors and Linux environments, such as Debian and Ubuntu.

Managed template remote actions and job documents

The following shows the different managed templates and the remote actions that can be performed on the devices. For each remote action, you'll also find information about its job document and a description of the various parameters. The device agent or the job handler in the Device Client uses the template name and input parameters to perform the remote operation, which is the device behavior.

All templates, except for the `AWS-Reboot` template, require an input parameter such as a list of packages to install, or a URL to download files from. You specify a value for these parameters when creating a job using the managed template. All managed templates have the following two optional parameters in common.

`runAsUser`

This parameter specifies whether to run the job handler as another user. If it's not specified during job creation, the job handler is run as the same user as the Device Client. When you run the job handler as another user, specify a string value that's not longer than 256 characters.

`pathToHandler`

The path to the job handler running on the device. If not specified during job creation, the Device Client uses the current working directory.

The following shows the different remote actions, their job documents, and parameters they accept. All these templates support the Linux environment for running the remote operation on the device.

[AWS-Reboot](#)

Template name

`AWS-Reboot`

Template description

A managed template provided by AWS for rebooting your device.

Input parameters

This template has no required parameters. You can specify the optional parameters `runAsUser` and `pathToHandler`.

Device behavior

The device reboots successfully.

Job document

The following shows the job document and its latest version. The template shows the path to the job handler and the shell script, `reboot.sh`, that the job handler must run to reboot the device.

```
{  
  "version": "1.0",  
  "steps": [  
    {  
      "action": {  
        "name": "Reboot",  
        "type": "runHandler",  
        "input": {  
          "handler": "reboot.sh",  
          "path": "${aws:iot:parameter:pathToHandler}"  
        },  
        "runAsUser": "${aws:iot:parameter:runAsUser}"  
      }  
    }  
  ]  
}
```

AWS-Download-File

Template name

AWS-Download-File

Template description

A managed template provided by AWS for downloading a file.

Input parameters

This template has the following required parameters. You can also specify the optional parameters `runAsUser` and `pathToHandler`.

downloadUrl

The URL to download the file from, which can be an internet resource, an object in Amazon S3 that can be publicly accessed, or an object in Amazon S3 that can only be accessed by your device using a presigned URL. For more information about using presigned URLs and granting permissions, see [Presigned URLs \(p. 639\)](#).

filePath

A local file path that shows the location in the device to store the downloaded file.

Device behavior

The device downloads the file from the specified location, verifies that the download is complete, and stores it locally.

Job document

The following shows the job document and its latest version. The template shows the path to the job handler and the shell script, `download-file.sh`, that the job handler must run to download the file. It also shows the required parameters `downloadUrl` and `filePath`.

```
{
  "version": "1.0",
  "steps": [
    {
      "action": {
        "name": "Download-File",
        "type": "runHandler",
        "input": {
          "handler": "download-file.sh",
          "args": [
            "${aws:iot:parameter:downloadUrl}",
            "${aws:iot:parameter:filePath}"
          ],
          "path": "${aws:iot:parameter:pathToHandler}"
        },
        "runAsUser": "${aws:iot:parameter:runAsUser}"
      }
    }
  ]
}
```

AWS-Install-Application

Template name

AWS-Install-Application

Template description

A managed template provided by AWS for installing one or more applications.

Input parameters

This template has the following required parameter, packages. You can also specify the optional parameters runAsUser and pathToHandler.

packages

A space-separated list of one or more applications to be installed.

Device behavior

The device installs the applications as specified in the job document.

Job document

The following shows the job document and its latest version. The template shows the path to the job handler and the shell script, install-packages.sh, that the job handler must run to download the file. It also shows the required parameter packages.

```
{
  "version": "1.0",
  "steps": [
    {
      "action": {
        "name": "Install-Application",
        "type": "runHandler",
        "input": {
          "handler": "install-packages.sh",
          "args": [
            "${aws:iot:parameter:packages}"
          ],
          "path": "${aws:iot:parameter:pathToHandler}"
        },
        "runAsUser": "${aws:iot:parameter:runAsUser}"
      }
    }
  ]
}
```

AWS–Remove–Application

Template name

AWS–Remove–Application

Template description

A managed template provided by AWS for uninstalling one or more applications.

Input parameters

This template has the following required parameter, packages. You can also specify the optional parameters runAsUser and pathToHandler.

packages

A space-separated list of one or more applications to be uninstalled.

Device behavior

The device uninstalls the applications as specified in the job document.

Job document

The following shows the job document and its latest version. The template shows the path to the job handler and the shell script, `remove-packages.sh`, that the job handler must run to download the file. It also shows the required parameter `packages`.

```
{  
    "version": "1.0",  
    "steps": [  
        {  
            "action": {  
                "name": "Remove-Application",  
                "type": "runHandler",  
                "input": {  
                    "handler": "remove-packages.sh",  
                    "args": [  
                        "${aws:iot:parameter:packages}"  
                    ],  
                    "path": "${aws:iot:parameter:pathToHandler}"  
                },  
                "runAsUser": "${aws:iot:parameter:runAsUser}"  
            }  
        }  
    ]  
}
```

AWS-Start-Application

Template name

AWS-Start-Application

Template description

A managed template provided by AWS for starting one or more services.

Input parameters

This template has the following required parameter, `services`. You can also specify the optional parameters `runAsUser` and `pathToHandler`.

services

A space-separated list of one or more applications to be started.

Device behavior

The specified applications start running on the device.

Job document

The following shows the job document and its latest version. The template shows the path to the job handler and the shell script, `start-services.sh`, that the job handler must run to start the system services. It also shows the required parameter `services`.

```
{  
    "version": "1.0",  
    "steps": [  
        {  
            "action": {  
                "name": "Start-Application",  
                "type": "runHandler",  
                "input": {  
                    "handler": "start-services.sh",  
                    "args": [  
                        "${aws:iot:parameter:services}"  
                    ],  
                    "path": "${aws:iot:parameter:pathToHandler}"  
                },  
                "runAsUser": "${aws:iot:parameter:runAsUser}"  
            }  
        }  
    ]  
}
```

```
        "handler": "start-services.sh",
        "args": [
            "${aws:iot:parameter:services}"
        ],
        "path": "${aws:iot:parameter:pathToHandler}"
    },
    "runAsUser": "${aws:iot:parameter:runAsUser}"
}
]
}
```

AWS–Stop–Application

Template name

AWS–Stop–Application

Template description

A managed template provided by AWS for stopping one or more services.

Input parameters

This template has the following required parameter, `services`. You can also specify the optional parameters `runAsUser` and `pathToHandler`.

services

A space-separated list of one or more applications to be stopped.

Device behavior

The specified applications stop running on the device.

Job document

The following shows the job document and its latest version. The template shows the path to the job handler and the shell script, `stop-services.sh`, that the job handler must run to stop the system services. It also shows the required parameter `services`.

```
{
    "version": "1.0",
    "steps": [
        {
            "action": {
                "name": "Stop-Application",
                "type": "runHandler",
                "input": {
                    "handler": "stop-services.sh",
                    "args": [
                        "${aws:iot:parameter:services}"
                    ],
                    "path": "${aws:iot:parameter:pathToHandler}"
                },
                "runAsUser": "${aws:iot:parameter:runAsUser}"
            }
        }
    ]
}
```

AWS–Restart–Application

Template name

AWS-Restart-Application

Template description

A managed template provided by AWS for stopping and restarting one or more services.

Input parameters

This template has the following required parameter, `services`. You can also specify the optional parameters `runAsUser` and `pathToHandler`.

`services`

A space-separated list of one or more applications to be restarted.

Device behavior

The specified applications are stopped and then restarted on the device.

Job document

The following shows the job document and its latest version. The template shows the path to the job handler and the shell script, `restart-services.sh`, that the job handler must run to restart the system services. It also shows the required parameter `services`.

```
{  
  "version": "1.0",  
  "steps": [  
    {  
      "action": {  
        "name": "Restart-Application",  
        "type": "runHandler",  
        "input": {  
          "handler": "restart-services.sh",  
          "args": [  
            "${aws:iot:parameter:services}"  
          ],  
          "path": "${aws:iot:parameter:pathToHandler}"  
        },  
        "runAsUser": "${aws:iot:parameter:runAsUser}"  
      }  
    }  
  ]  
}
```

Topics

- [Create AWS managed templates by using the AWS Management Console \(p. 656\)](#)
- [Create AWS managed templates by using the AWS CLI \(p. 658\)](#)

Create AWS managed templates by using the AWS Management Console

Use the AWS Management Console to get information about AWS managed templates and create a job by using these templates. You can then save the job you create as your own custom template.

Get details about managed templates

You can get information about the different managed templates that are available to use from the AWS IoT console.

1. To see your available managed templates, go to the [Job templates hub of the AWS IoT console](#) and choose the **Managed templates** tab.
2. Choose a managed template to view its details.

The details page contains the following information:

- Name, description, and Amazon Resource Name (ARN) of the managed template.
- The environment on which the remote operations can be performed, such as Linux.
- The JSON job document that specifies the path to the job handler and the commands to run on the device. For example, the following shows an example job document for the **AWS-Reboot** template. The template shows the path to the job handler and the shell script, `reboot.sh`, that the job handler must run to reboot the device.

```
{  
    "version": "1.0",  
    "steps": [  
        {  
            "action": {  
                "name": "Reboot",  
                "type": "runHandler",  
                "input": {  
                    "handler": "reboot.sh",  
                    "path": "${aws:iot:parameter:pathToHandler}"  
                },  
                "runAsUser": "${aws:iot:parameter:runAsUser}"  
            }  
        }  
    ]  
}
```

For more information about the job document and its parameters for various remote actions, see [Managed template remote actions and job documents \(p. 650\)](#).

- The latest version of the job document.

Create a job using managed templates

You can use the console to choose an AWS managed template to use to create a job. This section shows you how.

You can also start the job creation workflow and then choose the AWS managed template that you want to use while creating the job. For more information about this workflow, see [Create and manage jobs by using the AWS Management Console \(p. 640\)](#).

1. Choose your AWS managed template

Go to the [Job templates hub of the AWS IoT console](#), choose the **Managed templates** tab, and then choose your template.

2. Create a job using your managed template

1. In the details page of your template, choose **Create job**.

The console switches to the **Custom job properties** step of the **Create job** workflow where your template configuration has been added.

2. Enter a unique alphanumeric job name, and optional description and tags, and then choose **Next**.
3. Choose the things or thing groups as job targets that you want to run in this job.

In the **Job document** section, your template is displayed with its configuration settings.

4. On the **Job configuration** page, choose the job type as continuous or a snapshot job. A snapshot job is complete when it finishes its run on the target devices and groups. A continuous job applies to thing groups and runs on any device that you add to a specified target group.
5. Continue to add any additional configurations for your job and then review and create your job. For information about the additional configurations, see:
 - [Job rollout and abort configuration \(p. 676\)](#)
 - [Job executions timeout configuration \(p. 678\)](#)

Create custom job templates from managed templates

You can use an AWS managed template and a custom job as a starting point to create your own custom job template. To create a custom job template, first create a job from your AWS managed template as described in the previous section.

You can then save the custom job as a template to create your own custom job template. To save as template:

1. Go to the [Job hub of the AWS IoT console](#) and choose the job containing your managed template.
2. Choose **Save as a job template** and then create your custom job template. For more information about creating a custom job template, see [Create a job template from an existing job \(p. 661\)](#).

Create AWS managed templates by using the AWS CLI

Use the AWS CLI to get information about AWS managed templates and create a job by using these templates. You can further save the job as a template and then create your own custom template.

Get details about managed templates

The following AWS CLI command gets details about a specified job template. Specify the job template name and an optional template version. If the template version is not specified, the predefined, default version is returned.

```
aws iot describe-managed-job-template \
    --template-name template-name
```

The command displays the following output.

```
{
  "description": "string",
  "document": "string",
  "documentParameters": [
    {
      "description": "string",
      "example": "string",
      "key": "string",
      "optional": boolean,
      "regex": "string"
    }
  ],
  "environments": [ "string" ],
  "templateArn": "string",
  "templateName": "string",
  "templateVersion": "string"
}
```

For more information, see [DescribeManagedJobTemplate](#).

List managed templates

The following AWS CLI command lists all of the job templates in your AWS account.

```
aws iot list-managed-job-templates
```

The command displays the following output.

```
{
  "managedJobTemplates": [
    {
      "description": "string",
      "environments": [ "string" ],
      "templateArn": "string",
      "templateName": "string",
      "templateVersion": "string"
    }
  ],
  "nextToken": "string"
}
```

To retrieve an additional set of results, use the value of the `nextToken` field. For more information, see [ListManagedJobTemplates](#).

Create a job by using managed templates

The following AWS CLI command creates a job from a job template. It targets a device named `thingOne` and specifies the Amazon Resource Name (ARN) of the managed template to use as the basis for the job. You can override advanced configurations, such as timeout and cancel configurations, by passing the associated parameters of the `create-job` command.

```
aws iot create-job \
  --targets arn:aws:iot:region:123456789012:thing/thingOne \
  --job-template-arn arn:aws:iot:region::jobtemplate/managed-template-name
```

where:

- `region` is the AWS Region and,
- `managed-template-name` is the name of the managed template, such as `AWS-Restart-Application:1.0`.

Create a custom job template from managed templates

1. Create a job using a managed template as described in the previous section.
2. Create a custom job template by using the ARN of the job that you created. For more information, see [Create a job template from an existing job \(p. 663\)](#).

Create custom job templates

You can create job templates by using the AWS CLI and the AWS IoT console. You can also create jobs from job templates by using the AWS CLI, the AWS IoT console, and Fleet Hub for AWS IoT Device Management web applications. For more information about working with job templates in Fleet Hub applications, see [Working with job templates in Fleet Hub for AWS IoT Device Management](#).

Topics

- [Create custom job templates by using the AWS Management Console \(p. 660\)](#)
- [Create custom job templates by using the AWS CLI \(p. 662\)](#)

Create custom job templates by using the AWS Management Console

This topic explains how to create, delete, and view details about job templates by using the AWS IoT console.

Create a custom job template

You can either create an original custom job template or create a job template from an existing job. You can also create a custom job template from an existing job that was created using an AWS managed template. For more information, see [Create custom job templates from managed templates \(p. 658\)](#).

Create an original job template

1. Start creating your job template

1. Go to the [Job templates hub of the AWS IoT console](#) and choose the **Custom templates** tab.
2. Choose **Create job template**.

Note

You can also navigate to the **Job templates** page from the **Related services** page under **Fleet Hub**.

2. Specify job template properties

In the **Create job template** page, enter an alphanumeric identifier for your job name and an alphanumeric description to provide additional details about the template.

Note

We don't recommend using personally identifiable information in your job IDs or descriptions.

3. Provide job document

Provide a JSON job file that is either stored in an S3 bucket or as an inline job document that is specified within the job. This job file will become the job document when you create a job using this template.

If the job file is stored in an S3 bucket, enter the S3 URL or choose **Browse S3**, and then navigate to your job document and select it.

Note

You can select only S3 buckets in your current Region.

4. Continue to add any additional configurations for your job and then review and create your job. For information about the additional configurations, see:

- [Job rollout and abort configuration \(p. 676\)](#)
- [Job executions timeout configuration \(p. 678\)](#)

Create a job template from an existing job

1. Choose your job

1. Go to the [Job hub of the AWS IoT console](#) and choose the job that you want to use as the basis for your job template.
2. Choose **Save as a job template**.

Note

Optionally, you can choose a different job document or edit the advanced configurations from the original job, and then choose **Create job template**. Your new job template appears on the **Job templates** page.

2. Specify job template properties

In the **Create job template** page, enter an alphanumeric identifier for your job name and an alphanumeric description to provide additional details about the template.

Note

The job document is the job file that you specified when creating the template. If the job document is specified within the job instead of an S3 location, you can see the job document in the details page of this job.

3. Continue to add any additional configurations for your job and then review and create your job. For information about the additional configurations, see:

- [Job rollout and abort configuration \(p. 676\)](#)
- [Job executions timeout configuration \(p. 678\)](#)

Create a job from a custom job template

You can create a job from a custom job template by going to the details page of your job template as described in this topic. You can also create a job or by choosing the job template you want to use when running the job creation workflow. For more information, see [Create and manage jobs by using the AWS Management Console \(p. 640\)](#).

This topic shows how to create a job from the details page of a custom job template. You can also create a job from an AWS managed template. For more information, see [Create a job using managed templates \(p. 657\)](#).

1. Choose your custom job template

Go to the [Job templates hub of the AWS IoT console](#) and choose the **Custom templates** tab, and then choose your template.

2. Create a job using your custom template

To create a job:

1. In the details page of your template, choose **Create job**.

The console switches to the **Custom job properties** step of the **Create job** workflow where your template configuration has been added.

2. Enter a unique alphanumeric job name, and optional description and tags, and then choose **Next**.

3. Choose the things or thing groups as job targets that you want to run in this job.

In the **Job document** section, your template is displayed with its configuration settings. If you want to use a different job document, choose **Browse** and select a different bucket and document. Choose **Next**.

4. On the **Job configuration** page, choose the job type as continuous or a snapshot job. A snapshot job is complete when it finishes its run on the target devices and groups. A continuous job applies to thing groups and runs on any device that you add to a specified target group.
5. Continue to add any additional configurations for your job and then review and create your job. For information about the additional configurations, see:
 - [Job rollout and abort configuration \(p. 676\)](#)
 - [Job executions timeout configuration \(p. 678\)](#)

You can also create jobs from job templates with Fleet Hub web applications. For information about creating jobs in Fleet Hub, see [Working with job templates in Fleet Hub for AWS IoT Device Management](#).

Delete a job template

To delete a job template, first go to the [Job templates hub of the AWS IoT console](#) and choose the **Custom templates** tab. Then, choose the job template you want to delete and choose **Delete**.

Note

A deletion is permanent and the job template no longer appears on the **Custom templates** tab.

Create custom job templates by using the AWS CLI

This topic explains how to create, delete, and retrieve details about job templates by using the AWS CLI.

Create a job template from scratch

The following AWS CLI command shows how to create a job using a job document (*job-document.json*) stored in an S3 bucket and a role with permission to download files from Amazon S3 (*S3DownloadRole*).

```
aws iot create-job-template \
    --job-template-id 010 \
    --document-source https://s3.amazonaws.com/my-s3-bucket/job-document.json \
    --timeout-config inProgressTimeoutInMinutes=100 \
    --job-executions-rollout-config "{ \"exponentialRate\": { \"baseRatePerMinute\": 50, \"incrementFactor\": 2, \"rateIncreaseCriteria\": { \"numberOfNotifiedThings\": 1000, \"numberOfSucceededThings\": 1000}, \"maximumPerMinute\": 1000} }" \
    --abort-config "{ \"criterialist\": [ { \"action\": \"CANCEL\", \"failureType\": \"FAILED\", \"minNumberOfExecutedThings\": 100, \"thresholdPercentage\": 20}, { \"action\": \"CANCEL\", \"failureType\": \"TIMED_OUT\", \"minNumberOfExecutedThings\": 200, \"thresholdPercentage\": 50} ] }" \
    --presigned-url-config "{\"roleArn\":\"arn:aws:iam::123456789012:role/S3DownloadRole\", \"expiresInSec\":3600}"
```

The optional `timeout-config` parameter specifies the amount of time each device has to finish its execution of the job. The timer starts when the job execution status is set to `IN_PROGRESS`. If the job execution status isn't set to another terminal state before the time expires, it's set to `TIMED_OUT`.

The in-progress timer can't be updated and applies to all job executions for the job. Whenever a job execution remains in the `IN_PROGRESS` state for longer than this interval, the job execution fails and switches to the terminal `TIMED_OUT` status. AWS IoT also publishes an MQTT notification.

For more information about creating configurations about job rollouts and aborts, see [Job rollout and abort configuration](#).

Note

Job documents that are specified as Amazon S3 files are retrieved at the time you create the job. If you change the contents of the Amazon S3 file you used as the source of your job document after you create the job, what is sent to the targets of the job doesn't change.

Create a job template from an existing job

The following AWS CLI command creates a job template by specifying the Amazon Resource Name (ARN) of an existing job. The new job template uses all of the configurations specified in the job. Optionally, you can change any of the configurations in the existing job by using any of the optional parameters.

```
aws iot create-job-template \
    --job-arn arn:aws:iot:<region>:123456789012:job/<job-name> \
    --timeout-config inProgressTimeoutInMinutes=100
```

Get details about a job template

The following AWS CLI command gets details about a specified job template.

```
aws iot describe-job-template \
    --job-template-id <template-id>
```

The command displays the following output.

```
{
  "abortConfig": {
    "criteriaList": [
      {
        "action": "string",
        "failureType": "string",
        "minNumberOfExecutedThings": number,
        "thresholdPercentage": number
      }
    ]
  },
  "createdAt": number,
  "description": "string",
  "document": "string",
  "documentSource": "string",
  "jobExecutionsRolloutConfig": {
    "exponentialRate": {
      "baseRatePerMinute": number,
      "incrementFactor": number,
      "rateIncreaseCriteria": {
        "numberOfNotifiedThings": number,
        "numberOfSucceededThings": number
      }
    },
    "maximumPerMinute": number
  },
  "jobTemplateArn": "string",
  "jobTemplateId": "string",
  "presignedUrlConfig": {
```

```
        "expiresInSec": number,
        "roleArn": "string"
    },
    "timeoutConfig": {
        "inProgressTimeoutInMinutes": number
    }
}
```

List job templates

The following AWS CLI command lists all of the job templates in your AWS account.

```
aws iot list-job-templates
```

The command displays the following output.

```
{
    "jobTemplates": [
        {
            "createdAt": number,
            "description": "string",
            "jobTemplateArn": "string",
            "jobTemplateId": "string"
        }
    ],
    "nextToken": "string"
}
```

To retrieve additional pages of results, use the value of the `nextToken` field.

Delete a job template

The following AWS CLI command deletes a specified job template.

```
aws iot delete-job-template \
--job-template-id template-id
```

The command displays no output.

Create a job from a job template

The following AWS CLI command creates a job from a job template. It targets a device named `thingOne` and specifies the Amazon Resource Name (ARN) of the job template to use as the basis for the job. You can override advanced configurations, such as timeout and cancel configurations, by passing the associated parameters of the `create-job` command.

```
aws iot create-job \
--targets arn:aws:iot:region:123456789012:thing/thingOne \
--job-template-arn arn:aws:iot:region:123456789012:jobtemplate/template-id
```

Devices and jobs

Devices can communicate with AWS IoT Jobs using MQTT, HTTP Signature Version 4, or HTTP TLS. To determine the endpoint to use when your device communicates with AWS IoT Jobs, run the **DescribeEndpoint** command. For example, if you run this command:

```
aws iot describe-endpoint --endpoint-type iot:Data-ATS
```

you get a result similar to the following:

```
{  
    "endpointAddress": "a1b2c3d4e5f6g7-ats.iot.us-west-2.amazonaws.com"  
}
```

Using the MQTT protocol

Devices can communicate with AWS IoT Jobs using MQTT protocol. Devices subscribe to MQTT topics to be notified of new jobs and to receive responses from the AWS IoT Jobs service. Devices publish on MQTT topics to query or update the state of a job execution. Each device has its own general MQTT topic. For more information about publishing and subscribing to MQTT topics, see [Device communication protocols \(p. 77\)](#).

With this method of communication, your device uses its device-specific certificate and private key to authenticate with AWS IoT Jobs.

Your devices can subscribe to the following topics. *thing-name* is the name of the thing associated with the device.

- **\$aws/things/*thing-name*/jobs/notify**

Subscribe to this topic to notify you when a job execution is added or removed from the list of pending job executions.

- **\$aws/things/*thing-name*/jobs/notify-next**

Subscribe to this topic to notify you when the next pending job execution has changed.

- **\$aws/things/*thing-name*/request-name/accepted**

The AWS IoT Jobs service publishes success and failure messages on an MQTT topic. The topic is formed by appending *accepted* or *rejected* to the topic used to make the request. Here, *request-name* is the name of a request such as *Get* and the topic can be: `$aws/things/myThing/jobs/get`. AWS IoT Jobs then publishes success messages on the `$aws/things/myThing/jobs/get/accepted` topic.

- **\$aws/things/*thing-name*/request-name/rejected**

Here, *request-name* is the name of a request such as *Get*. If the request failed,

You can also use the following APIs:

- Update the status of a job execution by calling the [UpdateJobExecution](#) API.
- Query the status of a job execution by calling the [DescribeJobExecution](#) API.
- Retrieve a list of pending job executions by calling the [GetPendingJobExecutions](#) API.
- Retrieve the next pending job execution by calling the [DescribeJobExecution](#) API with *jobId* as *\$next*.
- Get and start the next pending job execution by calling the [StartNextPendingJobExecution](#) API.

Using HTTP Signature Version 4

Devices can communicate with AWS IoT Jobs using HTTP Signature Version 4 on port 443. This is the method used by the AWS SDKs and CLI. For more information about those tools, see [AWS CLI Command Reference: iot-jobs-data](#) or [AWS SDKs and Tools](#) and refer to the `IoTJobsDataPlane` section for your preferred language.

With this method of communication, your device uses IAM credentials to authenticate with AWS IoT Jobs.

The following commands are available using this method:

- **DescribeJobExecution**

```
aws iot-jobs-data describe-job-execution ...
```

- **GetPendingJobExecutions**

```
aws iot-jobs-data get-pending-job-executions ...
```

- **StartNextPendingJobExecution**

```
aws iot-jobs-data start-next-pending-job-execution ...
```

- **UpdateJobExecution**

```
aws iot-jobs-data update-job-execution ...
```

Using HTTP TLS

Devices can communicate with AWS IoT Jobs using HTTP TLS on port 8443 using a third-party software client that supports this protocol.

With this method, your device uses X.509 certificate-based authentication (for example, its device-specific certificate and private key).

The following commands are available using this method:

- **DescribeJobExecution**
- **GetPendingJobExecutions**
- **StartNextPendingJobExecution**
- **UpdateJobExecution**

Topics

- [Programming devices to work with jobs \(p. 666\)](#)

Programming devices to work with jobs

The examples in this section use MQTT to illustrate how a device works with the AWS IoT Jobs service. Alternatively, you could use the corresponding API or CLI commands. For these examples, we assume a device called `MyThing` that subscribes to the following MQTT topics:

- `$aws/things/MyThing/jobs/notify` (or `$aws/things/MyThing/jobs/notify-next`)
- `$aws/things/MyThing/jobs/get/accepted`
- `$aws/things/MyThing/jobs/get/rejected`
- `$aws/things/MyThing/jobs/jobID/get/accepted`
- `$aws/things/MyThing/jobs/jobID/get/rejected`

If you are using code signing for AWS IoT your device code must verify the signature of your code file. The signature is in the job document in the codesign property. For more information about verifying a code file signature, see [Device Agent Sample](#).

Topics

- [Device workflow \(p. 667\)](#)
- [Jobs workflow \(p. 668\)](#)
- [Jobs notifications \(p. 671\)](#)

Device workflow

A device can handle jobs that it runs using either of the following ways.

- **Get the next job**

1. When a device first comes online, it should subscribe to the device's `notify-next` topic.
2. Call the [DescribeJobExecution](#) MQTT API with `jobId $next` to get the next job, its job document, and other details, including any state saved in `statusDetails`. If the job document has a code file signature, you must verify the signature before proceeding with processing the job request.
3. Call the [UpdateJobExecution](#) MQTT API to update the job status. Or, to combine this and the previous step in one call, the device can call [StartNextPendingJobExecution](#).
4. (Optional) You can add a step timer by setting a value for `stepTimeoutInMinutes` when you call either [UpdateJobExecution](#) or [StartNextPendingJobExecution](#)
5. Perform the actions specified by the job document using the [UpdateJobExecution](#) MQTT API to report on the progress of the job.
6. Continue to monitor the job execution by calling the [DescribeJobExecution](#) MQTT API with this `jobId`. If the job execution is deleted, [DescribeJobExecution](#) returns a `ResourceNotFoundException`.

The device should be able to recover to a valid state if the job execution is canceled or deleted while the device is running the job.

7. Call the [UpdateJobExecution](#) MQTT API when finished with the job to update the job status and report success or failure.
8. Because this job's execution status has been changed to a terminal state, the next job available for execution (if any) changes. The device is notified that the next pending job execution has changed. At this point, the device should continue as described in step 2.

If the device remains online, it continues to receive notifications of the next pending job execution, including its job execution data, when it completes a job or a new pending job execution is added. When this occurs, the device continues as described in step 2.

- **Pick from available jobs**

1. When a device first comes online, it should subscribe to the thing's `notify` topic.
2. Call the [GetPendingJobExecutions](#) MQTT API to get a list of pending job executions.
3. If the list contains one or more job executions, pick one.
4. Call the [DescribeJobExecution](#) MQTT API to get the job document and other details, including any state saved in `statusDetails`.
5. Call the [UpdateJobExecution](#) MQTT API to update the job status. If the `includeJobDocument` field is set to `true` in this command, the device can skip the previous step and retrieve the job document at this point.
6. Optionally, you can add a step timer by setting a value for `stepTimeoutInMinutes` when you call [UpdateJobExecution](#).

7. Perform the actions specified by the job document using the [UpdateJobExecution](#) MQTT API to report on the progress of the job.
8. Continue to monitor the job execution by calling the [DescribeJobExecution](#) MQTT API with this jobId. If the job execution is canceled or deleted while the device is running the job, the device should be capable of recovering to a valid state.
9. Call the [UpdateJobExecution](#) MQTT API when finished with the job to update the job status and to report success or failure.

If the device remains online, it is notified of all pending job executions when a new pending job execution becomes available. When this occurs, the device can continue as described in step 2.

If the device is unable to execute the job, it calls the [UpdateJobExecution](#) MQTT API to update the job status to REJECTED.

Jobs workflow

The following shows the different steps in the jobs workflow from starting a new job to reporting the completion status of a job execution.

Start a new job

When a new job is created, AWS IoT Jobs publishes a message on the `$aws/things/thing-name/jobs/notify` topic for each target device.

The message contains the following information:

```
{
  "timestamp":1476214217017,
  "jobs":{
    "QUEUED": [
      {
        "jobId":"0001",
        "queuedAt":1476214216981,
        "lastUpdatedAt":1476214216981,
        "versionNumber" : 1
      }
    ]
  }
}
```

The device receives this message on the '`$aws/things/thingName/jobs/notify`' topic when the job execution is queued.

Get job information

To get more information about a job execution, the device calls the [DescribeJobExecution](#) MQTT API with the `includeJobDocument` field set to `true` (the default).

If the request is successful, the AWS IoT Jobs service publishes a message on the `$aws/things/MyThing/jobs/0023/get/accepted` topic:

```
{
  "clientToken" : "client-001",
  "timestamp" : 1489097434407,
  "execution" : {
    "approximateSecondsBeforeTimedOut": number,
    "jobId" : "023",
    "status" : "QUEUED",
    "queuedAt" : 1489097374841,
    "lastUpdatedAt" : 1489097374841,
```

```

        "versionNumber" : 1,
        "jobDocument" : {
            < contents of job document >
        }
    }
}

```

If the request fails, the AWS IoT Jobs service publishes a message on the `$aws/things/MyThing/jobs/0023/get/rejected` topic.

The device now has the job document that it can use to perform the remote operations for the job. If the job document contains an Amazon S3 presigned URL, the device can use that URL to download any required files for the job.

Report job execution status

As the device is executing the job, it can call the [UpdateJobExecution](#) MQTT API to update the status of the job execution.

For example, a device can update the job execution status to `IN_PROGRESS` by publishing the following message on the `$aws/things/MyThing/jobs/0023/update` topic:

```
{
    "status":"IN_PROGRESS",
    "statusDetails": {
        "progress":"50%"
    },
    "expectedVersion":"1",
    "clientToken":"client001"
}
```

Jobs responds by publishing a message to the `$aws/things/MyThing/jobs/0023/update/accepted` or `$aws/things/MyThing/jobs/0023/update/rejected` topic:

```
{
    "clientToken":"client001",
    "timestamp":1476289222841
}
```

The device can combine the two previous requests by calling [StartNextPendingJobExecution](#). That gets and starts the next pending job execution and allows the device to update the job execution status. This request also returns the job document when there is a job execution pending.

If the job contains a [TimeoutConfig](#), the in-progress timer starts running. You can also set a step timer for a job execution by setting a value for `stepTimeoutInMinutes` when you call [UpdateJobExecution](#). The step timer applies only to the job execution that you update. You can set a new value for this timer each time you update a job execution. You can also create a step timer when you call [StartNextPendingJobExecution](#). If the job execution remains in the `IN_PROGRESS` status for longer than the step timer interval, it fails and switches to the terminal `TIMED_OUT` status. The step timer has no effect on the in-progress timer that you set when you create a job.

The `status` field can be set to `IN_PROGRESS`, `SUCCEEDED`, or `FAILED`. You cannot update the status of a job execution that is already in a terminal state.

Report execution completed

When the device has finished executing the job, it calls the [UpdateJobExecution](#) MQTT API. If the job was successful, set `status` to `SUCCEEDED` and, in the message payload, in `statusDetails`, add other

information about the job as name-value pairs. The in-progress and step timers end when the job execution is complete.

For example:

```
{
    "status": "SUCCEEDED",
    "statusDetails": {
        "progress": "100%"
    },
    "expectedVersion": "2",
    "clientToken": "client-001"
}
```

If the job was not successful, set `status` to `FAILED` and, in `statusDetails`, add information about the error that occurred:

```
{
    "status": "FAILED",
    "statusDetails": {
        "errorCode": "101",
        "errorMsg": "Unable to install update"
    },
    "expectedVersion": "2",
    "clientToken": "client-001"
}
```

Note

The `statusDetails` attribute can contain any number of name-value pairs.

When the AWS IoT Jobs service receives this update, it publishes a message on the `$aws/things/MyThing/jobs/notify` topic to indicate the job execution is complete:

```
{
    "timestamp": 1476290692776,
    "jobs": {}
}
```

Additional jobs

If there are other job executions pending for the device, they are included in the message published to `$aws/things/MyThing/jobs/notify`.

For example:

```
{
    "timestamp": 1476290692776,
    "jobs": {
        "QUEUED": [
            {
                "jobId": "0002",
                "queuedAt": 1476290646230,
                "lastUpdatedAt": 1476290646230
            }
        ],
        "IN_PROGRESS": [
            {
                "jobId": "0003",
                "queuedAt": 1476290646230,
                "lastUpdatedAt": 1476290646230
            }
        ]
    }
}
```

Jobs notifications

The AWS IoT Jobs service publishes MQTT messages to reserved topics when jobs are pending or when the first job execution in the list changes. Devices can keep track of pending jobs by subscribing to these topics.

Job notification types

Job notifications are published to MQTT topics as JSON payloads. There are two kinds of notifications:

- A `ListNotification` contains a list of no more than 10 pending job executions. The job executions in this list have status values of either `IN_PROGRESS` or `QUEUED`. They are sorted by status (`IN_PROGRESS` job executions before `QUEUED` job executions) and then by the times when they were queued.

A `ListNotification` is published whenever one of the following criteria is met.

- A new job execution is queued or changes to a non-terminal status (`IN_PROGRESS` or `QUEUED`).
- An old job execution changes to a terminal status (`FAILED`, `SUCCEEDED`, `CANCELED`, `TIMED_OUT`, `REJECTED`, or `REMOVED`).
- A `NextNotification` contains summary information about the one job execution that is next in the queue.

A `NextNotification` is published whenever the first job execution in the list changes.

- A new job execution is added to the list as `QUEUED`, and it is the first one in the list.
- The status of an existing job execution that was not the first one in the list changes from `QUEUED` to `IN_PROGRESS` and becomes the first one in the list. (This happens when there are no other `IN_PROGRESS` job executions in the list or when the job execution whose status changes from `QUEUED` to `IN_PROGRESS` was queued earlier than any other `IN_PROGRESS` job execution in the list.)
- The status of the job execution that is first in the list changes to a terminal status and is removed from the list.

For more information about publishing and subscribing to MQTT topics, see [the section called “Device communication protocols” \(p. 77\)](#).

Note

Notifications are not available when you use HTTP Signature Version 4 or HTTP TLS to communicate with jobs.

Job pending

The AWS IoT Jobs service publishes a message on an MQTT topic when a job is added to or removed from the list of pending job executions for a thing or the first job execution in the list changes:

- `$aws/things/thingName/jobs/notify`
- `$aws/things/thingName/jobs/notify-next`

The messages contain the following example payloads:

```
$aws/things/thingName/jobs/notify:
```

```
{  
    "timestamp" : 10011,  
    "jobs" : {  
        "IN_PROGRESS" : [ {  
            "jobId" : "other-job",  
            "queuedAt" : 10003,
```

```

        "lastUpdatedAt" : 10009,
        "executionNumber" : 1,
        "versionNumber" : 1
    } ],
    "QUEUED" : [ {
        "jobId" : "this-job",
        "queuedAt" : 10011,
        "lastUpdatedAt" : 10011,
        "executionNumber" : 1,
        "versionNumber" : 0
    } ]
}
}

```

\$aws/things/*thingName*/jobs/notify-next:

```
{
    "timestamp" : 10011,
    "execution" : {
        "jobId" : "other-job",
        "status" : "IN_PROGRESS",
        "queuedAt" : 10009,
        "lastUpdatedAt" : 10009,
        "versionNumber" : 1,
        "executionNumber" : 1,
        "jobDocument" : {"c":"d"}
    }
}
```

Possible job execution status values are QUEUED, IN_PROGRESS, FAILED, SUCCEEDED, CANCELED, TIMED_OUT, REJECTED, and REMOVED.

The following series of examples show the notifications that are published to each topic as job executions are created and change from one state to another.

First, one job, called job1, is created. This notification is published to the jobs/notify topic:

```
{
    "timestamp": 1517016948,
    "jobs": {
        "QUEUED": [
            {
                "jobId": "job1",
                "queuedAt": 1517016947,
                "lastUpdatedAt": 1517016947,
                "executionNumber": 1,
                "versionNumber": 1
            }
        ]
    }
}
```

This notification is published to the jobs/notify-next topic:

```
{
    "timestamp": 1517016948,
    "execution": {
        "jobId": "job1",
        "status": "QUEUED",
        "queuedAt": 1517016947,
        "lastUpdatedAt": 1517016947,
        "versionNumber": 1,
```

```

        "executionNumber": 1,
        "jobDocument": {
            "operation": "test"
        }
    }
}

```

When another job is created (job2), this notification is published to the jobs/notify topic:

```

{
    "timestamp": 1517017192,
    "jobs": {
        "QUEUED": [
            {
                "jobId": "job1",
                "queuedAt": 1517016947,
                "lastUpdatedAt": 1517016947,
                "executionNumber": 1,
                "versionNumber": 1
            },
            {
                "jobId": "job2",
                "queuedAt": 1517017191,
                "lastUpdatedAt": 1517017191,
                "executionNumber": 1,
                "versionNumber": 1
            }
        ]
    }
}

```

A notification is not published to the jobs/notify-next topic because the next job in the queue (job1) has not changed. When job1 starts to execute, its status changes to IN_PROGRESS. No notifications are published because the list of jobs and the next job in the queue have not changed.

When a third job (job3) is added, this notification is published to the jobs/notify topic:

```

{
    "timestamp": 1517017906,
    "jobs": {
        "IN_PROGRESS": [
            {
                "jobId": "job1",
                "queuedAt": 1517016947,
                "lastUpdatedAt": 1517017472,
                "startedAt": 1517017472,
                "executionNumber": 1,
                "versionNumber": 2
            }
        ],
        "QUEUED": [
            {
                "jobId": "job2",
                "queuedAt": 1517017191,
                "lastUpdatedAt": 1517017191,
                "executionNumber": 1,
                "versionNumber": 1
            },
            {
                "jobId": "job3",
                "queuedAt": 1517017905,
                "lastUpdatedAt": 1517017905,
                "executionNumber": 1,
                "versionNumber": 1
            }
        ]
    }
}

```

```

        "versionNumber": 1
    }
}
}
```

A notification is not published to the `jobs/notify-next` topic because the next job in the queue is still `job1`.

When `job1` is complete, its status changes to `SUCCEEDED`, and this notification is published to the `jobs/notify` topic:

```
{
  "timestamp": 1517186269,
  "jobs": {
    "QUEUED": [
      {
        "jobId": "job2",
        "queuedAt": 1517017191,
        "lastUpdatedAt": 1517017191,
        "executionNumber": 1,
        "versionNumber": 1
      },
      {
        "jobId": "job3",
        "queuedAt": 1517017905,
        "lastUpdatedAt": 1517017905,
        "executionNumber": 1,
        "versionNumber": 1
      }
    ]
  }
}
```

At this point, `job1` has been removed from the queue, and the next job to be executed is `job2`. This notification is published to the `jobs/notify-next` topic:

```
{
  "timestamp": 1517186269,
  "execution": {
    "jobId": "job2",
    "status": "QUEUED",
    "queuedAt": 1517017191,
    "lastUpdatedAt": 1517017191,
    "versionNumber": 1,
    "executionNumber": 1,
    "jobDocument": {
      "operation": "test"
    }
  }
}
```

If `job3` must begin executing before `job2` (which is not recommended), the status of `job3` can be changed to `IN_PROGRESS`. If this happens, `job2` is no longer next in the queue, and this notification is published to the `jobs/notify-next` topic:

```
{
  "timestamp": 1517186779,
  "execution": {
    "jobId": "job3",
    "status": "IN_PROGRESS",
```

```
    "queuedAt": 1517017905,
    "startedAt": 1517186779,
    "lastUpdatedAt": 1517186779,
    "versionNumber": 2,
    "executionNumber": 1,
    "jobDocument": {
        "operation": "test"
    }
}
```

No notification is published to the `jobs/notify` topic because no job has been added or removed.

If the device rejects job2 and updates its status to `REJECTED`, this notification is published to the `jobs/notify` topic:

```
{
    "timestamp": 1517189392,
    "jobs": {
        "IN_PROGRESS": [
            {
                "jobId": "job3",
                "queuedAt": 1517017905,
                "lastUpdatedAt": 1517186779,
                "startedAt": 1517186779,
                "executionNumber": 1,
                "versionNumber": 2
            }
        ]
    }
}
```

If job3 (which is still in progress) is force deleted, this notification is published to the `jobs/notify` topic:

```
{
    "timestamp": 1517189551,
    "jobs": {}
}
```

At this point, the queue is empty. This notification is published to the `jobs/notify-next` topic:

```
{
    "timestamp": 1517189551
}
```

Job configurations

You can have the following additional configurations for each job that you deploy to the specified targets.

- **Rollout:** This configuration defines how many devices receive the job document every minute.
- **Abort:** If a certain number of devices don't receive the job document, use this configuration to cancel the job and avoid sending a bad update to an entire fleet.
- **Timeout:** If a response is not received from your job targets within a certain duration, the job can fail. You can keep track of the job that's running on these devices.

By using these configurations, you can track and monitor the status of your job execution and avoid a bad update from being sent to an entire fleet. The rollout and abort configurations are related to your job whereas the timeout configuration is related to an execution of the job.

Specify additional configurations

You can specify these additional configurations when creating a job or a job template by using the console or the Jobs API.

The following describes more about these configurations.

- [Job rollout and abort configuration \(p. 676\)](#)
- [Job executions timeout configuration \(p. 678\)](#)

Job rollout and abort configuration

You can specify how quickly targets are notified of a pending job execution. You can create a staged rollout to better manage updates, reboots, and other operations.

The following field can be added to the `CreateJob` request to specify the maximum number of job targets to inform per minute. This example sets a static rollout rate.

```
"jobExecutionRolloutConfig": {  
    "maximumPerMinute": "integer"  
}
```

You can also set a variable rollout rate with the `exponentialRate` field. The following example creates a rollout that has an exponential rate.

```
"jobExecutionsRolloutConfig": {  
    "exponentialRate": {  
        "baseRatePerMinute": integer,  
        "incrementFactor": integer,  
        "rateIncreaseCriteria": {  
            "numberOfNotifiedThings": integer, // Set one or the other  
            "numberOfSucceededThings": integer // of these two values.  
        },  
        "maximumPerMinute": integer  
    }  
}
```

AWS IoT jobs can be deployed using variable rollout rates as various criteria and thresholds are met. Job rollouts can also be aborted if the number of failed jobs matches a set of criteria. These rollout configurations give you more granular control over a job's blast radius. Job rollout rate criteria are set at job creation through the [JobExecutionsRolloutConfig](#) object. Job abort criteria are set at job creation through the [AbortConfig](#) object.

Using job rollout rates

You set a job rollout rate by configuring the `ExponentialRolloutRate` property of the `JobExecutionsRolloutConfig` object when you run the `CreateJob` API. The following example sets a variable rollout rate by using the `exponentialRate` parameter.

```
{
```

```

...
    "jobExecutionsRolloutConfig": {
        "exponentialRate": {
            "baseRatePerMinute": 50,
            "incrementFactor": 2,
            "rateIncreaseCriteria": {
                "numberOfNotifiedThings": 1000,
                "numberOfSucceededThings": 1000
            },
            "maximumPerMinute": 1000
        }
    }
...
}

```

The `baseRatePerMinute` parameter specifies the rate at which the jobs are executed until the `numberOfNotifiedThings` or `numberOfSucceededThings` threshold has been met.

The `incrementFactor` parameter specifies the exponential factor by which the rollout rate increases after the `numberOfNotifiedThings` or `numberOfSucceededThings` threshold has been met.

The `rateIncreaseCriteria` parameter is an object that specifies either the `numberOfNotifiedThings` or `numberOfSucceededThings` threshold.

The `maximumPerMinute` parameter specifies the upper limit of the rate at which job executions can occur. Valid values range from 1 to 1000. This parameter is required when you pass an `ExponentialRate` object. In a variable rate rollout, this value establishes the upper limit of a job rollout rate.

A job rollout with the configuration above would start at a rate of 50 job executions per minute. It would continue at that rate until either 1000 things have received job execution notifications (if a value for `numberOfNotifiedThings` has been specified) or 1000 successful job executions have occurred (if a value for `numberOfSucceededThings` has been specified).

The following table illustrates how the rollout would proceed over the first four increments.

Rollout rate per minute	50	100	200	400
Number of notified devices or successful executions	1000	2000	3000	4000

The following configuration sets a static rollout rate.

```

{
...
    "jobExecutionsRolloutConfig": {
        "maximumPerMinute": 1000
    }
...
}
```

The `maximumPerMinute` parameter specifies the upper limit of the rate at which job executions can occur. Valid values range from 1 to 1000. This parameter is optional. If you don't specify a value, the default value of 1000 is used.

Using job rollout abort configurations

You set up a job abort condition by configuring the optional [AbortConfig](#) object when you run the [CreateJob](#) API. This section describes the effect that the following sample configuration would have on a job rollout that was experiencing multiple failed executions.

```
"abortConfig": {  
    "criteriaList": [  
        {  
            "action": "CANCEL",  
            "failureType": "FAILED",  
            "minNumberOfExecutedThings": 100,  
            "thresholdPercentage": 20  
        },  
        {  
            "action": "CANCEL",  
            "failureType": "TIMED_OUT",  
            "minNumberOfExecutedThings": 200,  
            "thresholdPercentage": 50  
        }  
    ]  
}
```

The `action` parameter specifies the action to take when the abort criteria have been met. This parameter is required, and `CANCEL` is the only valid value.

The `failureType` parameter specifies which failure types should trigger a job abort. Valid values are `FAILED`, `REJECTED`, `TIMED_OUT`, and `ALL`.

The `minNumberOfExecutedThings` parameter specifies the number of completed job executions that must occur before the service checks to see if the job abort criteria have been met. In this example, AWS IoT doesn't check to see if a job abort should occur until at least 100 devices have completed job executions.

The `thresholdPercentage` parameter specifies the total number of executed things that initiate a job abort. In this example, AWS IoT initiates a job abort and cancels the job rollout if at least 20% of all completed executions have failed in any way after 100 executions have completed.

Note

Deletion of job executions affects the computation value of the total completed execution. When a job aborts, the service creates an automated comment and `reasonCode` to differentiate a user-driven cancellation from a job abort cancellation.

Job executions timeout configuration

Job timeouts make it possible to be notified whenever a job execution gets stuck in the `IN_PROGRESS` state for an unexpectedly long period of time. There are two types of timers: in-progress timers and step timers. When the job is `IN_PROGRESS`, you can monitor and track the progress of your job execution.

When you create a job, you can set a value for the `inProgressTimeoutInMinutes` property of the optional [TimeoutConfig](#) object. The in-progress timer can't be updated and applies to all job executions for the job. Whenever a job execution remains in the `IN_PROGRESS` status for longer than this interval, the job execution fails and switches to the terminal `TIMED_OUT` status. AWS IoT also publishes an MQTT notification.

You can also set a step timer for a job execution by setting a value for `stepTimeoutInMinutes` when you call [UpdateJobExecution](#). The step timer applies only to the job execution that you update. You can

set a new value for this timer each time you update a job execution. You can also create a step timer when you call [StartNextPendingJobExecution](#). If the job execution remains in the IN_PROGRESS status for longer than the step timer interval, it fails and switches to the terminal TIMED_OUT status. The step timer has no effect on the in-progress timer that you set when you create a job.

The following diagram and description is an example that illustrates the ways in which in-progress timeouts and step timeouts interact with each other in a 20-minute timeout period.

Job Creation: CreateJob sets an in-progress timer that expires in twenty minutes. This timer applies to all job executions and can't be updated.

12:00 PM: The job execution starts and switches to IN_PROGRESS status. The in-progress timer starts to run.

12:05 PM: UpdateJobExecution creates a step timer with a value of 7 minutes. If a new step timer isn't created, the job execution times out at 12:12 PM.

12:10 PM: UpdateJobExecution creates a new step timer with a value of 5 minutes. The previous step timer is discarded. If a new step timer isn't created, the job execution times out at 12:15 PM.

12:13 PM: UpdateJobExecution creates a new step timer with a value of 9 minutes. The job execution times out at 12:20 because the in-progress timer expires at 12:20. The step timer can't exceed the absolute bound created by the in-progress timer.

UpdateJobExecution can also discard a step timer that has already been created by creating a new step timer with a value of -1.

AWS IoT jobs APIs

AWS IoT Jobs API can either be used for management and control of jobs, or for devices executing those jobs.

Job management and control uses an HTTPS protocol API. Devices can use either an MQTT or an HTTPS protocol API. The HTTPS API is designed for a low volume of calls typical when creating and tracking jobs. It usually opens a connection for a single request, and then closes the connection after the response is received. The MQTT API allows long polling. It is designed for large amounts of traffic that can scale to millions of devices.

Each AWS IoT Jobs HTTPS API has a corresponding command that allows you to call the API from the AWS CLI. The commands are lowercase, with hyphens between the words that make up the name of the API. For example, you can invoke the CreateJob API on the CLI by typing:

```
aws iot create-job ...
```

Job management and control API

To determine the `endpoint-url` parameter for your CLI commands, run this command.

```
aws iot describe-endpoint --endpoint-type=iot:Jobs
```

This command returns the following output.

```
{
```

```
"endpointAddress": "account-specific-prefix.jobs.iot.aws-region.amazonaws.com"  
}
```

Note

The Jobs endpoint doesn't support ALPN z-amzn-http-ca.

Use the following API operations or CLI commands:

- [AssociateTargetsWithJob](#) or `associate-targets-with-job`
- [CancelJob](#) or `cancel-job`
- [CancelJobExecution](#) or `cancel-job-execution`
- [CreateJob](#) or `create-job`
- [DeleteJob](#) or `delete-job`
- [DeleteJobExecution](#) or `delete-job-execution`
- [DescribeJob](#) or `describe-job`
- [DescribeJobExecution](#) or `describe-job-execution`
- [GetJobDocument](#) or `get-job-document`
- [ListJobExecutionsForJob](#) or `list-job-executions-for-job`
- [ListJobExecutionsForThing](#) or `list-job-executions-for-thing`
- [ListJobs](#) or `list-jobs`
- [UpdateJob](#) or `update-job`

Job management and control data types

The following data types are used by management and control applications to communicate with AWS IoT Jobs.

- [Job](#) or `job`
- [JobSummary](#) or `job-summary`
- [JobExecution](#) or `job-execution`
- [JobExecutionSummary](#) or `job-execution-summary`
- [JobExecutionSummaryForJob](#) or `job-execution-summary-for-job`
- [JobExecutionSummaryForThing](#) or `job-execution-summary-for-thing`.

Jobs device MQTT and HTTPS APIs

Jobs device commands can be issued by publishing MQTT messages to the [Reserved topics used for Jobs commands \(p. 101\)](#). Your client is automatically subscribed to the response message topics of these commands, which means that the message broker will publish response message topics to the client that published the command message whether your client has subscribed to the response message topics or not. When subscribing to the job and jobExecution event topics for your fleet-monitoring solution, first enable [job and job execution events \(p. 982\)](#) to receive any events on the cloud side.

Because the message broker publishes response messages, even without an explicit subscription to them, your client must be configured to receive and identify the messages it receives. Your client must also confirm that the [`thingName`](#) in the incoming message topic applies to the client's thing name before the client acts on the message.

Messages that AWS IoT sends in response to MQTT Jobs API command messages are charged to your account, whether you subscribed to them explicitly or not.

The following commands are available over the MQTT and HTTPS protocols.

- [GetPendingJobExecutions](#) or `get-pending-job-executions`
- [StartNextPendingJobExecution](#) or `start-next-pending-job-execution`
- [DescribeJobExecution](#) or `describe-job-execution`
- [UpdateJobExecution](#) or `update-job-execution`

When you use the MQTT protocol, you can also perform the following updates:

JobExecutionsChanged

Sent whenever a job execution is added to or removed from the list of pending job executions for a thing.

Use the topic:

`$aws/things/thingName/jobs/notify`

Message payload:

```
{  
  "jobs" : {  
    "JobExecutionState": [  
      DescribeJobExecution ... ]  
  },  
  "timestamp": timestamp,  
}
```

NextJobExecutionChanged

Sent whenever there is a change to which job execution is next on the list of pending job executions for a thing, as defined for [DescribeJobExecution](#) with jobId `$next`. This message is not sent when the next job's execution details change, only when the next job that would be returned by [DescribeJobExecution](#) with jobId `$next` has changed. Consider job executions J1 and J2 with state QUEUED. J1 is next on the list of pending job executions. If the state of J2 is changed to IN_PROGRESS while the state of J1 remains unchanged, then this notification is sent and contains details of J2.

Use the topic:

`$aws/things/thingName/jobs/notify-next`

Message payload:

```
{  
  "execution" : JobExecution,  
  "timestamp": timestamp,  
}
```

Jobs device MQTT and HTTPS data types

The following data types are used to communicate with the AWS IoT Jobs service over the MQTT and HTTPS protocols.

- [JobExecution](#) or `job-execution`
- [JobExecutionState](#) or `job-execution-state`
- [JobExecutionSummary](#) or `job-execution-summary`

For errors occurred during an operation, you get an error response that contains information about the error.

ErrorResponse

Contains information about an error that occurred during an AWS IoT Jobs service operation.

Following shows the syntax of this operation:

```
{  
    "code": "ErrorCode",  
    "message": "string",  
    "clientToken": "string",  
    "timestamp": timestamp,  
    "executionState": JobExecutionState  
}
```

Following is a description of this `ErrorResponse`:

`code`

`ErrorCode` can be set to:

`InvalidTopic`

The request was sent to a topic in the AWS IoT Jobs namespace that does not map to any API.
`InvalidJson`

The contents of the request could not be interpreted as valid UTF-8-encoded JSON.
`InvalidRequest`

The contents of the request were invalid. For example, this code is returned when an `UpdateJobExecution` request contains invalid status details. The message contains details about the error.

`InvalidStateTransition`

An update attempted to change the job execution to a state that is invalid because of the job execution's current state (for example, an attempt to change a request in state `SUCCEEDED` to state `IN_PROGRESS`). In this case, the body of the error message also contains the `executionState` field.

`ResourceNotFound`

The `JobExecution` specified by the request topic does not exist.

`VersionMismatch`

The expected version specified in the request does not match the version of the job execution in the AWS IoT Jobs service. In this case, the body of the error message also contains the `executionState` field.

`InternalError`

There was an internal error during the processing of the request.

`RequestThrottled`

The request was throttled.

`TerminalStateReached`

Occurs when a command to describe a job is performed on a job that is in a terminal state.

`message`

An error message string.

`clientToken`

An arbitrary string used to correlate a request with its reply.

`timestamp`

The time, in seconds since the epoch.

`executionState`

A `JobExecutionState` object. This field is included only when the `code` field has the value `InvalidStateTransition` or `VersionMismatch`. This makes it unnecessary in these cases to perform a separate `DescribeJobExecution` request to obtain the current job execution status data.

Job limits

For job limit information, see [AWS AWS IoT Device Management endpoints and quotas](#) in the AWS General Reference.

AWS IoT secure tunneling

When devices are deployed behind restricted firewalls at remote sites, you need a way to gain access to those devices for troubleshooting, configuration updates, and other operational tasks. Secure tunneling helps customers establish bidirectional communication to remote devices over a secure connection that is managed by AWS IoT. Secure tunneling does not require updates to your existing inbound firewall rule, so you can keep the same security level provided by firewall rules at a remote site.

For example, a sensor device located at a factory that is a couple hundred miles away is having trouble measuring the factory temperature. You can use secure tunneling to open and quickly start a session to that sensor device. After you have identified the problem (for example, a bad configuration file), you can reset the file and restart the sensor device through the same session. Compared to a more traditional troubleshooting (for example, sending a technician to the factory to investigate the sensor device), secure tunneling decreases incident response and recovery time and operational costs.

Secure tunneling concepts

Client access token (CAT)

A pair of tokens generated by secure tunneling when a new tunnel is created. The CAT is used by the source and destination devices to connect to the Secure Tunneling service.

Destination application

The application that runs on the destination device. For example, the destination application can be an SSH daemon for establishing an SSH session using secure tunneling.

Destination device

The remote device you want to access.

Device agent

An IoT application that connects to the AWS IoT device gateway and listens for new tunnel notifications over MQTT.

Local proxy

A software proxy that runs on the source and destination devices and relays a data stream between the Secure Tunneling service and the device application. The local proxy can be run in source mode or destination mode. For more information, see [Local proxy \(p. 693\)](#).

Source device

The device an operator uses to initiate a session to the destination device, usually a laptop or desktop computer.

Tunnel

A logical pathway through AWS IoT that enables bidirectional communication between a source device and destination device.

Secure tunnel lifecycle

Tunnels can have one of the following statuses:

- OPEN
- CLOSED

Connections can have one of the following statuses:

- CONNECTED
- DISCONNECTED

When you open a tunnel, it has a status of OPEN. The tunnel's source and destination connection status is set to DISCONNECTED. When a device (source or destination) connects to the tunnel, the corresponding connection status changes to CONNECTED, while the tunnel status remains OPEN. When a device disconnects from the tunnel, the corresponding connection status changes back to DISCONNECTED. A device can connect to and disconnect from a tunnel repeatedly as long as the tunnel remains OPEN.

A tunnel's status becomes CLOSED when you call `CloseTunnel` or the tunnel remains OPEN for longer than the `MaxLifetimeTimeout` value. You can configure `MaxLifetimeTimeout` when calling `OpenTunnel`. `MaxLifetimeTimeout` defaults to 12 hours if you do not specify a value.

After a tunnel is CLOSED, it might be visible in the AWS IoT console for at least three hours before it is deleted. While the tunnel is visible, you can call `DescribeTunnel` and `ListTunnels` to view tunnel metadata. A tunnel cannot be reopened when it is CLOSED. The Secure Tunneling service rejects attempts to connect to a CLOSED tunnel.

Controlling access to tunnels

The Secure Tunneling service provides the following service-specific actions, resources, and condition context keys for use in IAM permission policies.

Tunnel access prerequisites

- Learn how to secure AWS resources by using [IAM policies](#).
- Learn how to create and evaluate [IAM conditions](#).
- Learn how to secure AWS resources using [resource tags](#).

iot:OpenTunnel

The `iot:OpenTunnel` policy action grants a principal permission to call `OpenTunnel`. You must specify the wildcard tunnel ARN `arn:aws:iot:aws-region:aws-account-id:tunnel/*` in the Resource element of the IAM policy statement. You can specify a thing ARN (`arn:aws:iot:aws-region:aws-account-id:thing/ <thing-name>`) in the Resource element of the IAM policy statement to manage `OpenTunnel` permission for specific IoT things.

For example, the following policy statement allows you to open a tunnel to the IoT thing named `TestDevice`.

```
{  
    "Effect": "Allow",  
    "Action": "iot:OpenTunnel",  
    "Resource": [  
        "arn:aws:iot:aws-region:aws-account-id:tunnel/*",  
        "arn:aws:iot:aws-region:aws-account-id:thing/TestDevice"  
    ]  
}
```

```
}
```

The `iot:OpenTunnel` policy action supports the following condition keys:

- `iot:ThingGroupArn`
- `iot:TunnelDestinationService`
- `aws:RequestTag/tag-key`
- `aws:SecureTransport`
- `aws:TagKeys`

The following policy statement allows you to open a tunnel to the thing if the thing belongs to a thing group with a name that starts with `TestGroup` and the configured destination service on the tunnel is SSH.

```
{
    "Effect": "Allow",
    "Action": "iot:OpenTunnel",
    "Resource": [
        "arn:aws:iot:aws-region:aws-account-id:tunnel/*"
    ],
    "Condition": {
        "ForAnyValue:StringLike": {
            "iot:ThingGroupArn": [
                "arn:aws:iot:aws-region:aws-account-id:thinggroup/TestGroup*"
            ]
        },
        "ForAllValues:StringEquals": {
            "iot:TunnelDestinationService": [
                "SSH"
            ]
        }
    }
}
```

You can also use resource tags to control permission to open tunnels. For example, the following policy statement allows a tunnel to be opened if the tag key `Owner` is present with a value of `Admin` and no other tags are specified.

```
{
    "Effect": "Allow",
    "Action": "iot:OpenTunnel",
    "Resource": [
        "arn:aws:iot:aws-region:aws-account-id:tunnel/*"
    ],
    "Condition": {
        "StringEquals": {
            "aws:RequestTag/Owner": "Admin"
        },
        "ForAllValues:StringEquals": {
            "aws:TagKeys": "Owner"
        }
    }
}
```

iot:DescribeTunnel

The `iot:DescribeTunnel` policy action grants a principal permission to call `DescribeTunnel`. You can specify a fully qualified tunnel ARN (for example, `arn:aws:iot:aws-region: aws-account-`

`id:tunnel/tunnel-id`) or use the wildcard tunnel ARN (`arn:aws:iot:aws-region:aws-account-id:tunnel/*`) in the Resource element of the IAM policy statement.

The `iot:DescribeTunnel` policy action supports the following condition keys:

- `aws:ResourceTag/tag-key`
- `aws:SecureTransport`

The following policy statement allows you to call `DescribeTunnel` if the requested tunnel is tagged with the key `Owner` with a value of `Admin`.

```
{
    "Effect": "Allow",
    "Action": "iot:DescribeTunnel",
    "Resource": [
        "arn:aws:iot:aws-region:aws-account-id:tunnel/*"
    ],
    "Condition": {
        "StringEquals": {
            "aws:ResourceTag/Owner": "Admin"
        }
    }
}
```

iot>ListTunnels

The `iot>ListTunnels` policy action grants a principal permission to call `ListTunnels`. You must specify the wildcard tunnel ARN (`arn:aws:iot:aws-region:aws-account-id:tunnel/*`) in the Resource element of the IAM policy statement. To manage `ListTunnels` permission on selected IoT things, you can also specify a thing ARN (`arn:aws:iot:aws-region:aws-account-id:thing/thing-name`) in the Resource element of the IAM policy statement.

The `iot>ListTunnels` policy action supports the following condition key:

- `aws:SecureTransport`

The following policy statement allows you to list tunnels for the thing named `TestDevice`.

```
{
    "Effect": "Allow",
    "Action": "iot>ListTunnels",
    "Resource": [
        "arn:aws:iot:aws-region:aws-account-id:tunnel/*",
        "arn:aws:iot:aws-region:aws-account-id:thing/TestDevice"
    ]
}
```

iot>ListTagsForResource

The `iot>ListTagsForResource` policy action grants a principal permission to call `ListTagsForResource`. You can specify a fully qualified tunnel ARN (`arn:aws:iot:aws-region:aws-account-id:tunnel/tunnel-id`) or use the wildcard tunnel ARN (`arn:aws:iot:aws-region:aws-account-id:tunnel/*`) in the Resource element of the IAM policy statement.

The `iot>ListTagsForResource` policy action supports the following condition key:

- aws:SecureTransport

iot:CloseTunnel

The `iot:CloseTunnel` policy action grants a principal permission to call `CloseTunnel`. You can specify a fully qualified tunnel ARN (`arn:aws:iot:aws-region:aws-account-id:tunnel/tunnel-id`) or use the wildcard tunnel ARN (`arn:aws:iot:aws-region:aws-account-id:tunnel/*`) in the `Resource` element of the IAM policy statement.

The `iot:CloseTunnel` policy action supports the following condition keys:

- iot:Delete
- aws:ResourceTag/*tag-key*
- aws:SecureTransport

The following policy statement allows you to call `CloseTunnel` if the request's `Delete` parameter is `false` and the requested tunnel is tagged with the key `Owner` with a value of `QATeam`.

```
{  
    "Effect": "Allow",  
    "Action": "iot:CloseTunnel",  
    "Resource": [  
        "arn:aws:iot:aws-region:aws-account-id:tunnel/*"  
    ],  
    "Condition": {  
        "Bool": {  
            "iot:Delete": "false"  
        },  
        "StringEquals": {  
            "aws:ResourceTag/Owner": "QATeam"  
        }  
    }  
}
```

iot:TagResource

The `iot:TagResource` policy action grants a principal permission to call `TagResource`. You can specify a fully qualified tunnel ARN (`arn:aws:iot:aws-region:aws-account-id:tunnel/tunnel-id`) or use the wildcard tunnel ARN (`arn:aws:iot:aws-region:aws-account-id:tunnel/*`) in the `Resource` element of the IAM policy statement.

The `iot:TagResource` policy action supports the following condition key:

- aws:SecureTransport

iot:UntagResource

The `iot:UntagResource` policy action grants a principal permission to call `UntagResource`. You can specify a fully qualified tunnel ARN (`arn:aws:iot:aws-region:aws-account-id:tunnel/tunnel-id`) or use the wildcard tunnel ARN (`arn:aws:iot:aws-region:aws-account-id:tunnel/*`) in the `Resource` element of the IAM policy statement.

The `iot:UntagResource` policy action supports the following condition key:

- aws:SecureTransport

For more information about AWS IoT security see [Identity and access management for AWS IoT \(p. 364\)](#).

AWS IoT Secure Tunneling tutorials

AWS IoT Secure tunneling helps customers establish bidirectional communication to remote devices that are behind firewall over a secure connection managed by AWS IoT. The following tutorials will help you learn how to get started and use Secure tunneling.

AWS IoT Secure tunneling tutorials

- [AWS IoT Secure Tunneling demo \(p. 689\)](#)
- [Open a tunnel and start SSH session to remote device \(p. 689\)](#)

AWS IoT Secure Tunneling demo

To quickly demo AWS IoT Secure Tunneling, use our [AWS IoT Secure Tunneling demo on GitHub](#).

Open a tunnel and start SSH session to remote device

In this tutorial, you open a tunnel and use it to start an SSH session to a remote device. The remote device is behind firewalls that block all inbound traffic, making direct SSH into the device impossible. Before you begin, make sure that you understand how to [register a device in the AWS IoT registry](#) and [connect a device to the AWS IoT device gateway](#).

Prerequisites

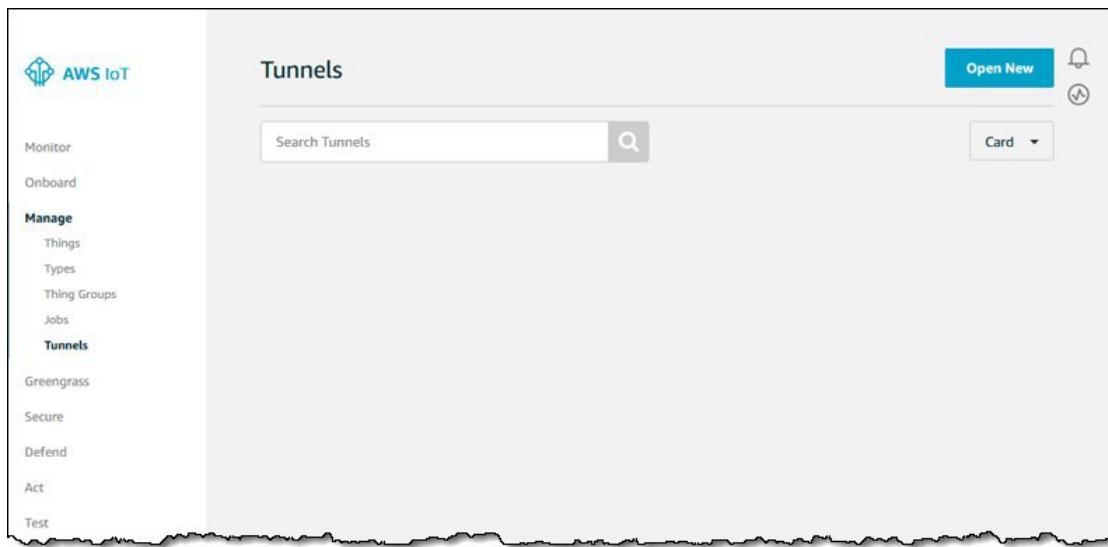
- The firewalls the remote device is behind must allow outbound traffic on port 443.
- You have created an IoT thing named `RemoteDeviceA` in the AWS IoT registry.
- You have an IoT device agent running on the remote device that connects to the AWS IoT device gateway and is configured with an MQTT topic subscription. This tutorial includes a snippet that shows you how to implement an agent. For more information, see [IoT agent snippet \(p. 701\)](#).
- You must have an SSH daemon running on the remote device.
- You have downloaded the local proxy source code from [GitHub](#) and built it for the platform of your choice. We'll refer to the built local proxy executable file as `localproxy` in this tutorial.

Open a tunnel

If you configure the destination when calling `OpenTunnel`, the Secure Tunneling service delivers the destination client access token to the remote device over MQTT and the reserved MQTT topic (`$aws/things/RemoteDeviceA/tunnels/notify`). For more information, see [Reserved topics \(p. 95\)](#). Upon receipt of the MQTT message, the IoT agent on the remote device starts the local proxy in destination mode. You can omit the destination configuration if you want to deliver the destination client access token to the remote device through another method. For more information, see [Configuring a remote device \(p. 703\)](#).

To open a tunnel in the console

1. In the [AWS IoT console](#), navigate to **Manage** and **Tunnels**.



2. Select **Open New**.
3. On the **Open a new secure tunnel** screen, enter the following:
 - a. Description: a description for your tunnel.
 - b. Thing Name: the thing you want to open a tunnel for.
 - c. Service: a service to be used on the thing (e.g. ssh, ftp etc).
 - d. Tunnel timeout configuration: specify a timeout duration for your tunnel.
 - e. Resource Tags: apply tags to your resources to help organize and identify them. Consists of a case-sensitive key-value pair.

OPEN TUNNEL

Open a new secure tunnel

AWS IoT Secure Tunneling allows you to securely connect to a deployed device and debug its software. Open a new tunnel to communicate with a remote device.

Description

SSH Debug Tunnel

Thing Name

Choose a thing to open a tunnel for:

Search things

Thing3

Thing2

Thing1

foo

Service

A service with maximum 8 characters (e.g. ssh, ftp etc) to be used on the thing.

SSH

Tunnel timeout configuration

Specify a timeout duration for your tunnel.

90 Minutes Close

Enter the timeout duration for your job execution. We support durations from 1 minute to 7 days.

Hours Minutes

90

Resource Tags

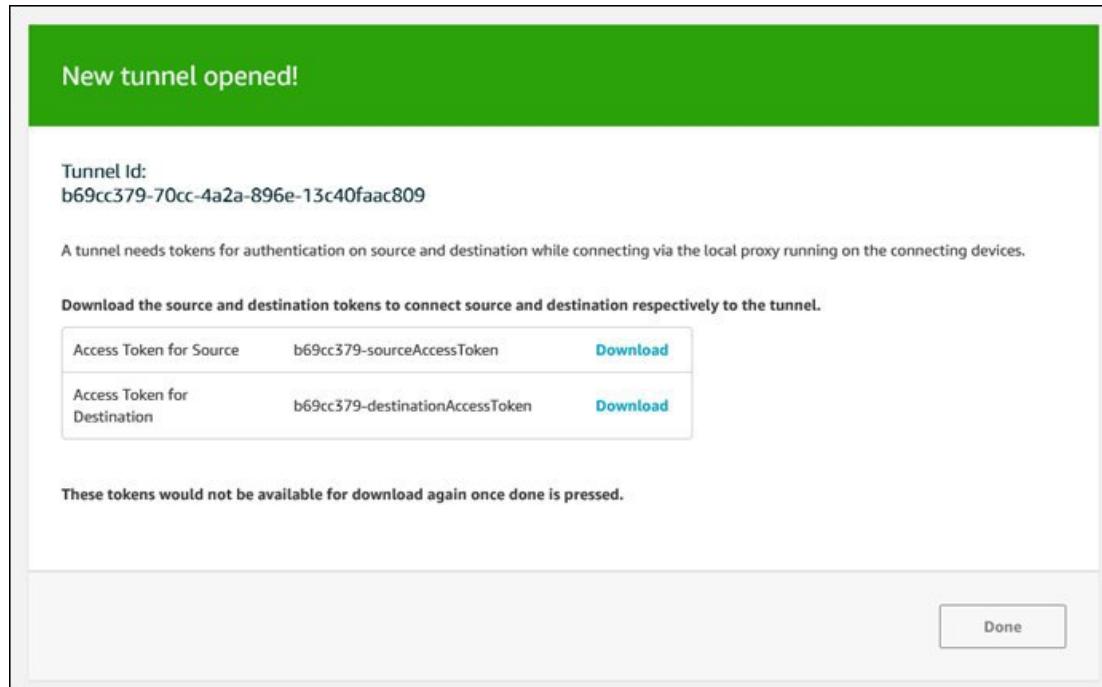
Apply tags to your resources to help organize and identify them. A tag consists of a case-sensitive key-value pair.

Tag Name	Value
Blog	IoT Day

Add tag Clear

Cancel Open New

4. Download the client access tokens for the source and destination.



5. Select Done.

Start the local proxy

Open a terminal on your laptop, copy the source client access token, and use it to start the local proxy in source mode. In the following command, the local proxy is configured to listen for new connections on port 5555.

`./localproxy -r us-east-1 -s 5555 -t source-client-access-token`

Note

The AWS Region in this command must be the same AWS Region where the tunnel was created.

`-r`

Specifies the AWS Region where your tunnel is created.

`-s`

Specifies the port to which the proxy should connect.

`-t`

Specifies the client token text.

Note

If you receive the following error, set up the CA path. For information, see [GitHub](#).

Could not perform SSL handshake with proxy server: certificate verify failed

Start an SSH session

Open another terminal and use the following command to start a new SSH session by connecting to the local proxy on port 5555.

```
ssh username@localhost -p 5555
```

You might be prompted for a password for the SSH session. When you are done with the SSH session, type **exit** to close the session.

Closing the tunnel

1. Open the [AWS IoT console](#).
2. Choose the tunnel and from **Actions**, choose **Close**. Closing the tunnel causes both local proxy instances to close.

Local proxy

The local proxy is a process that acts as the recipient or sender of incoming TCP connections. It transmits data sent by the device application through the Secure Tunneling service over a WebSocket secure connection. You can download the local proxy source from [GitHub](#). The local proxy can run in two modes: **source** or **destination**. In source mode, the local proxy runs on the same device or network as the client application that initiates the TCP connection. In destination mode, the local proxy runs on the remote device, along with the destination application. Currently, a single tunnel can support only one TCP connection at a time.

The local proxy first establishes a connection to the Secure Tunneling service. When you start the local proxy, use the **-r** argument to specify the AWS Region in which the tunnel is opened. Use the **-t** argument to pass either the source or destination client access token returned from the `OpenTunnel`. Two local proxies using the same client access token value cannot be connected at the same time.

After the WebSocket connection is established, the local proxy performs either source mode or destination mode behaviors, depending on its configuration.

By default, the local proxy attempts to reconnect to the Secure Tunneling service if any I/O errors occur or if the WebSocket connection is closed unexpectedly. This causes the TCP connection to close. If any TCP socket errors occur, the local proxy sends a message through the tunnel to notify the other side to close its TCP connection. By default, the local proxy always uses SSL communication.

After you use the tunnel, it is safe to stop the local proxy process. We recommend that you explicitly close the tunnel by calling `CloseTunnel`. Active tunnel clients might not be closed immediately after calling `CloseTunnel`.

Local proxy security best practices

When running the local proxy, follow these security best practices:

- Avoid the use of the **-t** local proxy argument to pass in an access token. We recommend that you use the `AWSIOT_TUNNEL_ACCESS_TOKEN` environment variable to set the access token for the local proxy.
- Run the local proxy executable with least privileges in the operating system or environment.
 - Avoid running the local proxy as an administrator on Windows.
 - Avoid running the local proxy as root on Linux and macOS.
- Consider running the local proxy on separate hosts, containers, sandboxes, chroot jail, or a virtualized environment.

- Build the local proxy with relevant security flags, depending on your toolchain.
- On devices with multiple network interfaces, use the `-b` argument to bind the TCP socket to the network interface used to communicate with the destination application.

Configure local proxy for devices that use web proxy

You can use local proxy on AWS IoT devices to communicate with AWS IoT secure tunneling service APIs. The local proxy transmits data sent by the device application through the secure tunneling service over a WebSocket secure connection. The local proxy can work in source or destination mode. In source mode, it runs on the same device or network that initiates the TCP connection. In destination mode, the local proxy runs on the remote device, along with the destination application. For more information, see [Local proxy \(p. 693\)](#).

The local proxy needs to connect directly to the internet to use AWS IoT secure tunneling. For a long-lived TCP connection with the secure tunneling service, the local proxy upgrades the HTTPS request to establish a WebSockets connection to one of the [secure tunneling device connection endpoints](#).

If your devices are in a network that use a web proxy, the web proxy can intercept the connections before forwarding them to the internet. To establish a long-lived connection to the secure tunneling device connection endpoints, configure your local proxy to use the web proxy as described in the [websocket specification](#).

Note

The [AWS IoT Device Client \(p. 1130\)](#) doesn't support devices that use a web proxy. To work with the web proxy, you'll need to use a local proxy and configure it to work with a web proxy as described below.

The following steps show how the local proxy works with a web proxy.

1. The local proxy sends an HTTP CONNECT request to the web proxy that contains the remote address of the secure tunneling service, along with the web proxy authentication information.
2. The web proxy will then create a long-lived connection to the remote secure tunneling endpoints.
3. The TCP connection is established and the local proxy will now work in both source and destination modes for data transmission.

To complete this procedure, perform the following steps.

- [Build the local proxy \(p. 694\)](#)
- [Configure your web proxy \(p. 694\)](#)
- [Configure and start the local proxy \(p. 695\)](#)

Build the local proxy

Open the [local proxy source code](#) in the GitHub repository and follow the instructions for building and installing the local proxy.

Configure your web proxy

The local proxy relies on the HTTP tunneling mechanism described by the [HTTP/1.1 specification](#). To comply with the specifications, your web proxy must allow devices to use the CONNECT method.

How you configure your web proxy depends on the web proxy you're using and the web proxy version. To make sure you configure the web proxy correctly, check your web proxy's documentation.

To configure your web proxy, first identify your web proxy URL and confirm whether your web proxy supports HTTP tunneling. The web proxy URL will be used later when you configure and start the local proxy.

1. Identify your web proxy URL

Your web proxy URL will be in the following format.

```
protocol://web_proxy_host_domain:web_proxy_port
```

AWS IoT secure tunneling supports only basic authentication for web proxy. To use basic authentication, you must specify the **username** and **password** as part of the web proxy URL. The web proxy URL will be in the following format.

```
protocol://username:password@web_proxy_host_domain:web_proxy_port
```

- **protocol** can be `http` or `https`. We recommend that you use `https`.
- **web_proxy_host_domain** is the IP address of your web proxy or a DNS name that resolves to the IP address of your web proxy.
- **web_proxy_port** is the port on which the web proxy is listening.
- The web proxy uses this **username** and **password** to authenticate the request.

2. Test your web proxy URL

To confirm whether your web proxy supports TCP tunneling, use a `curl` command and make sure that you get a `2xx` or a `3xx` response.

For example, if your web proxy URL is `https://server.com:1235`, use a `proxy-insecure` flag with the `curl` command because the web proxy might rely on a self-signed certificate.

```
export HTTPS_PROXY=https://server.com:1235
curl -I https://aws.amazon.com --proxy-insecure
```

If your web proxy URL has a `http` port (for example, `http://server.com:1234`), you don't have to use the `proxy-insecure` flag.

```
export HTTPS_PROXY=http://server.com:1234
curl -I https://aws.amazon.com
```

Configure and start the local proxy

To configure the local proxy to use a web proxy, you must configure the `HTTPS_PROXY` environment variable with either the DNS domain names or the IP addresses and port numbers that your web proxy uses.

After you've configured the local proxy, you can use the local proxy as explained in this [README](#) document.

Note

Your environment variable declaration is case sensitive. We recommend that you define each variable once using either all uppercase or all lowercase letters. The following examples show the environment variable declared in uppercase letters. If the same variable is specified using

both uppercase and lowercase letters, the variable specified using lowercase letters takes precedence.

The following commands show how to configure the local proxy that is running on your destination to use the web proxy and start the local proxy.

- **AWSIOT_TUNNEL_ACCESS_TOKEN:** This variable holds the client access token (CAT) for the destination.
- **HTTPS_PROXY:** This variable holds the web proxy URL or the IP address for configuring the local proxy.

The commands shown in the following examples depend on the operating system that you use and whether the web proxy is listening on an HTTP or an HTTPS port.

Web proxy listening on an HTTP port

If your web proxy is listening on an HTTP port, you can provide the web proxy URL or IP address for the **HTTPS_PROXY** variable.

Linux/macOS

In Linux or macOS, run the following commands in the terminal to configure and start the local proxy on your destination to use a web proxy listening to an HTTP port.

```
export AWSIOT_TUNNEL_ACCESS_TOKEN=${access_token}  
export HTTPS_PROXY=http://proxy.example.com:1234  
.localproxy -r us-east-1 -d 22
```

If you have to authenticate with the proxy, you must specify a **username** and **password** as part of the **HTTPS_PROXY** variable.

```
export AWSIOT_TUNNEL_ACCESS_TOKEN=${access_token}  
export HTTPS_PROXY=http://username:password@proxy.example.com:1234  
.localproxy -r us-east-1 -d 22
```

Windows

In Windows, you configure the local proxy similar to how you do for Linux or macOS, but how you define the environment variables is different from the other platforms. Run the following commands in the cmd window to configure and start the local proxy on your destination to use a web proxy listening to an HTTP port.

```
set AWSIOT_TUNNEL_ACCESS_TOKEN=${access_token}  
set HTTPS_PROXY=http://proxy.example.com:1234  
.localproxy -r us-east-1 -d 22
```

If you have to authenticate with the proxy, you must specify a **username** and **password** as part of the **HTTPS_PROXY** variable.

```
set AWSIOT_TUNNEL_ACCESS_TOKEN=${access_token}  
set HTTPS_PROXY=http://username:password@10.15.20.25:1234  
.localproxy -r us-east-1 -d 22
```

Web proxy listening on an HTTPS port

Run the following commands if your web proxy is listening on an HTTPS port.

Note

If you're using a self-signed certificate for the web proxy or if you're running the local proxy on an OS that doesn't have native OpenSSL support and default configurations, you'll have to set up your web proxy certificates as described in the [Certificate setup](#) section in the GitHub repository.

The following commands will look similar to how you configured your web proxy for an HTTP proxy, with the exception that you'll also specify the path to the certificate files that you installed as described previously.

Linux/macOS

In Linux or macOS, run the following commands in the terminal to configure the local proxy running on your destination to use a web proxy listening to an HTTPS port.

```
export AWSIOT_TUNNEL_ACCESS_TOKEN=${access_token}
export HTTPS_PROXY=http://proxy.example.com:1234
./localproxy -r us-east-1 -d 22 -c /path/to/certs
```

If you have to authenticate with the proxy, you must specify a **username** and **password** as part of the **HTTPS_PROXY** variable.

```
export AWSIOT_TUNNEL_ACCESS_TOKEN=${access_token}
export HTTPS_PROXY=http://username:password@proxy.example.com:1234
./localproxy -r us-east-1 -d 22 -c /path/to/certs
```

Windows

In Windows, run the following commands in the cmd window to configure and start the local proxy running on your destination to use a web proxy listening to an HTTP port.

```
set AWSIOT_TUNNEL_ACCESS_TOKEN=${access_token}
set HTTPS_PROXY=http://proxy.example.com:1234
.\localproxy -r us-east-1 -d 22 -c \path\to\certs
```

If you have to authenticate with the proxy, you must specify a **username** and **password** as part of the **HTTPS_PROXY** variable.

```
set AWSIOT_TUNNEL_ACCESS_TOKEN=${access_token}
set HTTPS_PROXY=http://username:password@10.15.20.25:1234
.\localproxy -r us-east-1 -d 22 -c \path\to\certs
```

Example command and output

The following shows an example of a command that you run on a Linux OS and the corresponding output. The example shows a web proxy that's listening on an HTTP port and how the local proxy can be configured to use the web proxy in both source and destination modes. Before you can run these commands, you must have already opened a tunnel and obtained the client access tokens for the source and destination. You must have also built the local proxy and configured your web proxy as described previously.

Here's an overview of the steps after you start the local proxy. The local proxy:

- Identifies the web proxy URL so that it can use the URL to connect to the proxy server.
- Establishes a TCP connection with the web proxy.

- Sends an HTTP CONNECT request to the web proxy and waits for the HTTP/1.1 200 response, which indicates that connection has been established.
- Upgrades the HTTPS protocol to WebSockets to establish a long-lived connection.
- Starts transmitting data through the connection to the secure tunneling device endpoints.

Note

The following commands used in the examples use the `verbosity` flag to illustrate an overview of the different steps described previously after you run the local proxy. We recommend that you use this flag only for testing purposes.

Running local proxy in source mode

The following commands show how to run the local proxy in source mode.

```
export AWSIOT_TUNNEL_ACCESS_TOKEN=${access_token}
export HTTPS_PROXY=http://username:password@10.15.10.25:1234
./localproxy -s 5555 -v 5 -r us-west-2
```

The following shows a sample output of running the local proxy in source mode.

```
...
Parsed basic auth credentials for the URL
Found Web proxy information in the environment variables, will use it to connect via the
proxy.

...
Starting proxy in source mode
Attempting to establish web socket connection with endpoint wss://data.tunneling.iot.us-
west-2.amazonaws.com:443
Resolved Web proxy IP: 10.10.0.11
Connected successfully with Web Proxy
Successfully sent HTTP CONNECT to the Web proxy
Full response from the Web proxy:
HTTP/1.1 200 Connection established
TCP tunnel established successfully
Connected successfully with proxy server
Successfully completed SSL handshake with proxy server
Web socket session ID: 0a109afffee745f5-00001341-000b8138-cc6c878d80e8adb0-f186064b
Web socket subprotocol selected: aws.iot.securetunneling-2.0
Successfully established websocket connection with proxy server: wss://
data.tunneling.iot.us-west-2.amazonaws.com:443
Setting up web socket pings for every 5000 milliseconds
Scheduled next read:

...
Starting web socket read loop continue reading...
Resolved bind IP: 127.0.0.1
Listening for new connection on port 5555
```

Running local proxy in destination mode

The following commands show how to run the local proxy in destination mode.

```
export AWSIOT_TUNNEL_ACCESS_TOKEN=${access_token}
export HTTPS_PROXY=http://username:password@10.15.10.25:1234
./localproxy -d 22 -v 5 -r us-west-2
```

The following shows a sample output of running the local proxy in destination mode.

```
...
Parsed basic auth credentials for the URL
Found Web proxy information in the environment variables, will use it to connect via the proxy.
...
Starting proxy in destination mode
Attempting to establish web socket connection with endpoint wss://data.tunneling.iot.us-west-2.amazonaws.com:443
Resolved Web proxy IP: 10.10.0.1
Connected successfully with Web Proxy
Successfully sent HTTP CONNECT to the Web proxy
Full response from the Web proxy:
HTTP/1.1 200 Connection established
TCP tunnel established successfully
Connected successfully with proxy server
Successfully completed SSL handshake with proxy server
Web socket session ID: 06717bffffed3fd05-00001355-000b8315-da3109a85da804dd-24c3d10d
Web socket subprotocol selected: aws.iot.securetunneling-2.0
Successfully established websocket connection with proxy server: wss://data.tunneling.iot.us-west-2.amazonaws.com:443
Seting up web socket pings for every 5000 milliseconds
Scheduled next read:
...
Starting web socket read loop continue reading...
```

Multiplex data streams in a secure tunnel

You can use multiple data streams per tunnel by using the Secure Tunneling multiplexing feature. With multiplexing, you can troubleshoot devices using multiple connections or ports (for example, a web browser that requires sending multiple HTTP and SSH data streams). You can also reduce your operational load by eliminating the need to build, deploy, and start multiple local proxies or open multiple tunnels to the same device.

Example use case

You can use the multiplexing feature in the event that a device in the field requires more than one connection to the device in order to properly troubleshoot it. For example, you might need to connect to an on-device web application to change some networking parameters, while simultaneously issuing shell commands through the terminal to verify that the device is working properly with the new networking parameters. In this scenario, you may need to connect to the device through both HTTP and SSH and transfer two parallel data streams in order to concurrently access the web application and terminal. With the multiplexing feature, these two independent streams can be transferred over the same tunnel at the same time.

How to set up a multiplexed tunnel

The following procedure will walk you through how to set up a multiplexed tunnel for troubleshooting devices using applications that require connections to multiple ports. You will set up one tunnel with two multiplexed streams: one HTTP stream and one SSH stream.

1. First, configure the destination device with configuration files. Configuration files can be provided on the device if the port mappings are unlikely to change. On the destination device, create a configuration directory called config in the same folder where the local proxy is running. Then, create a file called SSHSource.ini in this directory. The content of this file is:

```
HTTP1 = 5555
SSH1 = 3333
```

Note

You can skip this step if you prefer to specify the port mapping through the CLI or don't need to start local proxy on designated listening ports.

2. Next, configure the source device with configuration files. In the same folder as where the local proxy is running, create a configuration directory called config and give local proxy the read permission to this directory. Then, create a file called SSHDestination.ini in this directory. The content of this file is:

```
HTTP1 = 80
SSH1 = 22
```

Note

You can skip this step if you prefer to specify the port mapping through the CLI. If so, you will need to update the [tunnel agent \(p. 701\)](#) to use the new parameters.

3. Open a tunnel with the service identifier HTTP1 and SSH1. thingName is optional if your device isn't registered with AWS IoT.

```
aws iotsecuretunneling open-tunnel \
--destination-config thingName=foo,services=HTTP1,SSH1
```

A destination and source client access token will be given after this call. Note the destination_client_access_token and source_client_access_token for next steps. The output should look similar to the following.

```
{
  "tunnelId": "b2de92a3-b8ff-46c0-b0f2-afa28b00cecd",
  "tunnelArn": "arn:aws:iot:us-west-2:431600097591:tunnel/b2de92a3-b8ff-46c0-b0f2-
afa28b00cecd",
  "sourceAccessToken": "source_client_access_token",
  "destinationAccessToken": "destination_client_access_token"
}
```

4. Next, start the destination local proxy. You will be connected to the secure tunneling service upon token delivery. A local proxy running on destination devices starts in destination mode. You have two options to achieve this:

1. Start destination local proxy with configuration files from Step 1.

```
./localproxy -r us-east-1 -m dst -t destination_client_access_token
```

2. Start destination local proxy with the mapping specified through the CLI.

```
./localproxy -r us-east-1 -d HTTP1=80,SSH1=22 -t destination_client_access_token
```

5. Now, start the source local proxy. A local proxy running on source devices starts in source mode. You have three options to achieve this:

1. Start source local proxy with configuration files from Step 2.

```
./localproxy -r us-east-1 -m src -t source_client_access_token
```

- Start source local proxy with the mapping specified through the CLI.

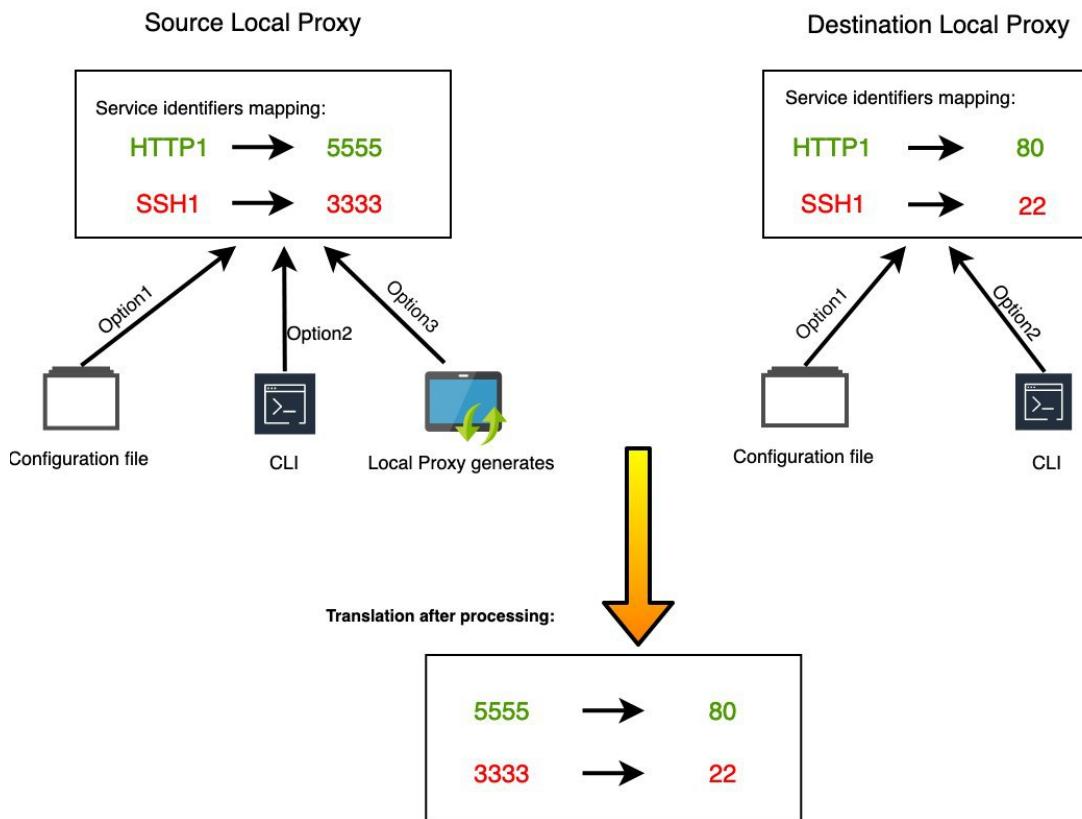
```
./localproxy -r us-east-1 -s HTTP1=5555,SSH1=3333 -t source_client_access_token
```

- Start source local proxy with no configuration files and no mapping specified from the CLI. Local proxy will pick up available ports to use and manage the mappings for you.

```
./localproxy -r us-east-1 -m src -t source_client_access_token
```

- Application data from SSH and HTTP connection can now be transferred concurrently over the multiplexed tunnel. As can be seen from the map below, the service identifier acts as a readable format to translate the port mapping between the source and destination device. With this configuration, the secure tunneling service will:

- Forward any incoming HTTP traffic from port 5555 on the source device to port 80 on the destination device.
- Forward any incoming SSH traffic from port 3333 on the source device to port 22 on the destination device.



IoT agent snippet

The IoT agent is used to receive the MQTT message that includes the client access token and start a local proxy on the remote device. You must install and run the IoT agent on the remote device if you want the

Secure Tunneling service to deliver the client access token. The IoT agent must subscribe to the following reserved IoT MQTT topic:

```
$aws/things/thing-name/tunnels/notify
```

Where *thing-name* is the name of IoT thing associated with the remote device.

The following is an example MQTT message payload:

```
{  
    "clientAccessToken": "destination-client-access-token",  
    "clientMode": "destination",  
    "region": "aws-region",  
    "services": [destination-service]  
}
```

After it receives an MQTT message, the IoT agent must start a local proxy on the remote device with the appropriate parameters.

The following Java code demonstrates how to use the [AWS IoT Device SDK](#) and [ProcessBuilder](#) from the Java library to build a simple IoT agent to work with the Secure Tunneling service.

```
// Find the IoT device endpoint for your AWS account  
final String endpoint = iotClient.describeEndpoint(new  
    DescribeEndpointRequest().withEndpointType("iot:Data-ATS")).getEndpointAddress();  
  
// Instantiate the IoT Agent with your AWS credentials  
final String thingName = "RemoteDeviceA";  
final String tunnelNotificationTopic = String.format("$aws/things/%s/tunnels/notify",  
    thingName);  
final AWSIotMqttClient mqttClient = new AWSIotMqttClient(endpoint, thingName,  
    "your_aws_access_key", "your_aws_secret_key");  
  
try {  
    mqttClient.connect();  
    final TunnelNotificationListener listener = new  
        TunnelNotificationListener(tunnelNotificationTopic);  
    mqttClient.subscribe(listener, true);  
}  
finally {  
    mqttClient.disconnect();  
}  
  
private static class TunnelNotificationListener extends AWSIotTopic {  
    public TunnelNotificationListener(String topic) {  
        super(topic);  
    }  
  
    @Override  
    public void onMessage(AWSIotMessage message) {  
        try {  
            // Deserialize the MQTT message  
            final JSONObject json = new JSONObject(message.getStringPayload());  
  
            final String accessToken = json.getString("clientAccessToken");  
            final String region = json.getString("region");  
  
            final String clientMode = json.getString("clientMode");  
            if (!clientMode.equals("destination")) {  
                throw new RuntimeException("Client mode " + clientMode + " in the MQTT  
message is not expected");  
            }  
        }
```

```
        final JSONArray servicesArray = json.getJSONArray("services");
        if (servicesArray.length() > 1) {
            throw new RuntimeException("Services in the MQTT message has more than 1
service");
        }
        final String service = servicesArray.get(0).toString();
        if (!service.equals("SSH")) {
            throw new RuntimeException("Service " + service + " is not supported");
        }

        // Start the destination local proxy in a separate process to connect to the
SSH Daemon listening port 22
        final ProcessBuilder pb = new ProcessBuilder("localproxy",
                "-t", accessToken,
                "-r", region,
                "-d", "localhost:22");
        pb.start();
    }
    catch (Exception e) {
        log.error("Failed to start the local proxy", e);
    }
}
}
```

Configuring a remote device

If you want to deliver the destination client access token to the remote device through methods other than subscribing to the reserved IoT MQTT topic, you might need two components on the remote device:

- A destination client access token listener.
- A local proxy.

The destination client access token listener should work with the client access token delivery mechanism of your choice. It must be able to start a local proxy in destination mode.

Device provisioning

AWS provides several different ways to provision a device and install unique client certificates on it. This section describes each way and how to select the best one for your IoT solution. These options are described in detail in the white paper titled, [Device Manufacturing and Provisioning with X.509 Certificates in AWS IoT Core](#).

Select the option that fits your situation best

- **You can install certificates on IoT devices before they are delivered**

If you can securely install unique client certificates on your IoT devices before they are delivered for use by the end user, you want to use [just-in-time provisioning \(JITP\) \(p. 712\)](#) or [just-in-time registration \(JITR\) \(p. 293\)](#).

Using JITP and JITR, the certificate authority (CA) used to sign the device certificate is registered with AWS IoT and is recognized by AWS IoT when the device first connects. The device is provisioned in AWS IoT on its first connection using the details of its provisioning template.

For more information on single thing, JITP, JITR, and bulk provisioning of devices that have unique certificates, see [the section called “Provisioning devices that have device certificates” \(p. 711\)](#).

- **End users or installers can use an app to install certificates on their IoT devices**

If you cannot securely install unique client certificates on your IoT device before they are delivered to the end user, but the end user or an installer can use an app to register the devices and install the unique device certificates, you want to use the [provisioning by trusted user \(p. 708\)](#) process.

Using a trusted user, such as an end user or an installer with a known account, can simplify the device manufacturing process. Instead of a unique client certificate, devices have a temporary certificate that enables the device to connect to AWS IoT for only 5 minutes. During that 5-minute window, the trusted user obtains a unique client certificate with a longer life and installs it on the device. The limited life of the claim certificate minimizes the risk of a compromised certificate.

For more information, see [the section called “Provisioning by trusted user” \(p. 708\)](#).

- **End users CANNOT use an app to install certificates on their IoT devices**

If neither of the previous options will work in your IoT solution, the [provisioning by claim \(p. 706\)](#) process is an option. With this process, your IoT devices have a claim certificate that is shared by other devices in the fleet. The first time a device connects with a claim certificate, AWS IoT registers the device using its provisioning template and issues the device its unique client certificate for subsequent access to AWS IoT.

This option enables automatic provisioning of a device when it connects to AWS IoT, but could present a larger risk in the event of a compromised claim certificate. If a claim certificate becomes compromised, you can deactivate the certificate. Deactivating the claim certificate prevents all devices with that claim certificate from being registered in the future. However; deactivating the claim certificate does not block devices that have already been provisioned.

For more information, see [the section called “Provisioning by claim” \(p. 706\)](#).

Provisioning devices in AWS IoT

When you provision a device with AWS IoT, you must create resources so your devices and AWS IoT can communicate securely. Other resources can be created to help you manage your device fleet. The following resources can be created during the provisioning process:

- An IoT thing.

IoT things are entries in the AWS IoT device registry. Each thing has a unique name and set of attributes, and is associated with a physical device. Things can be defined using a thing type or grouped into thing groups. For more information, see [Managing devices with AWS IoT \(p. 251\)](#).

Although not required, creating a thing makes it possible to manage your device fleet more effectively by searching for devices by thing type, thing group, and thing attributes. For more information, see [Fleet indexing service \(p. 732\)](#).

- An X.509 certificate.

Devices use X.509 certificates to perform mutual authentication with AWS IoT. You can register an existing certificate or have AWS IoT generate and register a new certificate for you. You associate a certificate with a device by attaching it to the thing that represents the device. You must also copy the certificate and associated private key onto the device. Devices present the certificate when connecting to AWS IoT. For more information, see [Authentication \(p. 279\)](#).

- An IoT policy.

IoT policies define the operations a device can perform in AWS IoT. IoT policies are attached to device certificates. When a device presents the certificate to AWS IoT, it is granted the permissions specified in the policy. For more information, see [Authorization \(p. 312\)](#). Each device needs a certificate to communicate with AWS IoT.

AWS IoT supports automated fleet provisioning using provisioning templates. Provisioning templates describe the resources AWS IoT requires to provision your device. Templates contain variables that enable you to use one template to provision multiple devices. When you provision a device, you specify values for the variables specific to the device using a dictionary or *map*. To provision another device, specify new values in the dictionary.

You can use automated provisioning whether or not your devices have unique certificates (and their associated private key).

Fleet provisioning APIs

There are several categories of APIs used in fleet provisioning:

- These control plane functions create and manage the fleet provisioning templates and configure trusted user policies.
 - [CreateProvisioningTemplate](#)
 - [CreateProvisioningTemplateVersion](#)
 - [DeleteProvisioningTemplate](#)
 - [DeleteProvisioningTemplateVersion](#)
 - [DescribeProvisioningTemplate](#)
 - [DescribeProvisioningTemplateVersion](#)
 - [ListProvisioningTemplates](#)
 - [ListProvisioningTemplateVersions](#)
 - [UpdateProvisioningTemplate](#)
- Trusted users can use this control plane function to generate a temporary onboarding claim. This temporary claim is passed to the device during Wi-Fi config or similar method.
 - [CreateProvisioningClaim](#).
- The MQTT API used during the provisioning process by devices with a provisioning claim certificate embedded in a device, or passed to it by a trusted user.

- the section called “[CreateCertificateFromCsr](#)” (p. 727)
- the section called “[CreateKeysAndCertificate](#)” (p. 728)
- the section called “[RegisterThing](#)” (p. 730)

Provisioning devices that don't have device certificates using fleet provisioning

By using AWS IoT fleet provisioning, AWS IoT can generate and securely deliver device certificates and private keys to your devices when they connect to AWS IoT for the first time. AWS IoT provides client certificates that are signed by the Amazon Root certificate authority (CA).

There are two ways to use fleet provisioning:

- By claim
- By trusted user

Provisioning by claim

Devices can be manufactured with a provisioning claim certificate and private key (which are special purpose credentials) embedded in them. If these certificates are registered with AWS IoT, the service can exchange them for unique device certificates that the device can use for regular operations. This process includes the following steps:

Before you deliver the device

1. Call [CreateProvisioningTemplate](#) to create a provisioning template. This API returns a template ARN. For more information, see [Device provisioning MQTT API](#) (p. 726).

You can also create a fleet provisioning template in the AWS IoT console.

- a. From the navigation pane, choose **Onboard**, then choose **Fleet provisioning templates**.
- b. Choose **Create** and follow the prompts.
2. Create certificates and associated private keys to be used as provisioning claim certificates.
3. Register these certificates with AWS IoT and associate an IoT policy that restricts the use of the certificates. The following example IoT policy restricts the use of the certificate associated with this policy to provisioning devices.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": ["iot:Connect"],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": ["iot:Publish","iot:Receive"],
            "Resource": [
                "arn:aws:iot:aws-region:aws-account-id:topic/$aws/certificates/create/*",
                "arn:aws:iot:aws-region:aws-account-id:topic/$aws/provisioning-templates/templateName/provision/*"
            ]
        }
    ]
}
```

```

    },
    {
        "Effect": "Allow",
        "Action": "iot:Subscribe",
        "Resource": [
            "arn:aws:iot:aws-region:aws-account-id:topicfilter/$aws/certificates/
create/*",
            "arn:aws:iot:aws-region:aws-account-id:topicfilter/$aws/provisioning-
templates/templateName/provision/*"
        ]
    }
}

```

4. Give the AWS IoT service permission to create or update IoT resources such as things and certificates in your account when provisioning devices. Do this by attaching the `AWSIoTThingsRegistration` managed policy to an IAM role (called the provisioning role) that trusts the AWS IoT service principal.
5. Manufacture the device with the provisioning claim certificate securely embedded in it.

The device is now ready to be delivered to where it will be installed for use.

Important

Provisioning claim private keys should be secured at all times, including on the device. We recommend that you use AWS IoT CloudWatch metrics and logs to monitor for indications of misuse. If you detect misuse, disable the provisioning claim certificate so it cannot be used for device provisioning.

To initialize the device for use

1. The device uses the [AWS IoT Device SDKs, Mobile SDKs, and AWS IoT Device Client \(p. 1127\)](#) to connect to and authenticate with AWS IoT using the provisioning claim certificate that is installed on the device.

Note

For security, the `certificateOwnershipToken` returned by [CreateCertificateFromCsr \(p. 727\)](#) and [CreateKeysAndCertificate \(p. 728\)](#) expires after one hour. [RegisterThing \(p. 730\)](#) must be called before the `certificateOwnershipToken` expires. If the token expires, the device can call [CreateCertificateFromCsr \(p. 727\)](#) or [CreateKeysAndCertificate \(p. 728\)](#) again to generate a new certificate.

2. The device obtains a permanent certificate and private key by using one of these options. The device will use the certificate and key for all future authentication with AWS IoT.
 - a. Call [CreateKeysAndCertificate \(p. 728\)](#) to create a new certificate and private key using the AWS certificate authority.

Or

 - b. Call [CreateCertificateFromCsr \(p. 727\)](#) to generate a certificate from a certificate signing request that keeps its private key secure.
 3. From the device, call [RegisterThing \(p. 730\)](#) to register the device with AWS IoT and create cloud resources.
- The Fleet Provisioning service creates cloud resources such as things, thing groups, and attributes, as defined in the provisioning template.
4. After saving the permanent certificate on the device, the device must disconnect from the session that it initiated with the provisioning claim certificate and reconnect using the permanent certificate.

The device is now ready to communicate normally with AWS IoT.

Provisioning by trusted user

In many cases, a device connects to AWS IoT for the first time when a trusted user, such as an end user or installation technician, uses a mobile app to configure the device in its deployed location.

Important

You must manage the trusted user's access and permission to perform this procedure. One way to do this is to provide and maintain an account for the trusted user that authenticates them and grants them access to the AWS IoT features and APIs required to perform this procedure.

Before you deliver the device

1. Call [CreateProvisioningTemplate](#) to create a provisioning template and return its `templateArn` and `templateName`.
2. Create an IAM role that is used by a trusted user to initiate the provisioning process. The provisioning template allows only that user to provision a device. For example:

```
{  
    "Effect": "Allow",  
    "Action": [  
        "iot:CreateProvisioningClaim",  
    ],  
    "Resource": [  
        "arn:aws:aws-region:aws-account-id:provisioningtemplate/templateName"  
    \]  
}
```

3. Give the AWS IoT service permission to create or update IoT resources, such as things and certificates in your account when provisioning devices. You do this by attaching the `AWSIoTThingsRegistration` managed policy to an IAM role (called the *provisioning role*) that trusts the AWS IoT service principal.
4. Provide the means to identify your trusted users, such as by providing them with an account that can authenticate them and authorize their interactions with the AWS APIs necessary to register their devices.

To initialize the device for use

1. A trusted user signs in to your provisioning mobile app or web service.
2. The mobile app or web application uses the IAM role and calls [CreateProvisioningClaim](#) to obtain a temporary provisioning claim certificate from AWS IoT.

Note

For security, the temporary provisioning claim certificate that `CreateProvisioningClaim` returns expires after five minutes. The following steps must successfully return a valid certificate before the temporary provisioning claim certificate expires. Temporary provisioning claim certificates do not appear in your account's list of certificates.

3. The mobile app or web application supplies the temporary provisioning claim certificate to the device along with any required configuration information, such as Wi-Fi credentials.
4. The device uses the temporary provisioning claim certificate to connect to AWS IoT using the [AWS IoT Device SDKs, Mobile SDKs, and AWS IoT Device Client \(p. 1127\)](#).
5. The device obtains a permanent certificate and private key by using one of these options within five minutes of connecting to AWS IoT with the temporary provisioning claim certificate. The device will use the certificate and key these options return for all future authentication with AWS IoT.
 - a. Call [CreateKeysAndCertificate \(p. 728\)](#) to create a new certificate and private key using the AWS certificate authority.

Or

- b. Call [CreateCertificateFromCsr \(p. 727\)](#) to generate a certificate from a certificate signing request that keeps its private key secure.

Note

Remember [CreateKeysAndCertificate \(p. 728\)](#) or [CreateCertificateFromCsr \(p. 727\)](#) must return a valid certificate within five minutes of connecting to AWS IoT with the temporary provisioning claim certificate.

6. The device calls [RegisterThing \(p. 730\)](#) to register the device with AWS IoT and create cloud resources.

The Fleet Provisioning service creates cloud resources such as IoT things, thing groups, and attributes, as defined in the provisioning template.

7. After saving the permanent certificate on the device, the device must disconnect from the session that it initiated with the temporary provisioning claim certificate and reconnect using the permanent certificate.

The device is now ready to communicate normally with AWS IoT.

Using pre-provisioning hooks with the AWS CLI

The following procedure creates a provisioning template with pre-provisioning hooks. The Lambda function used here is an example that can be modified.

To create and apply a pre-provisioning hook to a provisioning template

1. Create a Lambda function that has a defined input and output. Lambda functions are highly customizable the `allowProvisioning` and `parameterOverrides` are required for creating pre-provisioning hooks. For more information about creating Lambda functions, see [Using AWS Lambda with the AWS Command Line Interface](#).

The following is an example of a Lambda function output:

```
{  
  "allowProvisioning": True,  
  "parameterOverrides": {  
    "incomingKey0": "incomingValue0",  
    "incomingKey1": "incomingValue1"  
  }  
}
```

2. AWS IoT uses resource-based policies to call Lambda, so you must give AWS IoT permission to call your Lambda function.

The following is an example using [add-permission](#) give IoT permission to your Lambda.

```
aws lambda add-permission \  
  --function-name myLambdaFunction \  
  --statement-id iot-permission \  
  --action lambda:InvokeFunction \  
  --principal iot.amazonaws.com
```

3. Add a pre-provisioning hook to a template using either the [create-provisioning-template](#) or [update-provisioning-template](#) command.

The following CLI example uses the [create-provisioning-template](#) to create a provisioning template that has pre-provisioning hooks:

```
aws iot create-provisioning template \
--template-name myTemplate \
--provisioning-role-arn arn:aws:iam:us-east-1:1234564789012:role/myRole \
--template-body file://template.json \
--pre-provisioning-hook file://hooks.json
```

The output of this command looks like the following:

```
{  
    "templateArn": "arn:aws:iot:us-east-1:1234564789012:provisioningtemplate/  
myTemplate",  
    "defaultVersionId": 1,  
    "templateName": myTemplate  
}
```

You can also load a parameter from a file instead of typing it all as a command line parameter value to save time. For more information, see [Loading AWS CLI Parameters from a File](#). The following shows the `template` parameter in expanded JSON format:

```
{  
    "Parameters" : {  
        "DeviceLocation": {  
            "Type": "String"  
        }  
    },  
    "Mappings": {  
        "LocationTable": {  
            "Seattle": {  
                "LocationUrl": "https://example.aws"  
            }  
        }  
    },  
    "Resources" : {  
        "thing" : {  
            "Type" : "AWS::IoT::Thing",  
            "Properties" : {  
                "AttributePayload" : {  
                    "version" : "v1",  
                    "serialNumber" : "serialNumber"  
                },  
                "ThingName" : {"Fn::Join": "", ["ThingPrefix_",
{"Ref":"SerialNumber"}]},  
                "ThingTypeName" : {"Fn::Join": "", ["ThingTypePrefix_",
{"Ref":"SerialNumber"}]},  
                "ThingGroups" : ["widgets", "WA"],  
                "BillingGroup": "BillingGroup"  
            },  
            "OverrideSettings" : {  
                "AttributePayload" : "MERGE",  
                "ThingTypeName" : "REPLACE",  
                "ThingGroups" : "DO_NOTHING"  
            }  
        },  
        "certificate" : {  
            "Type" : "AWS::IoT::Certificate",  
            "Properties" : {  
                "CertificateId": {"Ref": "AWS::IoT::Certificate::Id"},  
                "Status" : "Active"  
            }  
        }  
    }  
}
```

```
        }
    },
    "policy" : {
        "Type" : "AWS::IoT::Policy",
        "Properties" : {
            "PolicyDocument" : {
                "Version": "2012-10-17",
                "Statement": [
                    {
                        "Effect": "Allow",
                        "Action": ["iot:Publish"],
                        "Resource": ["arn:aws:iot:us-east-1:504350838278:topic/foo/bar"]
                    }
                ]
            }
        }
    },
    "DeviceConfiguration": {
        "FallbackUrl": "https://www.example.com/test-site",
        "LocationUrl": {
            "Fn::FindInMap": ["LocationTable", {"Ref": "DeviceLocation"}, "LocationUrl"]
        }
    }
}
```

The following shows the `pre-provisioning-hook` parameter in expanded JSON format:

```
{
    "targetArn" : "arn:aws:lambda:us-east-1:765219403047:function:pre_provisioning_test",
    "payloadVersion" : "2020-04-01"
}
```

Provisioning devices that have device certificates

AWS IoT provides three ways to provision devices when they already have a device certificate (and associated private key) on them:

- Single-thing provisioning with a provisioning template. This is a good option if you only need to provision devices one at a time.
- Just-in-time provisioning (JITP) with a template that provisions a device when it first connects to AWS IoT. This is a good option if you need to register large numbers of devices, but you don't have information about them that you can assemble into a bulk provisioning list.
- Bulk registration. This option allows you to specify a list of single-thing provisioning template values that are stored in a file in an S3 bucket. This approach works well if you have a large number of known devices whose desired characteristics you can assemble into a list.

Topics

- [Single thing provisioning \(p. 712\)](#)
- [Just-in-time provisioning \(p. 712\)](#)
- [Bulk registration \(p. 715\)](#)

Single thing provisioning

To provision a thing, use the [RegisterThing](#) API or the `register-thing` CLI command. The `register-thing` CLI command takes the following arguments:

`--template-body`

The provisioning template.

`--parameters`

A list of name-value pairs for the parameters used in the provisioning template, in JSON format (for example, `{"ThingName" : "MyProvisionedThing", "CSR" : "csr-text"}`).

See [Provisioning templates \(p. 716\)](#).

[RegisterThing](#) or `register-thing` returns the ARNs for the resources and the text of the certificate it created:

```
{  
    "certificatePem": "certificate-text",  
    "resourceArns": {  
        "PolicyLogicalName": "arn:aws:iot:us-west-2:123456789012:policy/2A6577675B7CD1823E271C7AAD8184F44630FFD7",  
        "certificate": "arn:aws:iot:us-west-2:123456789012:cert/cd82bb924d4c6ccb14986dcba4f40f30d892cc6b3ce7ad5008ed6542eea2b049",  
        "thing": "arn:aws:iot:us-west-2:123456789012:thing/MyProvisionedThing"  
    }  
}
```

If a parameter is omitted from the dictionary, the default value is used. If no default value is specified, the parameter is not replaced with a value.

Just-in-time provisioning

You can have your devices provisioned when they first attempt to connect to AWS IoT with just-in-time provisioning (JITP). To provision the device, you must enable automatic registration and associate a provisioning template with the CA certificate used to sign the device certificate. Provisioning successes and errors are logged as [Device provisioning metrics \(p. 418\)](#) in Amazon CloudWatch.

You can make these settings when you register a CA certificate with the [RegisterCACertificate](#) API or the `register-ca-certificate` CLI command:

```
aws iot register-ca-certificate --ca-certificate file://your-ca-cert --verification-cert file://your-verification-cert --set-as-active --allow-auto-registration --registration-config file://your-template
```

For more information, see [Registering a CA Certificate](#).

You can also use the [UpdateCACertificate](#) API or the `update-ca-certificate` CLI command to update the settings for a CA certificate:

```
aws iot update-ca-certificate --certificate-id caCertificateId --new-auto-registration-status ENABLE --registration-config file://your-template
```

Note

Just-in-time provisioning JITP calls other AWS IoT control plane APIs during the provisioning process. These calls might exceed the [AWS IoT Throttling Quotas](#) set for your account and result in throttled calls. Contact [AWS Customer Support](#) to raise your throttling quotas, if necessary.

When a device attempts to connect to AWS IoT by using a certificate signed by a registered CA certificate, AWS IoT loads the template from the CA certificate and uses it to call [RegisterThing \(p. 730\)](#). The JITP workflow first registers a certificate with a status value of PENDING_ACTIVATION. When the device provisioning flow is complete, the status of the certificate is changed to ACTIVE.

AWS IoT defines the following parameters that you can declare and reference in provisioning templates:

- AWS::IoT::Certificate::Country
- AWS::IoT::Certificate::Organization
- AWS::IoT::Certificate::OrganizationalUnit
- AWS::IoT::Certificate::DistinguishedNameQualifier
- AWS::IoT::Certificate::StateName
- AWS::IoT::Certificate::CommonName
- AWS::IoT::Certificate::SerialNumber
- AWS::IoT::Certificate::Id

The values for these provisioning template parameters are limited to what JITP can extract from the subject field of the certificate of the device being provisioned. The certificate must contain values for all of the parameters in the template body. The AWS::IoT::Certificate::Id parameter refers to an internally generated ID, not an ID that is contained in the certificate. You can get the value of this ID using the `principal()` function inside an AWS IoT rule.

Note

You can provision devices using AWS IoT Core just-in-time provisioning (JITP) feature without having to send the entire trust chain on devices' first connection to AWS IoT Core. Presenting the CA certificate is optional but the device is required to send the [Server Name Indication \(SNI\)](#) extension when they connect.

The following JSON file is an example of a complete JITP template. The value of the `templateBody` field must be a JSON object specified as an escaped string and can use only the values in the preceding list. You can use a variety of tools to create the required JSON output, such as `json.dumps` (Python) or `JSON.stringify` (Node). The value of the `roleARN` field must be the ARN of a role that has the `AWSIoTThingsRegistration` attached to it. Also, your template can use an existing `PolicyName` instead of the inline `PolicyDocument` in the example. (The first example adds line breaks for readability, but you can copy and paste the template that appears directly below it.)

```
{
    "templateBody" : "{
        \r\n        \"Parameters\" : {\r\n            \r\n                \"AWS::IoT::Certificate::CommonName\" : {\r\n                    \r\n                        \"Type\" : \"String\"\r\n                },\r\n                \r\n                \"AWS::IoT::Certificate::SerialNumber\" : {\r\n                    \r\n                        \"Type\" : \"String\"\r\n                },\r\n                \r\n                \"AWS::IoT::Certificate::Country\" : {\r\n                    \r\n                        \"Type\" : \"String\"\r\n                },\r\n                \r\n                \"AWS::IoT::Certificate::Id\" : {\r\n                    \r\n                        \"Type\" : \"String\"\r\n                }\r\n            },\r\n            \r\n            \"Resources\" : {\r\n                \r\n                    \"thing\" : {\r\n                        \r\n                            \"Type\" : \"AWS::IoT::Thing\", \r\n                            \r\n                            \"Properties\" : {\r\n                                \r\n                                    \"ThingName\" : {\r\n                                        \r\n                                            \"Ref\" : \"AWS::IoT::Certificate::CommonName\"\r\n                                    },\r\n                                    \r\n                                    \"AttributePayload\" : {\r\n                                        \r\n                                            \"version\" : \"v1\", \r\n                                            \r\n                                            \"serialNumber\" : {\r\n                                                \r\n                                                \"Ref\" : \"AWS::IoT::Certificate::SerialNumber\"\r\n                                            }\r\n                                        }\r\n                                    }\r\n                                }\r\n                            }\r\n                        }\r\n                    }\r\n                }\r\n            }\r\n        }\r\n    }\r\n}
```

```

        \"Ref\": \"AWS::IoT::Certificate::SerialNumber
\"ThingTypeArn\": \"lightBulb-versionA\", \r\n
\"ThingGroups\": [\r\n
    \"v1-lightbulbs\", \r\n
    \r\n
        \"Ref\": \"AWS::IoT::Certificate::Country\"\r\n
\r\n } \r\n
    ], \r\n
}, \r\n
\"OverrideSettings\": {\r\n
    \"AttributePayload\": \"MERGE\", \r\n
    \"ThingTypeArn\": \"REPLACE\", \r\n
    \"ThingGroups\": \"DO NOTHING\" \r\n
}, \r\n
\"certificate\": {\r\n
    \"Type\": \"AWS::IoT::Certificate\", \r\n
    \"Properties\": {\r\n
        \"CertificateId\": {\r\n
            \"Ref\": \"AWS::IoT::Certificate::Id\" \r\n
        },
        \"Status\": \"ACTIVE\" \r\n
    },
    \"OverrideSettings\": {\r\n
        \"Status\": \"DO NOTHING\" \r\n
    },
    \"policy\": {\r\n
        \"Type\": \"AWS::IoT::Policy\", \r\n
        \"Properties\": {\r\n
            \"PolicyDocument\": \"{
                \\"Version\\\": \\"2012-10-17\\\",
                \\"Statement\\\": [
                    {
                        \\"Effect\\\": \\"Allow\",
                        \\"Action\\\": [\"iot:Publish\"],
                        \\"Resource\\\": [\"arn:aws:iot:us-east-1:123456789012:topic/sample/topic\"] ]
                }
            } \r\n
        }
    }
},
\"roleArn\" : \"arn:aws:iam::123456789012:role/Provisioning-JITP"
}

```

Here is a version you can copy and paste:

```
{
    "templateBody" : "{\r\n        \"Parameters\" : {\r\n            \"AWS::IoT::Certificate::CommonName\": {\r\n                \"Type\": \"String\"\r\n            },
            \"AWS::IoT::Certificate::SerialNumber\": {\r\n                \"Type\": \"String\"\r\n            },
            \"AWS::IoT::Certificate::Country\": {\r\n                \"Type\": \"String\"\r\n            },
            \"AWS::IoT::Certificate::Id\": {\r\n                \"Type\": \"String\"\r\n            },
            \"Resources\": {\r\n                \"thing\": {\r\n                    \"Type\": \"AWS::IoT::Thing\", \r\n                    \"Properties\": {\r\n                        \"ThingName\": {\r\n                            \"Ref\": \"AWS::IoT::Certificate::CommonName\" \r\n                        },
                        \"AttributePayload\": {\r\n                            \"serialNumber\": {\r\n                                \"Ref\": \"AWS::IoT::Certificate::SerialNumber\" \r\n                            },
                            \"ThingTypeArn\": \"lightBulb-versionA\", \r\n                            \"ThingGroups\": [\r\n                                \"v1-lightbulbs\", \r\n                                \r\n                                    \"Ref\": \"AWS::IoT::Certificate::Country\" \r\n                                ],
                                \r\n                                \"OverrideSettings\": {\r\n                                    \"AttributePayload\": \"MERGE\", \r\n                                    \"ThingTypeArn\": \"REPLACE\", \r\n                                    \"ThingGroups\": \"DO NOTHING\" \r\n                                },
                                \"certificate\": {\r\n                                    \"Type\": \"AWS::IoT::Certificate\", \r\n                                    \"Properties\": {\r\n                                        \"CertificateId\": {\r\n                                            \"Ref\": \"AWS::IoT::Certificate::Id\" \r\n                                        },
                                        \"Status\": \"ACTIVE\" \r\n                                    },
                                    \"OverrideSettings\": {\r\n                                        \"Status\": \"DO NOTHING\" \r\n                                    },
                                    \"policy\": {\r\n                                        \"Type\": \"AWS::IoT::Policy\", \r\n                                        \"Properties\": {\r\n                                            \"PolicyDocument\": \"{
                                                \\"Version\\\": \\"2012-10-17\\\",
                                                \\"Statement\\\": [
                                                    {
                                                        \\"Effect\\\": \\"Allow\",
                                                        \\"Action\\\": [\"iot:Publish\"],
                                                        \\"Resource\\\\": [\"arn:aws:iot:us-east-1:123456789012:topic/sample/topic\"] ]
                                                }
                                            } \r\n
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

```
\\"arn:aws:iot:us-east-1:123456789012:topic\\foo\\bar\\\"] }] }\\r\\n      }\\r\\n    }\\r\\n  },  
  \"roleArn\" : \"arn:aws:iam::123456789012:role/JITPRole\"\n}
```

This sample template declares values for the `AWS::IoT::Certificate::CommonName`, `AWS::IoT::Certificate::SerialNumber`, `AWS::IoT::Certificate::Country`, and `AWS::IoT::Certificate::Id` provisioning parameters that are extracted from the certificate and used in the Resources section. The JITP workflow then uses this template to perform the following actions:

- Register a certificate and set its status to PENDING_ACTIVE.
- Create one thing resource.
- Create one policy resource.
- Attach the policy to the certificate.
- Attach the certificate to the thing.
- Update the certificate status to ACTIVE.

Note that the device provisioning fails if the certificate doesn't have all of the properties mentioned in the `Parameters` section of the `templateBody`. For example, if `AWS::IoT::Certificate::Country` is included in the template, but the certificate doesn't have a `Country` property, the device provisioning fails.

You can also use CloudTrail to troubleshoot issues with your JITP template. For information about the metrics that are logged in Amazon CloudWatch, see [Device provisioning metrics \(p. 418\)](#).

Bulk registration

You can use the [`start-thing-registration-task`](#) command to register things in bulk. This command takes a registration template, an S3 bucket name, a key name, and a role ARN that allows access to the file in the S3 bucket. The file in the S3 bucket contains the values used to replace the parameters in the template. The file must be a newline-delimited JSON file. Each line contains all of the parameter values for registering a single device. For example:

```
{"ThingName": "foo", "SerialNumber": "123", "CSR": "csr1"}  
{"ThingName": "bar", "SerialNumber": "456", "CSR": "csr2"}
```

The following bulk registration-related APIs might be useful:

- [`ListThingRegistrationTasks`](#): Lists the current bulk thing provisioning tasks.
- [`DescribeThingRegistrationTask`](#): Provides information about a specific bulk thing registration task.
- [`StopThingRegistrationTask`](#): Stops a bulk thing registration task.
- [`ListThingRegistrationTaskReports`](#): Used to check the results and failures for a bulk thing registration task.

Note

- Only one bulk registration operation task can run at a time (per account).
- Bulk registration operations call other AWS IoT control plane APIs. These calls might exceed the [AWS IoT Throttling Quotas](#) in your account and cause throttle errors. Contact [AWS Customer Support](#) to raise your AWS IoT throttling quotas, if necessary.

Provisioning templates

A provisioning template is a JSON document that uses parameters to describe the resources your device must use to interact with AWS IoT. A template contains two sections: `Parameters` and `Resources`. There are two types of provisioning templates in AWS IoT. One is used for just-in-time provisioning (JITP) and bulk registration and the second is used for fleet provisioning.

Parameters section

The `Parameters` section declares the parameters used in the `Resources` section. Each parameter declares a name, a type, and an optional default value. The default value is used when the dictionary passed in with the template does not contain a value for the parameter. The `Parameters` section of a template document looks like the following:

```
{  
    "Parameters" : {  
        "ThingName" : {  
            "Type" : "String"  
        },  
        "SerialNumber" : {  
            "Type" : "String"  
        },  
        "Location" : {  
            "Type" : "String",  
            "Default" : "WA"  
        },  
        "CSR" : {  
            "Type" : "String"  
        }  
    }  
}
```

This template snippet declares four parameters: `ThingName`, `SerialNumber`, `Location`, and `CSR`. All of these parameters are of type `String`. The `Location` parameter declares a default value of "WA".

Resources section

The `Resources` section of the template declares the resources required for your device to communicate with AWS IoT: a thing, a certificate, and one or more IoT policies. Each resource specifies a logical name, a type, and a set of properties.

A logical name allows you to refer to a resource elsewhere in the template.

The type specifies the kind of resource you are declaring. Valid types are:

- `AWS::IoT::Thing`
- `AWS::IoT::Certificate`
- `AWS::IoT::Policy`

The properties you specify depend on the type of resource you are declaring.

Thing resources

Thing resources are declared using the following properties:

- `ThingName`: `String`.

- **AttributePayload**: Optional. A list of name-value pairs.
- **ThingTypeName**: Optional. String for an associated thing type for the thing.
- **ThingGroups**: Optional. A list of groups to which the thing belongs.

Certificate resources

You can specify certificates in one of the following ways:

- A certificate signing request (CSR).
- A certificate ID of an existing device certificate. (Only certificate IDs can be used with a fleet provisioning template.)
- A device certificate created with a CA certificate registered with AWS IoT. If you have more than one CA certificate registered with the same subject field, you must also pass in the CA certificate used to sign the device certificate.

Note

When you declare a certificate in a template, use only one of these methods. For example, if you use a CSR, you cannot also specify a certificate ID or a device certificate. For more information, see [X.509 client certificates \(p. 282\)](#).

For more information, see [X.509 Certificate overview \(p. 279\)](#).

Certificate resources are declared using the following properties:

- **CertificateSigningRequest**: String.
- **CertificateID**: String.
- **CertificatePem**: String.
- **CACertificatePem**: String.
- **Status**: Optional. String that can be ACTIVE or INACTIVE. Defaults to ACTIVE.

Examples:

- Certificate specified with a CSR:

```
{  
    "certificate" : {  
        "Type" : "AWS::IoT::Certificate",  
        "Properties" : {  
            "CertificateSigningRequest": {"Ref" : "CSR"},  
            "Status" : "ACTIVE"  
        }  
    }  
}
```

- Certificate specified with an existing certificate ID:

```
{  
    "certificate" : {  
        "Type" : "AWS::IoT::Certificate",  
        "Properties" : {  
            "CertificateId": {"Ref" : "CertificateId"}  
        }  
    }  
}
```

- Certificate specified with an existing certificate .pem and CA certificate .pem:

```
{
    "certificate" : {
        "Type" : "AWS::IoT::Certificate",
        "Properties" : {
            "CACertificatePem": {"Ref" : "CACertificatePem"},
            "CertificatePem": {"Ref" : "CertificatePem"}
        }
    }
}
```

Policy resources

Policy resources are declared using one of the following properties:

- **PolicyName**: Optional. String. Defaults to a hash of the policy document. The **PolicyName** can only reference AWS IoT policies but not IAM policies. If you are using an existing AWS IoT policy, for the **PolicyName** property, enter the name of the policy. Do not include the **PolicyDocument** property.
- **PolicyDocument**: Optional. A JSON object specified as an escaped string. If **PolicyDocument** is not provided, the policy must already be created.

Note

If a **Policy** section is present, **PolicyName** or **PolicyDocument**, but not both, must be specified.

Override settings

If a template specifies a resource that already exists, the **OverrideSettings** section allows you to specify the action to take:

DO NOTHING

Leave the resource as is.

REPLACE

Replace the resource with the resource specified in the template.

FAIL

Cause the request to fail with a **ResourceConflictsException**.

MERGE

Valid only for the **ThingGroups** and **AttributePayload** properties of a thing. Merge the existing attributes or group memberships of the thing with those specified in the template.

When you declare a thing resource, you can specify **OverrideSettings** for the following properties:

- **ATTRIBUTE_PAYLOAD**
- **THING_TYPE_NAME**
- **THING_GROUPS**

When you declare a certificate resource, you can specify **OverrideSettings** for the **Status** property.

OverrideSettings are not available for policy resources.

Resource example

The following template snippet declares a thing, a certificate, and a policy:

```
{
    "Resources" : {
        "thing" : {
            "Type" : "AWS::IoT::Thing",
            "Properties" : {
                "ThingName" : {"Ref" : "ThingName"},  

                "AttributePayload" : { "version" : "v1", "serialNumber" : {"Ref" : "SerialNumber"}},  

                "ThingTypeName" : "lightBulb-versionA",
                "ThingGroups" : ["v1-lightbulbs", {"Ref" : "Location"}]
            },
            "OverrideSettings" : {
                "AttributePayload" : "MERGE",
                "ThingTypeName" : "REPLACE",
                "ThingGroups" : "DO_NOTHING"
            }
        },
        "certificate" : {
            "Type" : "AWS::IoT::Certificate",
            "Properties" : {
                "CertificateSigningRequest": {"Ref" : "CSR"},  

                "Status" : "ACTIVE"
            }
        },
        "policy" : {
            "Type" : "AWS::IoT::Policy",
            "Properties" : {
                "PolicyDocument" : "{ \"Version\": \"2012-10-17\", \"Statement\": [{ \"Effect\": \"Allow\", \"Action\": [\"iot:Publish\"], \"Resource\": [\"arn:aws:iot:us-east-1:123456789012:topic/foo/bar\"] }] }"
            }
        }
    }
}
```

The thing is declared with:

- The logical name "thing".
- The type AWS::IoT::Thing.
- A set of thing properties.

The thing properties include the thing name, a set of attributes, an optional thing type name, and an optional list of thing groups to which the thing belongs.

Parameters are referenced by `{"Ref": "parameter-name"}`. When the template is evaluated, the parameters are replaced with the parameter's value from the dictionary passed in with the template.

The certificate is declared with:

- The logical name "certificate".
- The type AWS::IoT::Certificate.
- A set of properties.

The properties include the CSR for the certificate, and setting the status to ACTIVE. The CSR text is passed as a parameter in the dictionary passed with the template.

The policy is declared with:

- The logical name "policy".
- The type AWS::IoT::Policy.
- Either the name of an existing policy or a policy document.

Template example for JITP and bulk registration

The following JSON file is an example of a complete provisioning template that specifies the certificate with a CSR:

(The PolicyDocument field value must be a JSON object specified as an escaped string.)

```
{
    "Parameters" : {
        "ThingName" : {
            "Type" : "String"
        },
        "SerialNumber" : {
            "Type" : "String"
        },
        "Location" : {
            "Type" : "String",
            "Default" : "WA"
        },
        "CSR" : {
            "Type" : "String"
        }
    },
    "Resources" : {
        "thing" : {
            "Type" : "AWS::IoT::Thing",
            "Properties" : {
                "ThingName" : {"Ref" : "ThingName"},
                "AttributePayload" : { "version" : "v1", "serialNumber" : {"Ref" : "SerialNumber"} },
                "ThingTypeName" : "lightBulb-versionA",
                "ThingGroups" : [ "v1-lightbulbs", {"Ref" : "Location"} ]
            }
        },
        "certificate" : {
            "Type" : "AWS::IoT::Certificate",
            "Properties" : {
                "CertificateSigningRequest": {"Ref" : "CSR"},
                "Status" : "ACTIVE"
            }
        },
        "policy" : {
            "Type" : "AWS::IoT::Policy",
            "Properties" : {
                "PolicyDocument" : "{ \"Version\": \"2012-10-17\", \"Statement\": [{ \\"Effect\\": \"Allow\", \\"Action\\\":[\"iot:Publish\"], \\"Resource\\\": [\"arn:aws:iot:us-east-1:123456789012:topic/foo/bar\"] } ] }"
            }
        }
    }
}
```

The following JSON file is an example of a complete provisioning template that specifies an existing certificate with a certificate ID:

```
{
    "Parameters" : {
        "ThingName" : {
            "Type" : "String"
        },
        "SerialNumber" : {
            "Type" : "String"
        },
        "Location" : {
            "Type" : "String",
            "Default" : "WA"
        },
        "CertificateId" : {
            "Type" : "String"
        }
    },
    "Resources" : {
        "thing" : {
            "Type" : "AWS::IoT::Thing",
            "Properties" : {
                "ThingName" : {"Ref" : "ThingName" },
                "AttributePayload" : { "version" : "v1", "serialNumber" : {"Ref" : "SerialNumber"} },
                "ThingTypeName" : "lightBulb-versionA",
                "ThingGroups" : [ "v1-lightbulbs", {"Ref" : "Location"} ]
            }
        },
        "certificate" : {
            "Type" : "AWS::IoT::Certificate",
            "Properties" : {
                "CertificateId": {"Ref" : "CertificateId"}
            }
        },
        "policy" : {
            "Type" : "AWS::IoT::Policy",
            "Properties" : {
                "PolicyDocument" : "{ \"Version\": \"2012-10-17\", \"Statement\": [{ \"Effect\": \"Allow\", \"Action\":[\"iot:Publish\"], \"Resource\": [\"arn:aws:iot:us-east-1:123456789012:topic/foo/bar\"] }] }"
            }
        }
    }
}
```

Fleet provisioning

Fleet provisioning templates are used by AWS IoT to set up cloud and device configuration. These templates use the same parameters and resources as the JITP and bulk registration templates. For more information, see [Provisioning templates \(p. 716\)](#). Fleet provisioning templates can contain a `Mapping` section and a `DeviceConfiguration` section. You can use intrinsic functions inside a fleet provisioning template to generate device specific configuration. Fleet provisioning templates are named resources and are identified by ARNs (for example, `arn:aws:iot:us-west-2:1234568788:provisioningtemplate/templateName`).

Mappings

The optional `Mappings` section matches a key to a corresponding set of named values. For example, if you want to set values based on an AWS Region, you can create a mapping that uses the AWS Region name as a key and contains the values you want to specify for each specific Region. You use the `Fn::FindInMap` intrinsic function to retrieve values in a map.

You cannot include parameters, pseudo parameters, or call intrinsic functions in the `Mappings` section.

Device configuration

The device configuration section contains arbitrary data you want to send to your devices when provisioning. For example:

```
{  
    "DeviceConfiguration": {  
        "Foo": "Bar"  
    }  
}
```

If you're sending messages to your devices by using the JavaScript Object Notation (JSON) payload format, AWS IoT Core formats this data as JSON. If you're using the Concise Binary Object Representation (CBOR) payload format, AWS IoT Core formats this data as CBOR. The `DeviceConfiguration` section doesn't support nested JSON objects.

Intrinsic functions

Intrinsic functions are used in any section of the provisioning template except the `Mappings` section.

`Fn::Join`

Appends a set of values into a single value, separated by the specified delimiter. If a delimiter is the empty string, the set of values are concatenated with no delimiter.

`Fn::Select`

Returns a single object from a list of objects by index.

Important

`Fn::Select` does not check for `null` values or if the index is out of bounds of the array. Both conditions result in a provisioning error, so you should ensure you chose a valid index value, and that the list contains non-null values.

`Fn::FindInMap`

Returns the value corresponding to keys in a two-level map that is declared in the `Mappings` section.

`Fn::Split`

Splits a string into a list of string values so you can select an element from the list of strings. You specify a delimiter that determine where the string is split (for example, a comma). After you split a string, use `Fn::Select` to select an element.

For example, if a comma-delimited string of subnet IDs is imported to your stack template, you can split the string at each comma. From the list of subnet IDs, use `Fn::Select` to specify a subnet ID for a resource.

`Fn::Sub`

Substitutes variables in an input string with values that you specify. You can use this function to construct commands or outputs that include values that aren't available until you create or update a stack.

Fleet provisioning template example

```
{  
    "Parameters" : {  
        "ThingName" : {
```

```

        "Type" : "String"
    },
    "SerialNumber": {
        "Type": "String"
    },
    "DeviceLocation": {
        "Type": "String"
    }
}
},
"Mappings": {
    "LocationTable": {
        "Seattle": {
            "LocationUrl": "https://example.aws"
        }
    }
},
"Resources" : {
    "thing" : {
        "Type" : "AWS::IoT::Thing",
        "Properties" : {
            "AttributePayload" : {
                "version" : "v1",
                "serialNumber" : "serialNumber"
            },
            "ThingName" : {"Ref" : "ThingName"},
            "ThingTypeName" : {"Fn::Join": ["", ["ThingPrefix_",
{"Ref":"SerialNumber"}]]},
            "ThingGroups" : ["v1-lightbulbs", "WA"],
            "BillingGroup": "LightBulbBillingGroup"
        },
        "OverrideSettings" : {
            "AttributePayload" : "MERGE",
            "ThingTypeName" : "REPLACE",
            "ThingGroups" : "DO_NOTHING"
        }
    },
    "certificate" : {
        "Type" : "AWS::IoT::Certificate",
        "Properties" : {
            "CertificateId": {"Ref": "AWS::IoT::Certificate::Id"},
            "Status" : "Active"
        }
    },
    "policy" : {
        "Type" : "AWS::IoT::Policy",
        "Properties" : {
            "PolicyDocument" : {
                "Version": "2012-10-17",
                "Statement": [
                    {
                        "Effect": "Allow",
                        "Action": ["iot:Publish"],
                        "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/foo/bar"]
                    }
                ]
            }
        }
    },
    "DeviceConfiguration": {
        "FallbackUrl": "https://www.example.com/test-site",
        "LocationUrl": {
            "Fn::FindInMap": ["LocationTable", {"Ref": "DeviceLocation"}, "LocationUrl"]
        }
    }
}
}

```

Note

An existing provisioning template can be updated to add a [pre-provisioning hook \(p. 724\)](#).

Pre-provisioning hooks

When using AWS IoT fleet provisioning, you can set up a Lambda function to validate parameters passed from the device before allowing the device to be provisioned. This Lambda function must exist in your account before you provision a device because it's called every time a device sends a request through [the section called "RegisterThing" \(p. 730\)](#). For devices to be provisioned, your Lambda function must accept the input object and return the output object described in this section. The provisioning proceeds only if the Lambda function returns an object with "allowProvisioning": True.

Important

AWS recommends using pre-provisioning hooks when creating provisioning templates to allow more control of which and how many devices your account onboard.

Pre-provision hook input

AWS IoT sends this object to the Lambda function when a device registers with AWS IoT.

```
{  
    "claimCertificateId" : "string",  
    "certificateId" : "string",  
    "certificatePem" : "string",  
    "templateArn" : "arn:aws:iot:us-east-1:1234567890:provisioningtemplate/MyTemplate",  
    "clientId" : "221a6d10-9c7f-42f1-9153-e52e6fc869c1",  
    "parameters" : {  
        "string" : "string",  
        ...  
    }  
}
```

The `parameters` object passed to the Lambda function contains the properties in the `parameters` argument passed in the [the section called "RegisterThing" \(p. 730\)](#) request payload.

Pre-provision hook return value

The Lambda function must return a response that indicates whether it has authorized the provisioning request and the values of any properties to override.

The following is an example of a successful response from the pre-provisioning function.

```
{  
    "allowProvisioning": true,  
    "parameterOverrides" : {  
        "Key": "newCustomValue",  
        ...  
    }  
}
```

"`parameterOverrides`" values will be added to "`parameters`" parameter of the [the section called "RegisterThing" \(p. 730\)](#) request payload.

Note

- If the Lambda function fails or doesn't return the "allowProvisioning" parameter in the response, the provisioning request will fail and the error will be returned in the response.

- The Lambda function must finish running and return within 5 seconds, otherwise the provisioning request fails.

Pre-provisioning hook Lambda example

Python

An example of a pre-provisioning hook Lambda in Python.

```
import json

def pre_provisioning_hook(event, context):
    print(event)

    return {
        'allowProvisioning': True,
        'parameterOverrides': {
            'DeviceLocation': 'Seattle'
        }
    }
```

Java

An example of a pre-provisioning hook Lambda in Java.

Handler class:

```
package example;

import java.util.Map;
import java.util.HashMap;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

public class PreProvisioningHook implements RequestHandler<PreProvisioningHookRequest,
    PreProvisioningHookResponse> {

    public PreProvisioningHookResponse handleRequest(PreProvisioningHookRequest object,
    Context context) {
        Map<String, String> parameterOverrides = new HashMap<String, String>();
        parameterOverrides.put("DeviceLocation", "Seattle");

        PreProvisioningHookResponse response = PreProvisioningHookResponse.builder()
            .allowProvisioning(true)
            .parameterOverrides(parameterOverrides)
            .build();

        return response;
    }
}
```

Request class:

```
package example;

import java.util.Map;
import lombok.Builder;
import lombok.Data;
import lombok.AllArgsConstructor;
import lombok.NoArgsConstructor;
```

```

@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class PreProvisioningHookRequest {
    private String claimCertificateId;
    private String certificateId;
    private String certificatePem;
    private String templateArn;
    private String clientId;
    private Map<String, String> parameters;
}

```

Response class:

```

package example;

import java.util.Map;
import lombok.Builder;
import lombok.Data;
import lombok.AllArgsConstructor;
import lombok.NoArgsConstructor;

@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class PreProvisioningHookResponse {
    private boolean allowProvisioning;
    private Map<String, String> parameterOverrides;
}

```

Device provisioning MQTT API

The Fleet Provisioning service supports these MQTT APIs:

- the section called “[CreateCertificateFromCsr](#)” (p. 727)
- the section called “[CreateKeysAndCertificate](#)” (p. 728)
- the section called “[RegisterThing](#)” (p. 730)

This API supports response buffers in Concise Binary Object Representation (CBOR) format and JavaScript Object Notation (JSON), depending on the [payload-format](#) of the topic. For the sake of clarity, however, the response and request examples in this section are shown in JSON format.

payload-format	Response format data type
cbor	Concise Binary Object Representation (CBOR)
json	JavaScript Object Notation (JSON)

Important

Before publishing a request message topic, subscribe to the response topics to receive the response. The messages used by this API use MQTT's publish/subscribe protocol to provide a request and response interaction.

If you do not subscribe to the response topics *before* you publish a request, you might not receive the results of that request.

CreateCertificateFromCsr

Creates a certificate from a certificate signing request (CSR). AWS IoT provides client certificates that are signed by the Amazon Root certificate authority (CA). The new certificate has a PENDING_ACTIVATION status. When you call `RegisterThing` to provision a thing with this certificate, the certificate status changes to ACTIVE or INACTIVE as described in the template.

Note

For security, the `certificateOwnershipToken` returned by [CreateCertificateFromCsr \(p. 727\)](#) expires after one hour. [RegisterThing \(p. 730\)](#) must be called before the `certificateOwnershipToken` expires. If the token expires, the device can call [CreateCertificateFromCsr \(p. 727\)](#) to generate a new certificate.

CreateCertificateFromCsr request

Publish a message with the `$aws/certificates/create-from-csr/payload-format` topic.

payload-format

The message payload format as cbor or json.

CreateCertificateFromCsr request payload

```
{  
    "certificateSigningRequest": "string"  
}
```

`certificateSigningRequest`

The CSR, in PEM format.

CreateCertificateFromCsr response

Subscribe to `$aws/certificates/create-from-csr/payload-format/accepted`.

payload-format

The message payload format as cbor or json.

CreateCertificateFromCsr response payload

```
{  
    "certificateOwnershipToken": "string",  
    "certificateId": "string",  
    "certificatePem": "string"  
}
```

`certificateOwnershipToken`

The token to prove ownership of the certificate during provisioning.

`certificateId`

The ID of the certificate. Certificate management operations only take a certificateId.

`certificatePem`

The certificate data, in PEM format.

CreateCertificateFromCsr error

To receive error responses, subscribe to `$aws/certificates/create-from-csr/payload-format/rejected`.

`payload-format`

The message payload format as cbor or json.

CreateCertificateFromCsr error payload

```
{  
    "statusCode": int,  
    "errorCode": "string",  
    "errorMessage": "string"  
}
```

`statusCode`

The status code.

`errorCode`

The error code.

`errorMessage`

The error message.

CreateKeysAndCertificate

Creates new keys and a certificate. AWS IoT provides client certificates that are signed by the Amazon Root certificate authority (CA). The new certificate has a PENDING_ACTIVATION status. When you call `RegisterThing` to provision a thing with this certificate, the certificate status changes to ACTIVE or INACTIVE as described in the template.

Note

For security, the `certificateOwnershipToken` returned by

[CreateKeysAndCertificate \(p. 728\)](#) expires after one hour. [RegisterThing \(p. 730\)](#) must be called before the `certificateOwnershipToken` expires. If the token expires, the device can call [CreateKeysAndCertificate \(p. 728\)](#) to generate a new certificate.

CreateKeysAndCertificate request

Publish a message on `$aws/certificates/create/payload-format` with an empty message payload.

`payload-format`

The message payload format as cbor or json.

CreateKeysAndCertificate response

Subscribe to `$aws/certificates/create/payload-format/accepted`.

payload-format

The message payload format as cbor or json.

CreateKeysAndCertificate response

```
{  
    "certificateId": "string",  
    "certificatePem": "string",  
    "privateKey": "string",  
    "certificateOwnershipToken": "string"  
}
```

certificateId

The certificate ID.

certificatePem

The certificate data, in PEM format.

privateKey

The private key.

certificateOwnershipToken

The token to prove ownership of the certificate during provisioning.

CreateKeysAndCertificate error

To receive error responses, subscribe to `$aws/certificates/create/payload-format/rejected`.

payload-format

The message payload format as cbor or json.

CreateKeysAndCertificate error payload

```
{  
    "statusCode": int,  
    "errorCode": "string",  
    "errorMessage": "string"  
}
```

statusCode

The status code.

errorCode

The error code.

errorMessage

The error message.

RegisterThing

Provisions a thing using a pre-defined template.

RegisterThing request

Publish a message on `$aws/provisioning-templates/templateName/provision/payload-format`.

payload-format

The message payload format as cbor or json.

templateName

The provisioning template name.

RegisterThing request payload

```
{  
    "certificateOwnershipToken": "string",  
    "parameters": {  
        "string": "string",  
        ...  
    }  
}
```

certificateOwnershipToken

The token to prove ownership of the certificate. The token is generated by AWS IoT when you create a certificate over MQTT.

parameters

Optional. Key-value pairs from the device that are used by the [pre-provisioning hooks \(p. 724\)](#) to evaluate the registration request.

RegisterThing response

Subscribe to `$aws/provisioning-templates/templateName/provision/payload-format/accepted`.

payload-format

The message payload format as cbor or json.

templateName

The provisioning template name.

RegisterThing response payload

```
{  
    "deviceConfiguration": {  
        "string": "string",  
        ...  
    },  
}
```

```
    "thingName": "string"  
}
```

deviceConfiguration

The device configuration defined in the template.

thingName

The name of the IoT thing created during provisioning.

RegisterThing error response

To receive error responses, subscribe to `$aws/provisioning-templates/templateName/provision/payload-format/rejected`.

payload-format

The message payload format as cbor or json.

templateName

The provisioning template name.

RegisterThing error response payload

```
{  
    "statusCode": int,  
    "errorCode": "string",  
    "errorMessage": "string"  
}
```

statusCode

The status code.

errorCode

The error code.

errorMessage

The error message.

Fleet indexing service

Fleet Indexing is a managed service that you can use to index, search, and aggregate your registry data, shadow data, and device connectivity data (device lifecycle events) in the cloud. After you set up your fleet index, the service manages the indexing of updates for your thing groups, thing registries, and device shadows. For more information about aggregation queries, see [Querying for aggregate data \(p. 744\)](#). You can use a simple query language to search across this data. You can also create a [dynamic thing group \(p. 267\)](#) with a search query.

Note

It might take about 30 seconds for the Fleet Indexing service to update the fleet index after a thing is created, updated, or deleted.

When you enable indexing, AWS IoT creates an index for your things or thing groups. After it's active, you can run queries on your index, such as finding all devices that are handheld and have more than 70 percent battery life. AWS IoT keeps the index continuously updated with your latest data.

`AWS_Things` is the index created for all of your things. `AWS_ThingGroups` is the index that contains all of your thing groups.

You can use the [AWS IoT console](#) to manage your indexing configuration and run your search queries. Choose the indexes you would like to use in the console settings page. If you prefer programmatic access, you can use the AWS SDKs or the AWS Command Line Interface (AWS CLI).

For information about pricing this and other services, see the [AWS IoT Device Management Pricing](#) page.

Topics

- [Managing thing indexing \(p. 732\)](#)
- [Managing thing group indexing \(p. 742\)](#)
- [Querying for aggregate data \(p. 744\)](#)
- [Query syntax \(p. 749\)](#)
- [Example thing queries \(p. 750\)](#)
- [Example thing group queries \(p. 752\)](#)
- [Fleet metrics \(p. 753\)](#)

Managing thing indexing

`AWS_Things` is the index created for all of your things. You can control what to index: registry data, shadow data, and device connectivity status data (driven by device lifecycle events).

Enabling thing indexing

You use the `update-indexing-configuration` CLI command or the `UpdateIndexingConfiguration` API to create the `AWS_Things` index and control its configuration. The `--thing-indexing-configuration` (`thingIndexingConfiguration`) parameter allows you to control what kind of data (for example, registry, shadow, and device connectivity data) is indexed.

The `--thing-indexing-configuration` parameter takes a string with the following structure:

```
{  
    "thingIndexingMode": "OFF" | "REGISTRY" | "REGISTRY_AND_SHADOW",  
    "thingConnectivityIndexingMode": "OFF" | "STATUS",  
    "customFields": [  
        { name: field-name, type: String | Number | Boolean },  
        ...  
    ]  
}
```

The `thingIndexingMode` attribute controls what kind of data is indexed. Valid values are:

OFF

No indexing.

REGISTRY

Index registry data.

REGISTRY_AND_SHADOW

Index registry and thing shadow data.

The `thingConnectivityIndexingMode` attribute specifies if thing connectivity data is indexed. Valid values are:

OFF

Thing connectivity data is not indexed.

STATUS

Thing connectivity data is indexed.

The `customFields` attribute is a list of field and data type pairs. Aggregation queries can be performed over these fields based on the data type. The indexing mode you choose (REGISTRY or REGISTRY_AND_SHADOW) affects what fields can be specified in `customFields`. For example, if you specify the REGISTRY indexing mode, you cannot specify a field from a thing shadow. Custom fields must be specified in `customFields` to be indexed.

If there is a type inconsistency between a custom field in your configuration and the value being indexed, the Fleet Indexing service ignores the inconsistent value for aggregation queries. CloudWatch logs are helpful when troubleshooting aggregation query problems. For more information, see [Troubleshooting aggregation queries for the fleet indexing service \(p. 1137\)](#).

Managed fields contain data associated with IoT things, thing groups, and device shadows. The data type of managed fields are defined by AWS IoT. You specify the values of each managed field when you create an IoT thing. For example thing names, thing groups, and thing descriptions are all managed fields. The Fleet Indexing service indexes managed fields based on the indexing mode you specify:

- Managed fields for the registry

```
"managedFields" : [  
    {name:thingId, type:String},  
    {name:thingName, type:String},  
    {name:registry.version, type:Number},  
    {name:registry(thingType)Name, type:String},  
    {name:registry(thingGroupNames, type:String},  
]
```

- Managed fields for thing shadows

```
"managedFields" : [
    {name:shadow.version, type:Number},
    {name:shadow.hasDelta, type:Boolean}
]
```

- Managed fields for thing connectivity

```
"managedFields" : [
    {name:connectivity.timestamp, type:Number},
    {name:connectivity.version, type:Number},
    {name:connectivity.connected, type:Boolean},
    {name:connectivity.disconnectReason, type:String}
]
```

- Managed fields for thing groups

```
"managedFields" : [
    {name:description, type:String},
    {name:parentGroupNames, type:String},
    {name:thingGroupId, type:String},
    {name:thingGroupName, type:String},
    {name:version, type:Number}
]
```

Managed fields cannot be changed or appear in customFields.

The following is an example of how to use **update-indexing-configuration** to configure indexing:

```
aws iot update-indexing-configuration --thing-indexing-configuration
    'thingIndexingMode=REGISTRY_AND_SHADOW,customFields=[{name=attributes.version,type=Number},
    {name=attributes.color,
        type=String},{name=shadow.desired.power, type[Boolean]}]'
```

This command enables indexing for registry and shadow data. Aggregation queries work with the managed fields and the provided customFields based on the data type.

You can use the **get-indexing-configuration** CLI command or the [GetIndexingConfiguration](#) API to retrieve the current indexing configuration.

The following command shows how to use the **get-indexing-configuration** CLI command to retrieve the current thing indexing configuration where five custom fields (three registry custom fields and two shadow custom fields) are defined.

```
aws iot get-indexing-configuration
```

```
{
    "thingGroupIndexingConfiguration": {
        "thingGroupIndexingMode": "OFF"
    },
    "thingIndexingConfiguration": {
        "thingConnectivityIndexingMode": "STATUS",
        "customFields": [
            {
                "name": "attributes.customField_NUM",
                "type": "Number"
            }
        ]
    }
}
```

```

        },
        {
            "name": "shadow.desired.customField_STR",
            "type": "String"
        },
        {
            "name": "shadow.desired.customField_NUM",
            "type": "Number"
        },
        {
            "name": "attributes.customField_STR",
            "type": "String"
        },
        {
            "name": "attributes.customField_BOOL",
            "type": "Boolean"
        }
    ],
    "thingIndexingMode": "REGISTRY_AND_SHADOW",
    "managedFields": [
        {
            "name": "shadow.hasDelta",
            "type": "Boolean"
        },
        {
            "name": "registry.thingGroupNames",
            "type": "String"
        },
        {
            "name": "connectivity.version",
            "type": "Number"
        },
        {
            "name": "connectivity.disconnectReason",
            "type": "String"
        },
        {
            "name": "registry.thingTypeName",
            "type": "String"
        },
        {
            "name": "connectivity.connected",
            "type": "Boolean"
        },
        {
            "name": "registry.version",
            "type": "Number"
        },
        {
            "name": "thingId",
            "type": "String"
        },
        {
            "name": "connectivity.timestamp",
            "type": "Number"
        },
        {
            "name": "thingName",
            "type": "String"
        },
        {
            "name": "shadow.version",
            "type": "Number"
        }
    ]
}

```

}

The following table provides the allowed combinations of `thingIndexingMode` and `thingConnectivityIndexingMode`, and their associated effects. The required `thingIndexingMode` parameter specifies if the AWS_Things index contains just registry data or registry and shadow data. The optional `thingConnectivityIndexingMode` parameter specifies whether the index also contains connectivity status data (when devices have last connected and disconnected to AWS IoT).

<code>thingIndexingMode</code>	<code>thingConnectivityIndexingMode</code>	Result
OFF	<i>Not specified.</i>	No indexing or delete an index.
OFF	OFF	Equivalent to the previous entry.
REGISTRY	<i>Not specified.</i>	Create or configure the AWS_Things index to index registry data only.
REGISTRY	OFF	Equivalent to the previous entry. (Only registry data is indexed.)
REGISTRY_AND_SHADOW	<i>Not specified.</i>	Create or configure the AWS_Things index to index registry data and shadow data.
REGISTRY_AND_SHADOW	OFF	Equivalent to the previous entry. (Registry data and shadow data are indexed.)
REGISTRY	STATUS	Create or configure the AWS_Things index to index registry data and thing connectivity status data (REGISTRY_AND_CONNECTIVITY_STATUS).
REGISTRY_AND_SHADOW	STATUS	Create or configure the AWS_Things index to index registry data, shadow data, and thing connectivity status data (REGISTRY_AND_SHADOW_AND_CONNECTIVITY_STATUS).

You can use the AWS IoT **update-indexing-configuration** CLI command to update your indexing configuration. The following examples show how to use the **update-indexing-configuration** CLI command.

Short syntax:

```
aws iot update-indexing-configuration --thing-indexing-configuration
  "thingIndexingMode=REGISTRY_AND_SHADOW,thingConnectivityIndexingMode=STATUS,customFields=[{name=attrib
  {name=attributes.color,type=String},{name=shadow.desired.power,type=Boolean}]"
```

JSON syntax:

```
aws iot update-indexing-configuration --cli-input-json \ '{
  "thingIndexingConfiguration": { "thingIndexingMode": "REGISTRY_AND_SHADOW",
```

```
"thingConnectivityIndexingMode": "STATUS", "customFields": [ { "name": "shadow.desired.power", "type": "Boolean" }, { "name": "attributes.color", "type": "String" }, { "name": "attributes.version", "type": "Number" } ] } }
```

The output of these commands is:

```
{
    "thingIndexingConfiguration": {
        "thingConnectivityIndexingMode": "STATUS",
        "customFields": [
            {
                "type": "String",
                "name": "attributes.color"
            },
            {
                "type": "Number",
                "name": "attributes.version"
            },
            {
                "type": "Boolean",
                "name": "shadow.desired.power"
            }
        ],
        "thingIndexingMode": "REGISTRY_AND_SHADOW",
        "managedFields": [
            {
                "type": "Boolean",
                "name": "connectivity.connected"
            },
            {
                "type": "String",
                "name": "registry.thingTypeName"
            },
            {
                "type": "String",
                "name": "thingName"
            },
            {
                "type": "Number",
                "name": "shadow.version"
            },
            {
                "type": "String",
                "name": "thingId"
            },
            {
                "type": "Boolean",
                "name": "shadow.hasDelta"
            },
            {
                "type": "Number",
                "name": "connectivity.timestamp"
            },
            {
                "type": "String",
                "name": "connectivity.disconnectReason"
            },
            {
                "type": "String",
                "name": "registry.thingGroupNames"
            },
            {
                "type": "Number",
                "name": "connectivity.lastReconnectTimestamp"
            }
        ]
    }
}
```

```

        "name": "connectivity.version"
    },
{
    "type": "Number",
    "name": "registry.version"
}
],
"thingGroupIndexingConfiguration": {
    "thingGroupIndexingMode": "OFF"
}
}
}

```

In the following example, a new custom field is added to the configuration:

```

aws iot update-indexing-configuration --thing-indexing-configuration

'thingIndexingMode=REGISTRY_AND_SHADOW,customFields=[{name=attributes.version,type=Number},
{name=attributes.color,type=String},{name=shadow.desired.power,type=Boolean},
{name=shadow.desired.intensity,type=Number}]'

```

This command added `shadow.desired.intensity` to the indexing configuration.

Note

Updating the custom fields indexing configuration overwrites all existing custom fields. Make sure to specify all custom fields when calling `update-indexing-configuration`.

After the index is rebuilt you can, use aggregation query on the newly added fields, search registry data, shadow data, and thing connectivity status data.

When changing the indexing mode, make sure all of your custom fields are valid using the new indexing mode. For example, if you start off with `REGISTRY_AND_SHADOW` mode with a custom field called `shadow.desired.temperature` you must delete the `shadow.desired.temperature` custom field before changing the indexing mode to `REGISTRY`. If your indexing configuration contains custom fields that are not indexed by the indexing mode, the update fails.

Describing a thing index

The following command shows you how to use the `describe-index` CLI command to retrieve the current status of the thing index.

```

aws iot describe-index --index-name "AWS_Things"
{
    "indexName": "AWS_Things",
    "indexStatus": "BUILDING",
    "schema": "REGISTRY_AND_SHADOW_AND_CONNECTIVITY_STATUS"
}

```

The first time you enable indexing, AWS IoT builds your index. You can't query the index if `indexStatus` is in the `BUILDING` state. The schema for the things index indicates which type of data (`REGISTRY_AND_SHADOW_AND_CONNECTIVITY_STATUS`) is indexed.

Changing the configuration of your index causes the index to be rebuilt. During this process, the `indexStatus` is `REBUILDING`. You can execute queries on data in the things index while it is being rebuilt. For example, if you change the index configuration from `REGISTRY` to `REGISTRY_AND_SHADOW` while the index is being rebuilt, you can query registry data, including the latest updates. However, you can't query the shadow data until the rebuild is complete. The amount of time it takes to build or rebuild the index depends on the amount of data.

Querying a thing index

Use the **search-index** CLI command to query data in the index.

```
aws iot search-index --index-name "AWS_Things" --query-string  
    "thingName:mything*"
```

```
{  
    "things": [ {  
        "thingName": "mything1",  
        "thingGroupNames": [  
            "mygroup1"  
        ],  
        "thingId": "a4b9f759-b0f2-4857-8a4b-967745ed9f4e",  
        "attributes": {  
            "attribute1": "abc"  
        },  
        "connectivity": {  
            "connected": false,  
            "timestamp": 1556649874716,  
            "disconnectReason": "CONNECTION_LOST"  
        }  
    },  
    {  
        "thingName": "mything2",  
        "thingTypeName": "MyThingType",  
        "thingGroupNames": [  
            "mygroup1",  
            "mygroup2"  
        ],  
        "thingId": "01014ef9-e97e-44c6-985a-d0b06924f2af",  
        "attributes": {  
            "model": "1.2",  
            "country": "usa"  
        },  
        "shadow": {  
            "desired": {  
                "location": "new york",  
                "myvalues": [3, 4, 5]  
            },  
            "reported": {  
                "location": "new york",  
                "myvalues": [1, 2, 3],  
                "stats": {  
                    "battery": 78  
                }  
            },  
            "metadata": {  
                "desired": {  
                    "location": {  
                        "timestamp": 123456789  
                    },  
                    "myvalues": {  
                        "timestamp": 123456789  
                    }  
                },  
                "reported": {  
                    "location": {  
                        "timestamp": 34535454  
                    },  
                    "myvalues": {  
                        "timestamp": 34535454  
                    }  
                }  
            }  
        }  
    }  
}
```

```
        },
        "stats": {
            "battery": {
                "timestamp": 34535454
            }
        }
    },
    "version": 10,
    "timestamp": 34535454
},
"connectivity": {
    "connected": true,
    "timestamp": 1556649855046
}
},
"nextToken": "AQFCuvk7zZ3D9pOYMbFCeHbdZ+h=G"
}
```

In the JSON response, "connectivity" (as enabled by the `thingConnectivityIndexingMode=STATUS` setting) provides a Boolean value, a timestamp, and a `disconnectReason` that indicates if the device is connected to AWS IoT Core. The device "mything1" disconnected (`false`) at POSIX time 1556649874716 due to `CONNECTION_LOST`. For more information about disconnect reasons, read [Lifecycle events \(p. 993\)](#).

```
"connectivity": {
    "connected": false,
    "timestamp": 1556649874716,
    "disconnectReason": "CONNECTION_LOST"
}
```

The device "mything2" connected (`true`) at POSIX time 1556649855046:

```
"connectivity": {
    "connected": true,
    "timestamp": 1556649855046
}
```

Timestamps are given in milliseconds since epoch, so 1556649855046 represents 6:44:15.046 PM on Tuesday, April 30, 2019 (UTC).

Important

If a device has been disconnected for approximately an hour, the "timestamp" value and the "disconnectReason" value of the connectivity status might be missing.

Restrictions and limitations

These are the restrictions and limitations for AWS_Things.

Shadow fields with complex types

A shadow field is indexed only if the value of the field is a simple type, a JSON object that does not contain an array, or an array that consists entirely of simple types. Simple type means a string, number, or one of the literals `true` or `false`. For example, given the following shadow state, the value of field "palette" is not indexed because it's an array that contains items of complex types. The value of field "colors" is indexed because each value in the array is a string.

```
{
    "state": {
```

```

    "reported": {
        "switched": "ON",
        "colors": [ "RED", "GREEN", "BLUE" ],
        "palette": [
            {
                "name": "RED",
                "intensity": 124
            },
            {
                "name": "GREEN",
                "intensity": 68
            },
            {
                "name": "BLUE",
                "intensity": 201
            }
        ]
    }
}

```

Nested shadow field names

The names of nested shadow fields are stored as a period (.) delimited string. For example, given a shadow document:

```
{
    "state": {
        "desired": {
            "one": {
                "two": {
                    "three": "v2"
                }
            }
        }
    }
}
```

the name of field `three` is stored as `desired.one.two.three`. If you also have a shadow document like this:

```
{
    "state": {
        "desired": {
            "one.two.three": "v2"
        }
    }
}
```

both match a query for `shadow.desired.one.two.three:v2`. As a best practice, do not use periods in shadow field names.

Shadow metadata

A field in a shadow's metadata section is indexed, but only if the corresponding field in the shadow's "state" section is indexed. (In the previous example, the "palette" field in the shadow's metadata section is not indexed either.)

Unregistered shadows

If you use [UpdateThingShadow](#) to create a shadow using a thing name that hasn't been registered in your AWS IoT account, fields in this shadow are not indexed.

Numeric values

If any registry or shadow data is recognized by the service as a numeric value, it's indexed as such. You can form queries involving ranges and comparison operators on numeric values (for example, "attribute.foo<5" or "shadow.reported.foo:[75 TO 80]"). To be recognized as numeric, the value of the data must be a valid JSON number type literal (an integer in the range -2⁵³...2⁵³-1 or a double-precision floating point with optional exponential notation) or part of an array that contains only such values.

Null values

Null values are not indexed.

Maximum number of custom fields for aggregation queries

5

Maximum number of requested percentiles for aggregation queries.

100

Authorization

You can specify the things index as a resource ARN in an AWS IoT policy action, as follows.

Action	Resource
iot:SearchIndex	An index ARN (for example, arn:aws:iot: <i>your-aws-region</i> : <i>your-aws-account</i> :index/AWS_Things).
iot:DescribeIndex	An index ARN (for example, arn:aws:iot: <i>your-aws-region</i> :index/AWS_Things).

Note

If you have permissions to query the fleet index, you can access the data of things across the entire fleet.

Managing thing group indexing

AWS_ThingGroups is the index that contains all of your thing groups. You can use this index to search for groups based on group name, description, attributes, and all parent group names.

Enabling thing group indexing

You can use the `thing-group-indexing-configuration` setting in the [UpdateIndexingConfiguration](#) API to create the AWS_ThingGroups index and control its configuration. You can use the [GetIndexingConfiguration](#) API to retrieve the current indexing configuration.

Use the `get-indexing-configuration` CLI command to retrieve the current thing and thing group indexing configurations.

```
aws iot get-indexing-configuration
```

```
{  
    "thingGroupIndexingConfiguration": {  
        "thingGroupIndexingMode": "ON"  
    }  
}
```

Use the **update-indexing-configuration** CLI command to update the thing group indexing configurations.

```
aws iot update-indexing-configuration --thing-group-indexing-configuration  
    thingGroupIndexingMode=ON
```

Note

You can also update configurations for both thing and thing group indexing in a single command, as follows.

```
aws iot update-indexing-configuration --thing-indexing-configuration  
    thingIndexingMode=REGISTRY --thing-group-indexing-configuration  
    thingGroupIndexingMode=ON
```

The following are valid values for `thingGroupIndexingMode`.

OFF

No indexing/delete index.

ON

Create or configure the `AWS_ThingGroups` index.

Describing group indexes

Use the **describe-index** CLI command to retrieve the current status of the `AWS_ThingGroups` index.

```
aws iot describe-index --index-name "AWS_ThingGroups"  
{  
    "indexStatus": "ACTIVE",  
    "indexName": "AWS_ThingGroups",  
    "schema": "THING_GROUPS"  
}
```

AWS IoT builds your index the first time you enable indexing. You can't query the index if the `indexStatus` is `BUILDING`.

Querying a thing group index

Use the **search-index** CLI command to query data in the index:

```
aws iot search-index --index-name "AWS_ThingGroups" --query-string  
    "thingGroupName:mythinggroup*"
```

Authorization

You can specify the thing groups index as a resource ARN in an AWS IoT policy action, as follows.

Action	Resource
<code>iot:SearchIndex</code>	An index ARN (for example, <code>arn:aws:iot:<i>your-aws-region</i>:index/AWS_ThingGroups</code>).
<code>iot:DescribeIndex</code>	An index ARN (for example, <code>arn:aws:iot:<i>your-aws-region</i>:index/AWS_ThingGroups</code>).

Querying for aggregate data

AWS IoT provides four APIs (`GetStatistics`, `GetCardinality`, `GetPercentiles`, and `GetBucketsAggregation`) that allow you to search your device fleet for aggregate data.

Note

For issues with missing or unexpected values for the aggregation APIs, read [Fleet indexing troubleshooting guide \(p. 1137\)](#).

GetStatistics

The `GetStatistics` API and the `get-statistics` CLI command return the count, average, sum, minimum, maximum, sum of squares, variance, and standard deviation for the specified aggregated field.

The `get-statistics` CLI command takes the following parameters:

`index-name`

The name of the index to search. The default value is `AWS_Things`.

`query-string`

The query used to search the index. You can specify "*" to get the count of all indexed things in your AWS account.

`aggregationField`

Optional. The field to aggregate. This field must be a managed or custom field defined when you call `update-indexing-configuration`. If you don't specify an aggregation field, `registry.version` is used as aggregation field.

`query-version`

The version of the query to use. The default value is `2017-09-30`.

The type of aggregation field can affect the statistics returned.

GetStatistics with string values

If you aggregate on a string field, calling `GetStatistics` returns a count of devices that have attributes that match the query. For example:

```
aws iot get-statistics --aggregation-field 'attributes.stringAttribute'
--query-string '*'
```

This command returns the number of devices that contain an attribute named `stringAttribute`:

```
{
  "statistics": {
    "count": 3
```

```
}
```

GetStatistics with Boolean values

When you call `GetStatistics` with a boolean aggregation field:

- `AVERAGE` is the percentage of devices that match the query.
- `MINIMUM` is 0 or 1 according to the following rules:
 - If all the values for the aggregation field are `false`, `MINIMUM` is 0.
 - If all the values for the aggregation field are `true`, `MINIMUM` is 1.
 - If the values for the aggregation field are a mixture of `false` and `true`, `MINIMUM` is 0.
- `MAXIMUM` is 0 or 1 according to the following rules:
 - If all the values for the aggregation field are `false`, `MAXIMUM` is 0.
 - If all the values for the aggregation field are `true`, `MAXIMUM` is 1.
 - If the values for the aggregation field are a mixture of `false` and `true`, `MAXIMUM` is 1.
- `SUM` is the sum of the integer equivalent of the boolean values.
- `COUNT` is the count of things that match the query string criteria and contain a valid aggregation field value.

GetStatistics with numerical values

When you call `GetStatistics` and specify an aggregation field of type `Number`, `GetStatistics` returns the following values:

`count`

The count of things that match the query string criteria and contain a valid aggregation field value.
`average`

The average of the numerical values that match the query.

`sum`

The sum of the numerical values that match the query.

`minimum`

The smallest of the numerical values that match the query.

`maximum`

The largest of the numerical values that match the query.

`sumOfSquares`

The sum of the squares of the numerical values that match the query.

`variance`

The variance of the numerical values that match the query. The variance of a set of values is the average of the squares of the differences of each value from the average value of the set.

`stdDeviation`

The standard deviation of the numerical values that match the query. The standard deviation of a set of values is a measure of how spread out the values are.

The following example shows how to call `get-statistics` with a numerical custom field.

```
aws iot get-statistics --aggregation-field 'attributes.numericAttribute2'  
--query-string '*'
```

```
{  
    "statistics": {  
        "count": 3,  
        "average": 33.33333333333336,  
        "sum": 100.0,  
        "minimum": -125.0,  
        "maximum": 150.0,  
        "sumOfSquares": 43750.0,  
        "variance": 13472.22222222222,  
        "stdDeviation": 116.06990230986766  
    }  
}
```

For numerical aggregation fields, if the field values exceed the maximum double value, the statistics values are empty

GetCardinality

The [GetCardinality](#) API and the **get-cardinality** CLI command return the approximate count of unique values that match the query. For example, you might want to find the number of devices with battery levels at less than 50 percent:

```
aws iot get-cardinality --index-name AWS_Things --query-string "batteryLevel  
> 50" --aggregation-field "shadow.reported.batteryLevel"
```

This command returns the number of things with battery levels at more than 50 percent:

```
{  
    "cardinality": 100  
}
```

cardinality is always returned by **get-cardinality** even if there are no matching fields. For example:

```
aws iot get-cardinality --query-string "thingName:Non-existent*"  
--aggregation-field "attributes.customField_STR"
```

```
{  
    "cardinality": 0  
}
```

The **get-cardinality** CLI command takes the following parameters:

index-name

The name of the index to search. The default value is AWS_Things.

query-string

The query used to search the index. You can specify "*" to get the count of all indexed things in your AWS account.

aggregationField

The field to aggregate.

`query-version`

The version of the query to use. The default value is 2017-09-30.

GetPercentiles

The [GetPercentiles](#) API and the **get-percentiles** CLI command groups the aggregated values that match the query into percentile groupings. The default percentile groupings are: 1,5,25,50,75,95,99, although you can specify your own when you call GetPercentiles. This function returns a value for each percentile group specified (or the default percentile groupings). The percentile group "1" contains the aggregated field value that occurs in approximately one percent of the values that match the query. The percentile group "5" contains the aggregated field value that occurs in approximately five percent of the values that match the query, and so on. The result is an approximation, the more values that match the query, the more accurate the percentile values.

The following example shows how to call the **get-percentiles** CLI command.

```
aws iot get-percentiles --query-string "thingName:/*" --aggregation-field  
    "attributes.customField_NUM" --percents 10 20 30 40 50 60 70 80 90 99
```

```
{  
    "percentiles": [  
        {  
            "value": 3.0,  
            "percent": 80.0  
        },  
        {  
            "value": 2.5999999999999996,  
            "percent": 70.0  
        },  
        {  
            "value": 3.0,  
            "percent": 90.0  
        },  
        {  
            "value": 2.0,  
            "percent": 50.0  
        },  
        {  
            "value": 2.0,  
            "percent": 60.0  
        },  
        {  
            "value": 1.0,  
            "percent": 10.0  
        },  
        {  
            "value": 2.0,  
            "percent": 40.0  
        },  
        {  
            "value": 1.0,  
            "percent": 20.0  
        },  
        {  
            "value": 1.4,  
            "percent": 30.0  
        },  
        {  
            "value": 3.0,  
            "percent": 99.0  
        }  
    ]  
}
```

```
        }
    ]  
}
```

The following command shows the output returned from **get-percentiles** when there are no matching documents.

```
aws iot get-percentiles --query-string "thingName:Non-existent*"
--aggregation-field "attributes.customField_NUM"
```

```
{
    "percentiles": []
}
```

The **get-percentile** CLI command takes the following parameters:

index-name

The name of the index to search. The default value is `AWS_Things`.

query-string

The query used to search the index. You can specify "*" to get the count of all indexed things in your AWS account.

aggregationField

The field to aggregate, which must be of `Number` type.

query-version

The version of the query to use. The default value is `2017-09-30`.

percents

Optional. You can use this parameter to specify custom percentile groupings.

GetBucketsAggregation

The [GetBucketsAggregation](#) API and the **get-buckets-aggregation** CLI command return a list of buckets and the total number of things that fit the query string criteria.

The following example shows how to call the **get-buckets-aggregation** CLI command.

```
aws iot get-buckets-aggregation --query-string '*' --index-name AWS_Things --
aggregation-field 'shadow.reported.batteryLevelPercent' --buckets-aggregation-type
'termsAggregation={maxBuckets=5}'
```

This command returns the following:

```
{
    "totalCount": 20,
    "buckets": [
        {
            "keyValue": "100",
            "count": 12
        },
        {
            "keyValue": "90",
            "count": 5
        },
        {
            "keyValue": "80",
            "count": 3
        },
        {
            "keyValue": "70",
            "count": 2
        },
        {
            "keyValue": "60",
            "count": 1
        }
    ]
}
```

```
        "keyValue": "75",
        "count": 3
    ]
}
```

The get-buckets-aggregation CLI command takes the following parameters:

index-name

The name of the index to search. The default value is AWS_Things.

query-string

The query used to search the index. You can specify "*" to get the count of all indexed things in your AWS account.

aggregation-field

The field to aggregate.

buckets-aggregation-type

The basic control of the response shape and the bucket aggregation type to perform.

Authorization

You can specify the thing groups index as a resource ARN in an AWS IoT policy action, as follows.

Action	Resource
iot:GetStatistics	An index ARN (for example, arn:aws:iot: <i>your-aws-region</i> :index/AWS_Things or arn:aws:iot: <i>your-aws-region</i> :index/AWS_ThingGroups).

Query syntax

Queries are specified using a query syntax.

The query syntax supports the following features.

- Terms and phrases
- Searching fields
- Prefix search
- Range search
- Boolean operators AND, OR, NOT and -. The hyphen is used to exclude something from search results (for example, `thingName:(tv* AND -plasma)`).
- Grouping
- Field grouping
- Escaping special characters (as with \)

The query syntax does not support the following features:

- Leading wildcard search (such as `*xyz`), but searching for `**` matches all things
- Regular expressions

- Boosting
- Ranking
- Fuzzy searches
- Proximity search
- Sorting
- Aggregation

A few things to note about the query language:

- The default operator is AND. A query for "thingName:abc thingType:xyz" is equivalent to "thingName:abc AND thingType:xyz".
- If a field isn't specified, AWS IoT searches for the term in all fields.
- All field names are case sensitive.
- Search is case insensitive. Words are separated by white space characters as defined by Java's `Character.isWhitespace(int)`.
- Indexing of device shadow data includes reported, desired, delta, and metadata sections.
- Device shadow and registry versions are not searchable, but are present in the response.
- The maximum number of terms in a query is 5.

Example thing queries

Queries are specified in a query string using a query syntax and passed to the [SearchIndex API](#). The following table lists some example query strings.

Query string	Result
abc	Queries for "abc" in any registry or shadow field.
thingName:myThingName	Queries for a thing with name "myThingName".
thingName:my*	Queries for things with names that begin with "my".
thingName:ab?	Queries for things with names that have "ab" plus one additional character (for example: "aba", "abb", "abc", and so on.)
thingTypeName:aa	Queries for things that are associated with type aa.
attributes.myAttribute:75	Queries for things with an attribute named "myAttribute" that has the value 75.
attributes.myAttribute:[75 TO 80]	Queries for things with an attribute named "myAttribute" whose value falls within a numeric range (75–80, inclusive).
attributes.myAttribute:{75 TO 80}	Queries for things with an attribute named "myAttribute" whose value falls within the numeric range (>75 and <=80).
attributes.serialNumber:["abcd" TO "abcf"]	Queries for things with an attribute named "serialNumber" whose value is within an

Query string	Result
	alphanumeric string range. This query returns things with a "serialNumber" attribute with values "abcd", "abce", or "abcf".
attributes.myAttribute:i*t	Queries for things with an attribute named "myAttribute" whose value is 'i', followed by any number of characters, followed by 't'.
attributes.attr1:abc AND attributes.attr2<5 NOT attributes.attr3>10	Queries for things that combine terms using Boolean expressions. This query returns things that have an attribute named "attr1" with a value "abc", an attribute named "attr2" that is less than 5, and an attribute named "attr3" that is not greater than 10.
shadow.hasDelta:true	Queries for things whose shadow has a delta element.
NOT attributes.model:legacy	Queries for things where the attribute named "model" is not "legacy".
shadow.reported.stats.battery:{70 TO 100} (v2 OR v3) NOT attributes.model:legacy	Queries for things with the following: <ul style="list-style-type: none"> The thing's shadow stats .battery attribute has a value between 70 and 100. The text "v2" or "v3" occurs in a thing's name, type name, or attribute values. The thing's model attribute is not set to "legacy".
shadow.reported.myvalues:2	Queries for things where the myvalues array in the shadow's reported section contains a value of 2.
shadow.reported.location:* NOT shadow.desired.stats.battery:*	Queries for things with the following: <ul style="list-style-type: none"> The location attribute exists in the shadow's reported section. The stats .battery attribute does not exist in the shadow's desired section.
connectivity.connected:true	Queries for all connected devices.
connectivity.connected:false	Queries for all disconnected devices.
connectivity.connected:true AND connectivity.timestamp : [1557651600000 TO 1557867600000]	Queries for all connected devices with a connect timestamp \geq 1557651600000 and \leq 1557867600000. Timestamps are given in milliseconds since epoch.
connectivity.connected:false AND connectivity.timestamp : [1557651600000 TO 1557867600000]	Queries for all disconnected devices with a disconnect timestamp \geq 1557651600000 and \leq 1557867600000. Timestamps are given in milliseconds since epoch.

Query string	Result
connectivity.connected:true AND connectivity.timestamp > 1557651600000	Queries for all connected devices with a connect timestamp > 1557651600000. Timestamps are given in milliseconds since epoch.
connectivity.connected:*	Queries for all devices with connectivity information present.
connectivity.disconnectReason:*	Queries for all devices with connectivity disconnectReason present.
connectivity.disconnectReason:CLIENT_INITIATED_DISCONNECT	Queries for all devices disconnected due to CLIENT_INITIATED_DISCONNECT.

Example thing group queries

Queries are specified in a query string using a query syntax and passed to the [SearchIndex API](#). The following table lists some example query strings.

Query string	Result
abc	Queries for "abc" in any field.
thingGroupName:myGroupThingName	Queries for a thing group with name "myGroupThingName".
thingGroupName:my*	Queries for thing groups with names that begin with "my".
thingGroupName:ab?	Queries for thing groups with names that have "ab" plus one additional character (for example: "aba", "abb", "abc", and so on.)
attributes.myAttribute:75	Queries for thing groups with an attribute named "myAttribute" that has the value 75.
attributes.myAttribute:[75 TO 80]	Queries for thing groups with an attribute named "myAttribute" whose value falls within a numeric range (75–80, inclusive).
attributes.myAttribute:[75 TO 80]	Queries for thing groups with an attribute named "myAttribute" whose value falls within the numeric range (>75 and <=80).
attributes.myAttribute:[“abcd” TO “abcf”]	Queries for thing groups with an attribute named "myAttribute" whose value is within an alphanumeric string range. This query returns thing groups with a "serialNumber" attribute with values "abcd", "abce", or "abcf".
attributes.myAttribute:i*t	Queries for thing groups with an attribute named "myAttribute" whose value is 'i', followed by any number of characters, followed by 't'.
attributes.attr1:abc AND attributes.attr2<5 NOT attributes.attr3>10	Queries for thing groups that combine terms using Boolean expressions. This query returns

Query string	Result
	thing groups that have an attribute named "attr1" with a value "abc", an attribute named "attr2" that is less than 5, and an attribute named "attr3" that is not greater than 10.
NOT attributes.myAttribute:cde	Queries for thing groups where the attribute named "myAttribute" is not "cde".
parentGroupNames:(myParentThingGroupName)	Queries for thing groups whose parent group name matches "myParentThingGroupName".
parentGroupNames:(myParentThingGroupName OR myRootThingGroupName)	Queries for thing groups whose parent group name matches "myParentThingGroupName" or "myRootThingGroupName".
parentGroupNames:(myParentThingGroupNa*)	Queries for thing groups whose parent group name begins with "myParentThingGroupNa".

Fleet metrics

Fleet metrics is a feature of [fleet indexing \(p. 732\)](#), a managed service that allows you to index, search, and aggregate your devices' data in AWS IoT. With fleet metrics, you can monitor your fleet devices' aggregate state in [CloudWatch](#) over time, including reviewing your fleet devices' disconnection rate or average battery level changes of a specified period.

Using fleet metrics, you can build [aggregation queries \(p. 744\)](#) whose results are continually emitted to [CloudWatch](#) as metrics for analyzing trends and creating alarms. For your monitoring tasks, you can specify the aggregation queries of different aggregation types (**Statistics**, **Cardinality**, and **Percentile**). You can save all of your aggregation queries to create fleet metrics for reuse in the future.

Getting started tutorial

In this tutorial, you create a [fleet metric \(p. 753\)](#) to monitor your sensors' temperatures to detect potential anomalies. When creating the fleet metric, you define an [aggregation query \(p. 744\)](#) that detects the number of sensors with temperatures exceeding 80 degrees Fahrenheit. You specify the query to run every 60 seconds and the query results are emitted to CloudWatch, where you can view the number of sensors that have potential high-temperature risks, and set alarms. To complete this tutorial, you'll use [AWS CLI](#).

In this tutorial, you'll learn how to:

- [Set up \(p. 754\)](#)
- [Create fleet metrics \(p. 755\)](#)
- [View metrics in CloudWatch \(p. 756\)](#)
- [Clean up resources \(p. 758\)](#)

This tutorial takes about 15 minutes to complete.

Prerequisites

- Install the latest version of [AWS CLI](#)
- Familiarize yourself with [Querying for aggregate data](#)
- Familiarize yourself with [Using Amazon CloudWatch metrics](#)

Set up

To use fleet metrics, you should enable fleet indexing. To enable fleet indexing for your things or thing groups with specified data sources and associated configurations, follow the instructions in [Managing thing indexing \(p. 732\)](#) and [Managing thing group indexing \(p. 742\)](#).

To set up

- Run the following command to enable fleet indexing and specify the data sources to search from.

```
aws iot update-indexing-configuration \
--thing-indexing-configuration
"thingIndexingMode=REGISTRY_AND_SHADOW,customFields=[{name=attributes.temperature,type=Number},
{name=attributes.rackId,type=String},
{name=attributes.stateNormal,type=Boolean}],thingConnectivityIndexingMode=STATUS" \
```

The example CLI command above enables fleet indexing to support searching registry data, shadow data, and thing connectivity status using the AWS_Things index.

The configuration change can take a few minutes to complete. Make sure your fleet indexing is enabled before you create fleet metrics.

To check if your fleet indexing has been enabled, run the following CLI command:

```
aws --region us-east-1 iot describe-index --index-name "AWS_Things"
```

For more information, read [Enable thing indexing \(p. 732\)](#).

- Run the following bash script to create ten things and describe them.

```
# Bash script. Type `bash` before running in other shells.

Temperatures=(70 71 72 73 74 75 47 97 98 99)
Racks=(Rack1 Rack1 Rack2 Rack2 Rack3 Rack4 Rack5 Rack6 Rack6 Rack6)
IsNormal=(true true true true true true false false false)

for ((i=0; i < 10; i++))
do
    thing=$(aws iot create-thing --thing-name "TempSensor$i" --attribute-payload
    attributes="{temperature=${Temperatures[@]:$i:1},rackId=${Racks[@]:$i:1},stateNormal=
    ${IsNormal[@]:$i:1}}")
    aws iot describe-thing --thing-name "TempSensor$i"
done
```

This script creates ten things to represent ten sensors. Each thing has attributes of `temperature`, `rackId`, and `stateNormal` as described in the following table:

Attribute	Data type	Description
<code>temperature</code>	Number	Temperature value in Fahrenheit
<code>rackId</code>	String	ID of the server rack that contains sensors
<code>stateNormal</code>	Boolean	Whether the sensor's temperature value is normal or not

The output of this script contains ten JSON files. One of the JSON file looks like the following:

```
{  
    "version": 1,  
    "thingName": "TempSensor0",  
    "defaultClientId": "TempSensor0",  
    "attributes": {  
        "rackId": "Rack1",  
        "stateNormal": "true",  
        "temperature": "70"  
    },  
    "thingArn": "arn:aws:iot:region:account:thing/TempSensor0",  
    "thingId": "example-thing-id"  
}
```

For more information, read [Create a thing](#).

Create fleet metrics

To create a fleet metric

1. Run the following command to create a fleet metric named **high_temp_FM**. You create the fleet metric to monitor the number of sensors with temperatures exceeding 80 degrees Fahrenheit in CloudWatch.

```
aws iot create-fleet-metric --metric-name "high_temp_FM" --query-string  
"thingName:TempSensor* AND attributes.temperature >80" --period 60 --aggregation-field  
"attributes.temperature" --aggregation-type name=Statistics,values=count
```

--metric-name

Data type: string. The --metric-name parameter specifies a fleet metric name. In this example, you're creating a fleet metric named **high_temp_FM**.

--query-string

Data type: string. The --query-string parameter specifies the query string. In this example, the query string means to query all the things with names starting with *TempSensor* and with temperatures higher than 80 degrees Fahrenheit. For more information, read [Query syntax \(p. 749\)](#).

--period

Data type: integer. The --period parameter specifies the time to retrieve the aggregated data in seconds. In this example, you specify that the fleet metric you're creating retrieves the aggregated data every 60 seconds.

--aggregation-field

Data type: string. The --aggregation-field parameter specifies the attribute to evaluate. In this example, the temperature attribute is to be evaluated.

--aggregation-type

The --aggregation-type parameter specifies the statistical summary to display in the fleet metric. For your monitoring tasks, you can customize the aggregation query properties for the

different aggregation types (**Statistics**, **Cardinality**, and **Percentile**). In this example, you specify **count** for the aggregation type **Statistics** to return the count of devices that have attributes that match the query, in other words, to return the count of the devices with names starting with *TempSensor* and with temperatures higher than 80 degrees Fahrenheit. For more information, read [Querying for aggregate data \(p. 744\)](#).

The output of this command looks like the following:

```
{  
    "metricArn": "arn:aws:iot:region:111122223333:fleetmetric/high_temp_FM",  
    "metricName": "high_temp_FM"  
}
```

Note

It can take a moment for the data points to display in CloudWatch.

To learn more about how to create a fleet metric, read [Managing fleet metrics \(p. 758\)](#).

If you can't create a fleet metric, read [Troubleshooting fleet metrics \(p. 1137\)](#).

2. (Optional) Run the following command to describe your fleet metric named *high_temp_FM*:

```
aws iot describe-fleet-metric --metric-name "high_temp_FM"
```

The output of this command looks like the following:

```
{  
    "queryVersion": "2017-09-30",  
    "lastModifiedDate": 1625249775.834,  
    "queryString": "*",  
    "period": 60,  
    "metricArn": "arn:aws:iot:region:111122223333:fleetmetric/high_temp_FM",  
    "aggregationField": "registry.version",  
    "version": 1,  
    "aggregationType": {  
        "values": [  
            "sum"  
        ],  
        "name": "Statistics"  
    },  
    "indexName": "AWS_Things",  
    "creationDate": 1625249775.834,  
    "metricName": "high_temp_FM"  
}
```

View fleet metrics in CloudWatch

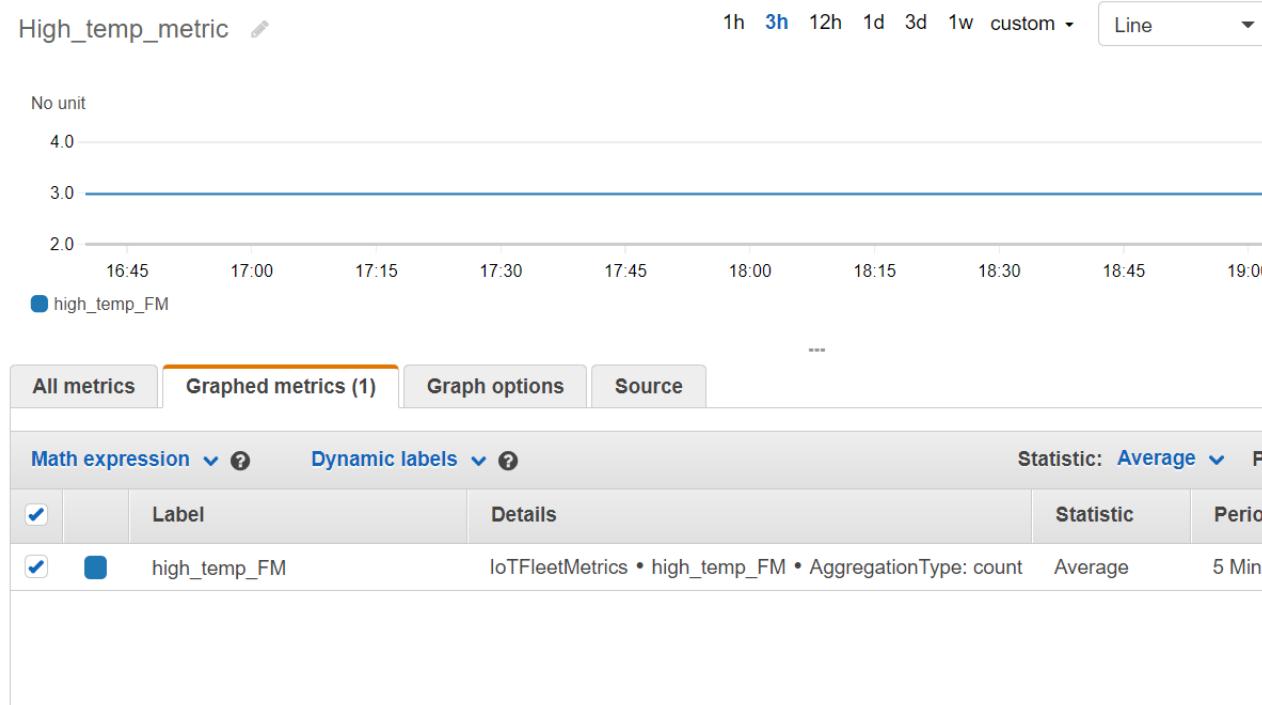
After creating the fleet metric, you can view the metric data in CloudWatch. In this tutorial, you will see the metric that shows the number of sensors with names starting with *TempSensor* and with temperatures higher than 80 degrees Fahrenheit.

To view data points in CloudWatch

1. Open the CloudWatch console at <https://console.aws.amazon.com/cloudwatch/>.
2. On the CloudWatch menu on the left, choose **Metrics** to expand the sub-menu and then choose **All metrics**. This opens the page with the upper half to display the graph and the lower half containing four tabbed sections.

3. The first tabbed section **All metrics** lists all the metrics you can view in groups, choose **IoTFleetMetrics** that contains all your fleet metrics.
4. On the **Aggregation type** section of the **All metrics** tab, choose **Aggregation type** to view all the fleet metrics you created.
5. Choose the fleet metric to display graph on the left of the **Aggregation type** section. You will see the value **sum** to the left of your **Metric name**, and this is value of the aggregation type you specified in the [Create fleet metrics \(p. 755\)](#) section of this tutorial.
6. Choose the second tab named **Graphed metrics** to the right of the **All metrics** tab to view the fleet metric you chose from the previous step.

You should be able to see a graph that displays the number of sensors with temperatures higher than 80 degrees Fahrenheit like the following:



Note

The **Period** attribute in CloudWatch defaults to 5 minutes. It's the time interval between data points displaying in CloudWatch. You can change the **Period** setting based on your needs.

7. (Optional) You can set a metric alarm.

1. On the CloudWatch menu on the left, choose **Alarms** to expand the sub-menu and then choose **All alarms**.
2. On the **Alarms** page, choose **Create alarm** on the upper right corner. Follow the **Create alarm** instructions in console to create an alarm as needed. For more information, read [Using Amazon CloudWatch alarms](#).

To learn more, read [Using Amazon CloudWatch metrics](#).

If you can't see data points in CloudWatch, read [Troubleshooting fleet metrics \(p. 1137\)](#).

Clean up

To delete fleet metrics

You use the **delete-fleet-metric** CLI command to delete fleet metrics.

Run the following command to delete the fleet metric named *high_temp_FM*.

```
aws iot delete-fleet-metric --metric-name "high_temp_FM"
```

To clean up things

You use the **delete-thing** CLI command to delete things.

Run the following script to delete the ten things you created:

```
# Bash script. Type `bash` before running in other shells.  
  
for ((i=0; i < 10; i++))  
do  
    thing=$(aws iot delete-thing --thing-name "TempSensor$i")  
done
```

To clean up metrics in CloudWatch

CloudWatch doesn't support metrics deletion. Metrics expire based on their retention schedules. To learn more, read [Using Amazon CloudWatch metrics](#).

Managing fleet metrics

This topic shows how to use the AWS IoT console and AWS CLI to manage your fleet metrics.

Managing fleet metrics (Console)

Make sure you've enabled fleet indexing with associated data sources and configurations before creating fleet metrics.

Enable fleet indexing

If you've already enabled fleet indexing, skip this section.

If you haven't enabled fleet indexing, follow these instructions.

1. Open your AWS IoT console at <https://console.aws.amazon.com/iot/>.
2. On the AWS IoT menu, choose **Settings**.
3. To view the detailed settings, on the **Settings** page, scroll down to the **Fleet indexing** section.
4. To update your fleet indexing settings, to the right of the **Fleet indexing** section, select **Manage indexing**.
5. On the **Manage fleet indexing** page, update your fleet indexing settings based on your needs.
 - **Configuration**

To turn on thing indexing, toggle **Thing indexing** on, and then select the data sources you want to index from.

To turn on thing group indexing, toggle **Thing group indexing** on.

- **Custom search fields - optional**

Custom search fields are a list of field name and field type pairs.

To add a custom field pair, choose **Add new field**. Enter a custom field name such as `attributes.temperature`, then select a field type from the **Field type** menu. Note that a custom field name begins with `attributes.` and will be saved as an attribute to run [thing aggregations queries](#).

To update and save the setting, choose **Update**.

Create a fleet metric

1. Open your AWS IoT console at <https://console.aws.amazon.com/iot/>.
2. On the AWS IoT menu, choose **Manage**, and then choose **Fleet metrics**.
3. On the **Fleet metrics** page, choose **Create fleet metric** and complete the creation steps.
4. In step 1 **Configure fleet metrics**
 - In **Query** section, enter a query string to specify the things or thing groups you want to perform the aggregate search. The query string consists of an attribute and a value. For **Properties**, choose the attribute you want, or, if it doesn't appear in the list, enter the attribute in the field. Enter the value after `:`. An example query string can be `thingName:TempSensor*`. For each query string you enter, press **enter** in your keyboard. If you enter multiple query strings, specify their relationship by selecting **and**, **or**, **and not**, or **or not** between them.
 - In **Report properties**, choose **Index name**, **Aggregation type**, and **Aggregation field** from their respective lists. Next, select the data you want to aggregate in **Select data**, where you can select multiple data values.
 - Choose **Next**.
5. In step 2 **Specify fleet metric properties**
 - In **Fleet metric name** field, enter a name for the fleet metric you're creating.
 - In **Description - optional** field, enter a description for the fleet metric you're creating. This field is optional.
 - In **Hours** and **Minutes** fields, enter the time (how often) you want the fleet metric to emit data to CloudWatch.
 - Choose **Next**.
6. In step 3 **Review and create**
 - Review the settings of step 1 and step 2. To edit the settings, choose **Edit**.
 - Choose **Create fleet metric**.

After successful creation, the fleet metric is listed on the **Fleet metric** page.

Update a fleet metric

1. On the **Fleet metric** page, choose the fleet metric you want to update.
2. On the fleet metric **Details** page, choose **Edit**. This opens the creation steps where you can update your fleet metric in any of the three steps.
3. After you finish updating the fleet metric, choose **Update fleet metric**.

Delete a fleet metric

1. On the **Fleet metric** page, choose the fleet metric you want to delete.
2. On the next page that shows details of your fleet metric, choose **Delete**.
3. In the dialog box, enter the name of your fleet metric to confirm deletion.

4. Choose **Delete**. This step deletes your fleet metric permanently.

Managing fleet metrics (CLI)

The following sections show how to use the AWS CLI to manage your fleet metrics. Make sure you've enabled fleet indexing with associated data sources and configurations before creating fleet metrics. To enable fleet indexing for your things or thing groups, follow the instructions in [Managing thing indexing \(p. 732\)](#) or [Managing thing group indexing \(p. 742\)](#).

Create a fleet metric

You use the create-fleet-metric CLI command to create a fleet metric.

```
aws iot create-fleet-metric --metric-name "YourFleetMetricName" --query-string "*" --period 60 --aggregation-field "registry.version" --aggregation-type name=Statistics,values=sum
```

The output of this command contains the name and Amazon Resource Name (ARN) of your fleet metric. The output looks like the following:

```
{  
    "metricArn": "arn:aws:iot:us-east-1:111122223333:fleetmetric/YourFleetMetricName",  
    "metricName": "YourFleetMetricName"  
}
```

List fleet metrics

You use the list-fleet-metric CLI command to list all the fleet metrics in your account.

```
aws iot list-fleet-metrics
```

The output of this command contains all your fleet metrics. The output looks like the following:

```
{  
    "fleetMetrics": [  
        {  
            "metricArn": "arn:aws:iot:us-east-1:111122223333:fleetmetric/YourFleetMetric1",  
            "metricName": "YourFleetMetric1"  
        },  
        {  
            "metricArn": "arn:aws:iot:us-east-1:111122223333:fleetmetric/YourFleetMetric2",  
            "metricName": "YourFleetMetric2"  
        }  
    ]  
}
```

Describe a fleet metric

You use the describe-fleet-metric CLI command to display more detailed information about a fleet metric.

```
aws iot describe-fleet-metric --metric-name "YourFleetMetricName"
```

The output of command contains the detailed information about the specified fleet metric. The output looks like the following:

```
{
    "queryVersion": "2017-09-30",
    "lastModifiedDate": 1625790642.355,
    "queryString": "*",
    "period": 60,
    "metricArn": "arn:aws:iot:us-east-1:111122223333:fleetmetric/YourFleetMetricName",
    "aggregationField": "registry.version",
    "version": 1,
    "aggregationType": {
        "values": [
            "sum"
        ],
        "name": "Statistics"
    },
    "indexName": "AWS_Things",
    "creationDate": 1625790642.355,
    "metricName": "YourFleetMetricName"
}
```

Update a fleet metric

You use the update-fleet-metric CLI command to update a fleet metric.

```
aws iot update-fleet-metric --metric-name "YourFleetMetricName" --query-string
"*" --period 120 --aggregation-field "registry.version" --aggregation-type
name=Statistics,values=sum,count --index-name AWS_Things
```

The update-fleet-metric command doesn't produce any output. You can use the describe-fleet-metric CLI command to see the result.

```
{
    "queryVersion": "2017-09-30",
    "lastModifiedDate": 1625792300.881,
    "queryString": "*",
    "period": 120,
    "metricArn": "arn:aws:iot:us-east-1:111122223333:fleetmetric/YourFleetMetricName",
    "aggregationField": "registry.version",
    "version": 2,
    "aggregationType": {
        "values": [
            "sum",
            "count"
        ],
        "name": "Statistics"
    },
    "indexName": "AWS_Things",
    "creationDate": 1625792300.881,
    "metricName": "YourFleetMetricName"
}
```

Delete a fleet metric

You use the delete-fleet-metric CLI command to delete a fleet metric.

```
aws iot delete-fleet-metric --metric-name "YourFleetMetricName"
```

This command doesn't produce any output if the deletion is successful or if you specify a fleet metric that doesn't exist.

For more information, read [Troubleshooting fleet metrics \(p. 1137\)](#).

MQTT-based file delivery

One option you can use to manage files and transfer them to AWS IoT devices in your fleet is MQTT-based file delivery. With this feature in the AWS Cloud you can create a *stream* that contains multiple files, you can update stream data (the file list and descriptions), get the stream data, and more. AWS IoT MQTT-based file delivery can transfer data in small blocks to your IoT devices, using the MQTT protocol with support for request and response messages in JSON or CBOR.

For more information on ways to transfer data to and from IoT devices using AWS IoT, see [Connecting devices to AWS IoT \(p. 74\)](#).

Topics

- [What is a stream? \(p. 762\)](#)
- [Managing a stream in the AWS Cloud \(p. 763\)](#)
- [Using AWS IoT MQTT-based file delivery in devices \(p. 764\)](#)
- [An example use case in FreeRTOS OTA \(p. 771\)](#)

What is a stream?

In AWS IoT, a *stream* is a publicly addressable resource that is an abstraction for a list of files that can be transferred to an IoT device. A typical stream contains the following information:

- **An Amazon Resource Name (ARN)** that uniquely identifies a stream at a given time. This ARN has the pattern `arn:partition:iot:region:account-ID:stream/stream ID`.
- **A stream ID** that identifies your stream and is used (and usually required) in AWS Command Line Interface (AWS CLI) or SDK commands.
- **A stream description** that provides a description of the stream resource.
- **A stream version** that identifies a particular version of the stream. Because stream data can be modified immediately before devices start the data transfer, the stream version can be used by the devices to enforce a consistency check.
- **A list of files** that can be transferred to devices. For each file in the list, the stream records a file ID, the file size, and the address information of the file, which consists of, for example, the Amazon S3 bucket name, object key, and object version.
- **An AWS Identity and Access Management (IAM) role** that grants AWS IoT MQTT-based file delivery the permission to read stream files stored in data storage.

AWS IoT MQTT-based file delivery provides the following functionality so that devices can transfer data from the AWS Cloud:

- Data transfer using the MQTT protocol.
- Support for JSON or CBOR formats.
- The ability to describe a stream (`DescribeStream` API) to get a stream file list, stream version, and related information.
- The ability to send data in small blocks (`GetStream` API) so that devices with hardware constraints can receive the blocks.

- Support for a dynamic block size per request, to support devices that have different memory capacities.
- Optimization for concurrent streaming requests when multiple devices request data blocks from the same stream file.
- Amazon S3 as data storage for stream files.
- Support for data transfer log publishing from AWS IoT MQTT-based file delivery to CloudWatch.

For MQTT-based file delivery quotas, see [AWS IoT Core Service Quotas](#) in the *AWS General Reference*.

Managing a stream in the AWS Cloud

AWS IoT provides AWS SDK and AWS CLI commands that you can use to manage a stream in the AWS Cloud. You can use these commands to do the following:

- Create a stream. [CLI / SDK](#)
- Describe a stream to get its information. [CLI / SDK](#)
- List streams in your AWS account. [CLI / SDK](#)
- Update the file list or stream description in a stream. [CLI / SDK](#)
- Delete a stream. [CLI / SDK](#)

Note

At this time, streams are not visible in the AWS Management Console. You must use the AWS CLI or AWS SDK to manage a stream in AWS IoT.

Before you use AWS IoT MQTT-based file delivery from your devices, you must follow the steps in the next sections to make sure that your devices are properly authorized and can connect to the AWS IoT Device Gateway.

Grant permissions to your devices

You can follow the steps in [Create an AWS IoT policy](#) to create a device policy or use an existing device policy. Attach the policy to the certificates associated with your devices and add the following permissions to the device policy.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [ "iot:Connect" ],  
            "Resource": [  
                "arn:partition:iot:region:accountID:client/  
${iot:Connection.Thing.ThingName}"  
            ]  
        },  
        {  
            "Effect": "Allow",  
            "Action": [ "iot:Receive", "iot:Publish" ],  
            "Resource": [  
                "arn:partition:iot:region:accountID:topic/$aws/things/  
${iot:Connection.Thing.ThingName}/streams/*"  
            ]  
        }  
    ]  
}
```

```
        "Effect": "Allow",
        "Action": "iot:Subscribe",
        "Resource": [
            "arn:partition:iot:region:accountID:topicfilter/$aws/things/
${iot:Connection.Thing.ThingName}/streams/*"
        ]
    }
}
```

Connect your devices to AWS IoT

Devices that use AWS IoT MQTT-based file delivery are required to connect with AWS IoT. AWS IoT MQTT-based file delivery integrates with AWS IoT in the AWS Cloud, so your devices should directly connect to [the endpoint of the AWS IoT Data Plane](#).

Note

The endpoint of the AWS IoT data plane is specific to the AWS account and Region. You must use the endpoint for the AWS account and the Region in which your devices are registered in AWS IoT.

See [Connecting to AWS IoT Core \(p. 66\)](#) for more information.

Using AWS IoT MQTT-based file delivery in devices

To initiate the data transfer process, a device must receive an **initial data set**, which includes a stream ID at minimum. You can use an [Jobs \(p. 637\)](#) to schedule data transfer tasks for your devices by including the initial data set in the job document. When a device receives the initial data set, it should then start the interaction with AWS IoT MQTT-based file delivery. To exchange data with AWS IoT MQTT-based file delivery, a device should:

- Use the MQTT protocol to subscribe to the [MQTT-based file delivery topics \(p. 108\)](#).
- Send requests and then wait to receive the responses using MQTT messages.

You can optionally include a stream file ID and a stream version in the initial data set. Sending a stream file ID to a device can simplify the programming of the device's firmware/software, because it eliminates the need to make a `DescribeStream` request from the device to get this ID. The device can specify the stream version in a `GetStream` request to enforce a consistency check in case the stream has been updated unexpectedly.

Use `DescribeStream` to get stream data

AWS IoT MQTT-based file delivery provides the `DescribeStream` API to send stream data to a device. The stream data returned by this API includes the stream ID, stream version, stream description and a list of stream files, each of which has a file ID and the file size in bytes. With this information, a device can select arbitrary files to initiate the data transfer process.

Note

You don't need to use the `DescribeStream` API if your device receives all required stream file IDs in the initial data set.

Follow these steps to make a `DescribeStream` request.

1. Subscribe to the "accepted" topic filter `$aws/things/ThingName/streams/StreamId/description/json`.

2. Subscribe to the "rejected" topic filter `$aws/things/ThingName/streams/StreamId/rejected/json`.
3. Publish a `DescribeStream` request by sending a message to `$aws/things/ThingName/streams/StreamId/describe/json`.
4. If the request was accepted, your device receives a `DescribeStream` response on the "accepted" topic filter.
5. If the request was rejected, your device receives the error response on the "rejected" topic filter.

Note

If you replace `json` with `cbor` in the topics and topic filters shown, your device receives messages in the CBOR format, which is more compact than JSON.

DescribeStream request

A typical `DescribeStream` request in JSON looks like the following example.

```
{  
    "c": "ec944cfb-1e3c-49ac-97de-9dc4aaad0039"  
}
```

- (Optional) "c" is the client token field.

The client token can't be longer than 64 bytes. A client token that is longer than 64 bytes causes an error response and an `InvalidRequest` error message.

DescribeStream response

A `DescribeStream` response in JSON looks like the following example.

```
{  
    "c": "ec944cfb-1e3c-49ac-97de-9dc4aaad0039",  
    "s": 1,  
    "d": "This is the description of stream ABC.",  
    "r": [  
        {  
            "f": 0,  
            "z": 131072  
        },  
        {  
            "f": 1,  
            "z": 51200  
        }  
    ]  
}
```

- "c" is the client token field. This is returned if it was given in the `DescribeStream` request. Use the client token to associate the response with its request.
- "s" is the stream version as an integer. You can use this version to perform a consistency check with your `GetStream` requests.
- "r" contains a list of the files in the stream.
 - "f" is the stream file ID as an integer.
 - "z" is the stream file size in number of bytes.
- "d" contains the description of the stream.

Get data blocks from a stream file

You can use the `GetStream` API so that a device can receive stream files in small data blocks, so it can be used by those devices that have constraints on processing large block sizes. To receive an entire data file, a device might need to send or receive multiple requests and responses until all data blocks are received and processed.

GetStream request

Follow these steps to make a `GetStream` request.

1. Subscribe to the "accepted" topic filter `$aws/things/ThingName/streams/StreamId/data/json`.
2. Subscribe to the "rejected" topic filter `$aws/things/ThingName/streams/StreamId/rejected/json`.
3. Publish a `GetStream` request to the topic `$aws/things/ThingName/streams/StreamId/get/json`.
4. If the request was accepted, your device will receive one or more `GetStream` responses on the "accepted" topic filter. Each response message contains basic information and a data payload for a single block.
5. Repeat steps 3 and 4 to receive all data blocks. You must repeat these steps if the amount of data requested is larger than 128 KB. You must program your device to use multiple `GetStream` requests to receive all of the data requested.
6. If the request was rejected, your device will receive the error response on the "rejected" topic filter.

Note

- If you replace "json" with "cbor" in the topics and topic filters shown, your device will receive messages in the CBOR format, which is more compact than JSON.
- AWS IoT MQTT-based file delivery limits the size of a block to 128 KB. If you make a request for a block that is more than 128 KB, the request will fail.
- You can make a request for multiple blocks whose total size is greater than 128 KB (for example, if you make a request for 5 blocks of 32 KB each for a total of 160 KB of data). In this case, the request doesn't fail, but your device must make multiple requests in order to receive all of the data requested. The service will send additional blocks as your device makes additional requests. We recommend that you continue with a new request only after the previous response has been correctly received and processed.
- Regardless of the total size of data requested, you should program your device to initiate retries when blocks are not received, or not received correctly.

A typical `GetStream` request in JSON looks like the following example.

```
{  
  "c": "1bb8aaa1-5c18-4d21-80c2-0b44fee10380",  
  "s": 1,  
  "f": 0,  
  "l": 4096,  
  "o": 2,  
  "n": 100,  
  "b": "..."  
}
```

- [optional] "c" is the client token field.

The client token can be no longer than 64 bytes. A client token that is longer than 64 bytes causes an error response and an `InvalidRequest` error message.

- [optional] "s" is the stream version field (an integer).

MQTT-based file delivery applies a consistency check based on this requested version and the latest stream version in the cloud. If the stream version sent from a device in a `GetStream` request doesn't match the latest stream version in the cloud, the service sends an error response and a `VersionMismatch` error message. Typically, a device receives the expected (latest) stream version in the initial data set or in the response to `DescribeStream`.

- "f" is the stream file ID (an integer in the range 0 to 255).

The stream file ID is required when you create or update a stream using the AWS CLI or SDK. If a device requests a stream file with an ID that doesn't exist, the service sends an error response and a `ResourceNotFound` error message.

- "l" is the data block size in bytes (an integer in the range 256 to 131,072).

Refer to [Build a bitmap for a GetStream request \(p. 768\)](#) for instructions on how to use the bitmap fields to specify what portion of the stream file will be returned in the `GetStream` response. If a device specifies a block size that is out of range, the service sends an error response and a `BlockSizeOutOfBounds` error message.

- [optional] "o" is the offset of the block in the stream file (an integer in the range 0 to 98,304).

Refer to [Build a bitmap for a GetStream request \(p. 768\)](#) for instructions on how to use the bitmap fields to specify what portion of the stream file will be returned in the `GetStream` response. The maximum value of 98,304 is based on a 24 MB stream file size limit and 256 bytes for the minimum block size. The default is 0 if not specified.

- [optional] "n" is the number of blocks requested (an integer in the range 0 to 98,304).

The "n" field specifies either (1) the number of blocks requested, or (2) when the bitmap field ("b") is used, a limit on the number of blocks that will be returned by the bitmap request. This second use is optional. If not defined, it defaults to 131072/`DataBlockSize`.

- [optional] "b" is a bitmap that represents the blocks being requested.

Using a bitmap, your device can request non-consecutive blocks, which makes handling retries following an error more convenient. Refer to [Build a bitmap for a GetStream request \(p. 768\)](#) for instructions on how to use the bitmap fields to specify which portion of the stream file will be returned in the `GetStream` response. For this field, convert the bitmap to a string representing the bitmap's value in hexadecimal notation. The bitmap must be less than 12,288 bytes.

Important

Either "n" or "b" should be specified. If neither of them is specified, the `GetStream` request might not be valid when the file size is less than 131072 bytes (128 KB).

GetStream response

A `GetStream` response in JSON looks like this example for each data block that is requested.

```
{  
    "c": "1bb8aaa1-5c18-4d21-80c2-0b44fee10380",  
    "f": 0,  
    "l": 4096,  
    "i": 2,  
    "p": "..."  
}
```

- "c" is the client token field. This is returned if it was given in the `GetStream` request. Use the client token to associate the response with its request.
 - "f" is the ID of the stream file to which the current data block payload belongs.
 - "l" is the size of the data block payload in bytes.
 - "i" is the ID of the data block contained in the payload. Data blocks are numbered starting from 0.
 - "p" contains the data block payload. This field is a string, which represents the value of the data block in [Base64](#) encoding.

Build a bitmap for a GetStream request

You can use the **bitmap** field (**b**) in a `GetStream` request to get non-consecutive blocks from a stream file. This helps devices with limited RAM capacity deal with network delivery issues. A device can request only those blocks that were not received or not received correctly. The bitmap determines which blocks of the stream file will be returned. For each bit, which is set to 1 in the bitmap, a corresponding block of the stream file will be returned.

Here's an example of how to specify a bitmap and its supporting fields in a GetStream request. For example, you want to receive a stream file in chunks of 256 bytes (the block size). Think of each block of 256 bytes as having a number that specifies its position in the file, starting from 0. So block 0 is the first block of 256 bytes in the file, block 1 is the second, and so on. You want to request blocks 20, 21, 24 and 43 from the file.

Block offset

Because the first block is number 20, specify the offset (field o) as 20 to save space in the bitmap.

Number of blocks

To ensure that your device doesn't receive more blocks than it can handle with limited memory resources, you can specify the maximum number of blocks that should be returned in each message sent by MQTT-based file delivery. Note that this value is disregarded if the bitmap itself specifies less than this number of blocks, or if it would make the total size of the response messages sent by MQTT-based file delivery greater than the service limit of 128 KB per `GetStream` request.

Block bitmap

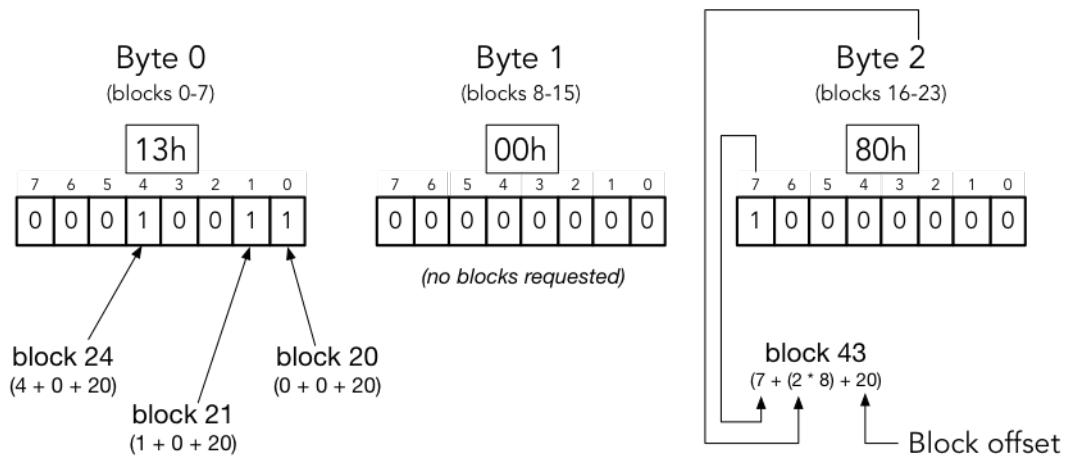
The bitmap itself is an array of unsigned bytes expressed in hexadecimal notation, and included in the `GetStream` request as a string representation of the number. But to construct this string, let's start by thinking of the bitmap as a long sequence of bits (a binary number). If a bit in this sequence is set to 1, the corresponding block from the stream file will be sent back to the device. For our example, we want to receive blocks 20, 21, 24, and 43, so we must set bits 20, 21, 24, and 43 in our bitmap. We can use the block offset to save space, so after we subtract the offset from each block number, we want to set bits 0, 1, 4, and 23, like the following example.

Taking one byte (8 bits) at a time, this is conventionally written as: "0b00010011", "0b00000000", and "0b10000000". Bit 0 shows up in our binary representation at the end of the first byte, and bit 23 at the beginning of the last. This can be confusing unless you know the conventions. The first byte contains bits 7-0 (in that order), the second byte contains bits 15-8, the third byte contains bits 23-16, and so on. In hexadecimal notation, this converts to "0x130080".

Tip

You can convert the standard binary to hexadecimal notation. Take four binary digits at a time and convert these to their hexadecimal equivalent. For example, "0001" becomes "1", "0011" becomes "3" and so on.

Block bitmap breakdown



block number = (bit position + (byte offset * 8) + base offset)

Putting this all together, the JSON for our GetStream request looks like the following.

```
        "s" : 1,           // expected stream version
        "l" : 256,         // block size
        "f" : 1,           // source file index id
        "o" : 20,          // block offset
        "n" : 32,          // number of blocks
        "b" : "0x130080" // A missing blockId bitmap starting from the offset
    }
```

Handling errors from AWS IoT MQTT-based file delivery

An error response that is sent to a device for both `DescribeStream` and `GetStream` APIs contains a client token, an error code and an error message. A typical error response looks like the following example.

```
{  
    "o": "BlockSizeOutOfBounds",  
    "m": "The block size is out of bounds",  
    "c": "1bb8aa1-5c18-4d21-80c2-0b44fee10380"  
}
```

- "o" is the error code that indicates the reason an error occurred. Refer to the error codes later in this section for more details.
 - "m" is the error message that contains details of the error.
 - "c" is the client token field. This may be returned if it was given in the `DescribeStream` request. You can use the client token to associate the response with its request.

The client token field is not always included in an error response. When the client token given in the request isn't valid or is malformed, it's not returned in the error response.

Note

For backward compatibility, fields in the error response may be in non-abbreviated form. For example, the error code might be designated by either "code" or "o" fields and the client token field may be designated by either "clientToken" or "c" fields. We recommend that you use the abbreviation form shown above.

InvalidTopic

The MQTT topic of the stream message is invalid.

InvalidJson

The Stream request is not a valid JSON document.

InvalidCbor

The Stream request is not valid CBOR document.

InvalidRequest

The request is generally identified as malformed. For more information, see the error message.

Unauthorized

The request is not authorized to access the stream data files in the storage medium, such as Amazon S3. For more information, see the error message.

BlockSizeOutOfBounds

The block size is out of bounds. Refer to the "[MQTT-based File Delivery](#)" section in [AWS IoT Core Service Quotas](#).

OffsetOutOfBounds

The offset is out of bounds. Refer to the "[MQTT-based File Delivery](#)" section in [AWS IoT Core Service Quotas](#).

BlockCountLimitExceeded

The number of request block(s) is out of bounds. Refer to the "[MQTT-based File Delivery](#)" section in [AWS IoT Core Service Quotas](#).

BlockBitmapLimitExceeded

The size of the request bitmap is out of bounds. Refer to the "[MQTT-based File Delivery](#)" section in [AWS IoT Core Service Quotas](#).

ResourceNotFound

The requested stream, files, file versions or blocks were not found. Refer to the error message for more details.

VersionMismatch

The stream version in the request doesn't match with the stream version in the MQTT-based file delivery feature. This indicates that the stream data had been modified since the stream version was initially received by the device.

ETagMismatch

The S3 ETag in the stream doesn't match with the ETag of the latest S3 object version.

InternalError

An internal error occurred in MQTT-based file delivery.

An example use case in FreeRTOS OTA

The FreeRTOS OTA (over-the-air) agent uses AWS IoT MQTT-based file delivery to transfer FreeRTOS firmware images to FreeRTOS devices. To send the initial data set to a device, it uses the AWS IoT Job service to schedule an OTA update job to FreeRTOS devices.

For a reference implementation of an MQTT-based file delivery client, see [FreeRTOS OTA agent codes](#) in the FreeRTOS documentation.

AWS IoT Device Defender

AWS IoT Device Defender is a security service that allows you to audit the configuration of your devices, monitor connected devices to detect abnormal behavior, and mitigate security risks. It gives you the ability to enforce consistent security policies across your AWS IoT device fleet and respond quickly when devices are compromised.

IoT fleets can consist of large numbers of devices that have diverse capabilities, are long-lived, and are geographically distributed. These characteristics make fleet setup complex and error-prone. And because devices are often constrained in computational power, memory, and storage capabilities, this limits the use of encryption and other forms of security on the devices themselves. Also, devices often use software with known vulnerabilities. These factors make IoT fleets an attractive target for hackers and make it difficult to secure your device fleet on an ongoing basis.

AWS IoT Device Defender addresses these challenges by providing tools to identify security issues and deviations from best practices. AWS IoT Device Defender can audit device fleets to ensure they adhere to security best practices and detect abnormal behavior on devices.

AWS training and certification

Take the following course to get started with AWS IoT Device Defender: [AWS IoT Device Defender Primer](#).

Getting started with AWS IoT Device Defender

You can use the following tutorials to work with AWS IoT Device Defender.

Topics

- [Setting up \(p. 772\)](#)
- [Audit guide \(p. 774\)](#)
- [ML Detect guide \(p. 786\)](#)
- [Customize when and how you view AWS IoT Device Defender audit results \(p. 808\)](#)

Setting up

Before you use AWS IoT Device Defender for the first time, complete the following tasks:

- [Sign up for AWS \(p. 772\)](#)
- [Create an IAM user \(p. 773\)](#)

These tasks create an AWS account and an IAM user with administrator privileges for the account.

Sign up for AWS

When you sign up for AWS, your account is automatically signed up for all services in AWS, including AWS IoT Device Defender. If you have an AWS account already, skip to the next task. If you don't have an AWS account, use the following procedure to create one.

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open <https://portal.aws.amazon.com/billing/signup>.
2. Follow the online instructions.

Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

Note your AWS account number, because you need it for the next task.

Create an IAM user

This procedure describes how to create a IAM user for yourself and add that user to a group that has administrative permissions from an attached managed policy.

To create an administrator user for yourself and add the user to an administrators group (console)

1. Sign in to the [IAM console](#) as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

Note

We strongly recommend that you adhere to the best practice of using the **Administrator** IAM user that follows and securely lock away the root user credentials. Sign in as the root user only to perform a few [account and service management tasks](#).

2. In the navigation pane, choose **Users** and then choose **Add user**.
3. For **User name**, enter **Administrator**.
4. Select the check box next to **AWS Management Console access**. Then select **Custom password**, and then enter your new password in the text box.
5. (Optional) By default, AWS requires the new user to create a new password when first signing in. You can clear the check box next to **User must create a new password at next sign-in** to allow the new user to reset their password after they sign in.
6. Choose **Next: Permissions**.
7. Under **Set permissions**, choose **Add user to group**.
8. Choose **Create group**.
9. In the **Create group** dialog box, for **Group name** enter **Administrators**.
10. Choose **Filter policies**, and then select **AWS managed - job function** to filter the table contents.
11. In the policy list, select the check box for **AdministratorAccess**. Then choose **Create group**.

Note

You must activate IAM user and role access to Billing before you can use the **AdministratorAccess** permissions to access the AWS Billing and Cost Management console. To do this, follow the instructions in [step 1 of the tutorial about delegating access to the billing console](#).

12. Back in the list of groups, select the check box for your new group. Choose **Refresh** if necessary to see the group in the list.
13. Choose **Next: Tags**.
14. (Optional) Add metadata to the user by attaching tags as key-value pairs. For more information about using tags in IAM, see [Tagging IAM entities](#) in the [IAM User Guide](#).
15. Choose **Next: Review** to see the list of group memberships to be added to the new user. When you are ready to proceed, choose **Create user**.

You can use this same process to create more groups and users and to give your users access to your AWS account resources. To learn about using policies that restrict user permissions to specific AWS resources, see [Access management](#) and [Example policies](#).

Audit guide

This tutorial provides instructions on how to configure a recurring audit, setting up alarms, reviewing audit results and mitigating audit issues.

Topics

- [Prerequisites \(p. 774\)](#)
- [Enable audit checks \(p. 774\)](#)
- [View audit results \(p. 778\)](#)
- [Creating audit mitigation actions \(p. 779\)](#)
- [Apply mitigation actions to your audit findings \(p. 781\)](#)
- [Enable SNS notifications \(optional\) \(p. 782\)](#)
- [Enable logging \(optional\) \(p. 784\)](#)
- [Creating a AWS IoT Device Defender Audit IAM role \(optional\) \(p. 785\)](#)

Prerequisites

To complete this tutorial, you need the following:

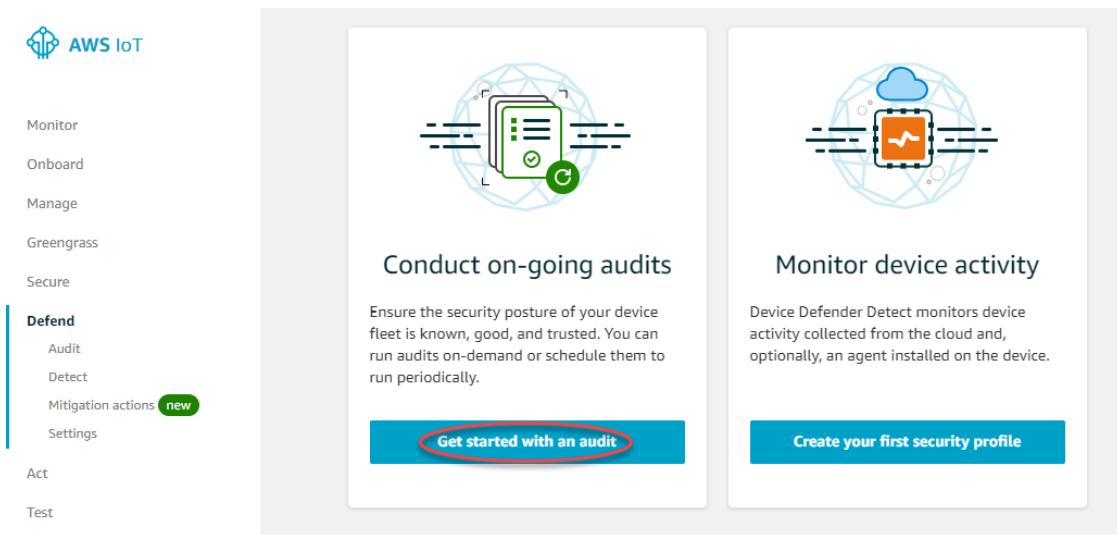
- An AWS account. If you don't have this, see [Setting up](#).

Enable audit checks

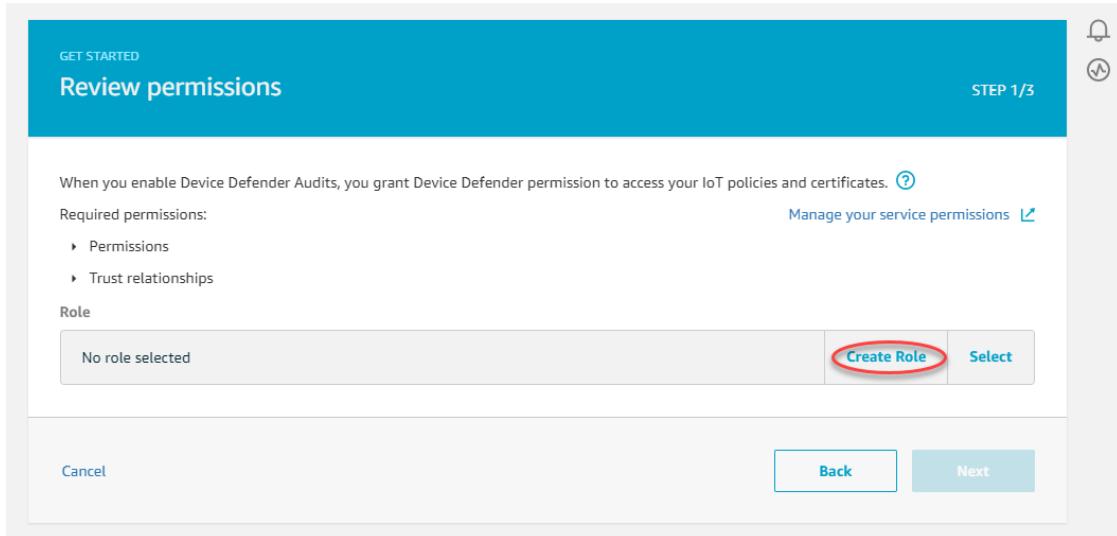
In the following procedure, you enable audit checks that look at account and device settings and policies to ensure security measures are in place. In this tutorial we instruct you to enable all audit checks, but you're able to select whichever checks you wish.

Audit pricing is per device count per month (fleet devices connected to AWS IoT). Therefore, adding or removing audit checks would not affect your monthly bill when using this feature.

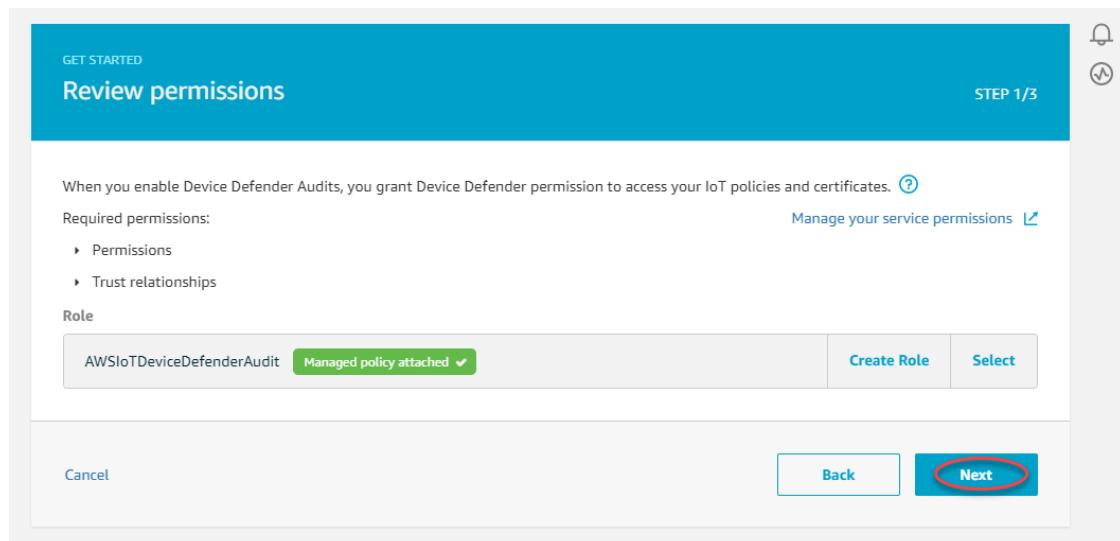
1. In the [AWS IoT console](#), in the navigation pane, expand **Defend** and select **Get started with an audit**.



2. The **Get started with Device Defender Audit** screen gives an overview of the steps required to enable the audit checks. Once you've reviewed the screen, select **Next**.
3. If you already have a role to use, you can select it. Otherwise select **Create Role** and name it **AWSIoTDeviceDefenderAudit**.



You should see the required permissions automatically attached to the role. Select the triangles next to **Permissions** and **Trust relationships** to see what permissions are granted. Select **Next** when you're ready to move on.



4. On the **Select checks** screen you will see all audit checks you can select. For this tutorial, we instruct you to select all checks but you can select whichever checks you want. Next to each audit check is a help icon that describes what the audit check does. For more information about audit checks, see [Audit Checks](#).

Select **Next** once you've selected your checks.

GET STARTED

Select checks

STEP 2/3

The checks you select here will be available when you set up audits. Data collection begins when a check has been enabled. All the free checks have been pre-selected for you. You can enable or disable checks at any time through the Device Defender Audit settings.

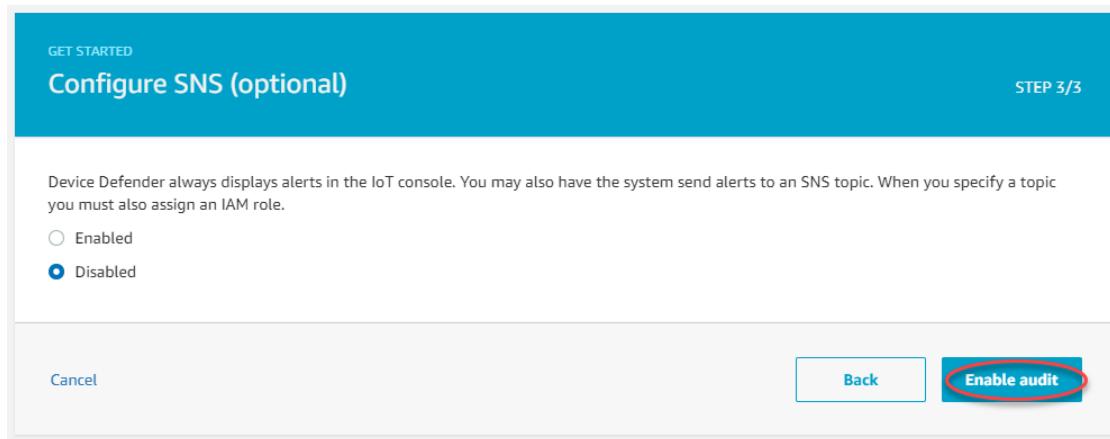
?

Check name	Severity	Resource type
<input checked="" type="checkbox"/> Authenticated Cognito role overly permissive ?	Critical	Cognito pool
<input checked="" type="checkbox"/> CA certificate key quality ?	Critical	CA certificate
<input checked="" type="checkbox"/> CA certificate revoked but device certificates still active ?	Critical	CA certificate
<input checked="" type="checkbox"/> Device Certificate key quality ?	Critical	Device certificate
<input checked="" type="checkbox"/> Device certificate shared ?	Critical	Device certificate
<input checked="" type="checkbox"/> IoT policies overly permissive ?	Critical	Policy
<input checked="" type="checkbox"/> Role Alias overly permissive ?	Critical	Role alias
<input checked="" type="checkbox"/> Unauthenticated Cognito role overly permissive ?	Critical	Cognito pool
<input checked="" type="checkbox"/> Conflicting MQTT client IDs ?	High	Client ID
<input checked="" type="checkbox"/> CA certificate expiring ?	Medium	CA certificate
<input checked="" type="checkbox"/> Device certificate expiring ?	Medium	Device certificate
<input checked="" type="checkbox"/> Revoked device certificate still active ?	Medium	Device certificate
<input checked="" type="checkbox"/> Role Alias allows access to unused services ?	Medium	Role alias
<input checked="" type="checkbox"/> Logging disabled ?	Low	Account settings

Cancel Back Next

You can always change your configured Audit checks under **Settings**.

5. On the **Configure SNS (optional)** screen, select **Enable audit**. If you'd like to enable SNS notifications, see [Enable SNS notifications \(optional\) \(p. 782\)](#).



6. You'll be redirected to **Schedules** under **Audit**.

View audit results

The following procedure shows you how to view your audit results. In this tutorial, you see the audit results from the audit checks set up in [Enable audit checks \(p. 774\)](#) tutorial.

To view audit results

1. In the [AWS IoT console](#), in the navigation pane, expand **Defend**, select **Audit**, and select **Results**.
2. The **Summary** will tell you if you have any non-compliant checks.

Name	Date	Status	Summary
On-demand	July 28, 2020, 14:14:18 (UTC-0700)	⚠️ Not compliant	1 of 14 non-compliant
On-demand	July 28, 2020, 11:55:43 (UTC-0700)	🟢 Compliant	14 of 14 completed
AWSIoTDeviceDefenderDailyAudit	July 28, 2020, 05:12:39 (UTC-0700)	⚠️ Not compliant	1 of 14 non-compliant
AWSIoTDeviceDefenderDailyAudit	July 27, 2020, 05:12:39 (UTC-0700)	⚠️ Not compliant	1 of 14 non-compliant
AWSIoTDeviceDefenderDailyAudit	July 26, 2020, 05:12:39 (UTC-0700)	⚠️ Not compliant	1 of 14 non-compliant
AWSIoTDeviceDefenderDailyAudit	July 25, 2020, 05:12:39 (UTC-0700)	⚠️ Not compliant	1 of 14 non-compliant
AWSIoTDeviceDefenderDailyAudit	July 24, 2020, 05:12:39 (UTC-0700)	⚠️ Not compliant	1 of 14 non-compliant
AWSIoTDeviceDefenderDailyAudit	July 23, 2020, 05:12:39 (UTC-0700)	⚠️ Not compliant	1 of 14 non-compliant
AWSIoTDeviceDefenderDailyAudit	July 22, 2020, 05:12:39 (UTC-0700)	⚠️ Not compliant	1 of 14 non-compliant
AWSIoTDeviceDefenderDailyAudit	July 21, 2020, 05:12:39 (UTC-0700)	⚠️ Not compliant	1 of 14 non-compliant

- Select the **Name** of the audit check you'd like to investigate.
daily_audit_check - May 14, 2020 8:51:27 AM -0700

Audit findings

Audit task ID 302ce82f9e3377e1f6be784532ff04c0 Started at May 14, 2020 8:51:27 AM -0700

▼ Non-compliant checks (2 of 14)

Check name	Severity	Non-compliant	% Resources	Mitigation
IoT policies overly permissive	Critical	1	50.0%	Use restrictive policies ⓘ
Logging disabled	Low	1	100%	Enable logging ⓘ

▶ Compliant checks (12 of 14)

▼ Mitigation actions

0 of 0

Created date	Task name	Status
You don't have any Audit action tasks yet.		

- Use the question marks for guidance on how to make your non-compliant checks compliant. For example, you can follow [Enable logging \(optional\) \(p. 784\)](#) to make the "Logging disabled" check compliant.

Creating audit mitigation actions

In the following procedure, you will create a AWS IoT Device Defender Audit Mitigation Action to enable AWS IoT logging. Each audit check has mapped mitigation actions that will affect which **Action type** you choose for the audit check you want to fix. For more information, see [Mitigation actions](#).

To use the AWS IoT console to create mitigation actions

- Open the [AWS IoT console](#).
- In the left navigation pane, choose **Defend**, and then choose **Mitigation Actions**.
- On the **Mitigation Actions** page, choose **Create**.

4. On the **Create a Mitigation Action** page, in **Action name**, enter a unique name for your mitigation action such as *EnableErrorLoggingAction*.
5. In **Action type**, choose **Enable IoT logging**.
6. In **Action execution role**, select **Create Role**. For **Name**, use *IoTMitigationActionErrorLoggingRole*. Then, choose **Create role**.
7. In **Parameters**, under **Role for logging**, select **AWSIoTLoggingRole**. For **Log level**, choose **Error**.

The screenshot shows the 'Create a new mitigation action' wizard with the following sections:

- Permissions**: Shows the 'Action execution role' dropdown set to 'IoTMitigationActionErrorLoggingRole' with 'Managed policy attached'. Buttons for 'Create Role' and 'Select' are visible.
- Parameters**: Shows the 'Role for logging' dropdown set to 'AWSIoTLoggingRole' with 'Clear' and 'Select' buttons. The 'Log level' dropdown is set to 'Error'.
- Tags**: Shows fields for 'Tag name' ('Provide a tag name, e.g. Manufacturer') and 'Value' ('Provide a tag value, e.g. Acme-Corporation'), both with 'Clear' buttons. A 'Save' button is at the bottom right.

8. Choose **Save** to save your mitigation action to your AWS account.

- Once created, you will see the following screen indication that your mitigation action was created successfully.

The screenshot shows the AWS IoT Device Defender interface. At the top, a green banner displays a checkmark icon and the text "Successfully created mitigation action". Below this, a table titled "Mitigation actions (1)" lists one item:

	Created date	Action name	ARN
<input type="radio"/>	Jun 18, 2020 8:30:07 AM -0700	EnableErrorLoggingAction	arn:aws:iot:us-east-1:765219403047:mitigationaction/EnableErrorLoggingAction

Apply mitigation actions to your audit findings

The following procedure shows you how to apply mitigation actions to your audit results.

To mitigate non-compliant audit findings

- Open the [AWS IoT console](#).
- In the left navigation pane, choose **Audit**, and then choose **Results**. Select the name of the audit that you want to respond to.
- Check your results. Notice that **Logging disabled** is located under **Non-compliant checks**.
- Select **Start mitigation actions**.

The screenshot shows the AWS IoT Audit Results page for the audit "AWSIoTDeviceDefenderDailyAudit" from Jun 16, 2020, at 11:23:17 PM -0700. A red oval highlights the "Start mitigation actions" button.

Audit findings

Audit task ID 08a38dcf887736bd947a95478646a3f1 Started at Jun 16, 2020 11:23:17 PM -0700

Non-compliant checks (2 of 14)

Check name	Severity	Non-compliant	% Resources	Mitigation
IoT policies overly permissive	Critical	1	50.0%	Use restrictive p... ?
Logging disabled	Low	1	100%	Enable logging ?

Compliant checks (12 of 14)

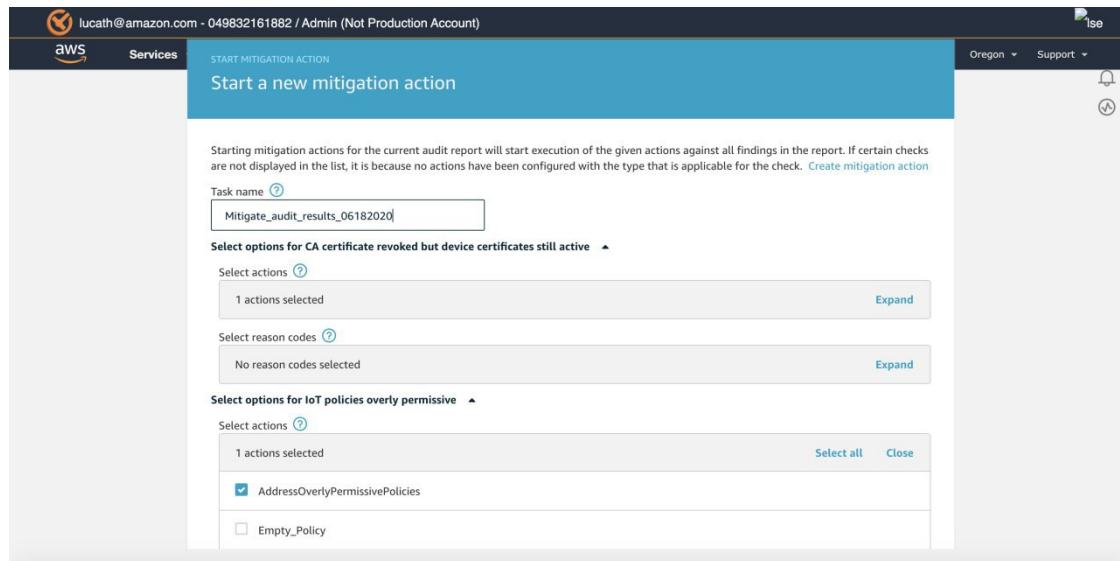
Mitigation actions

0 of 0

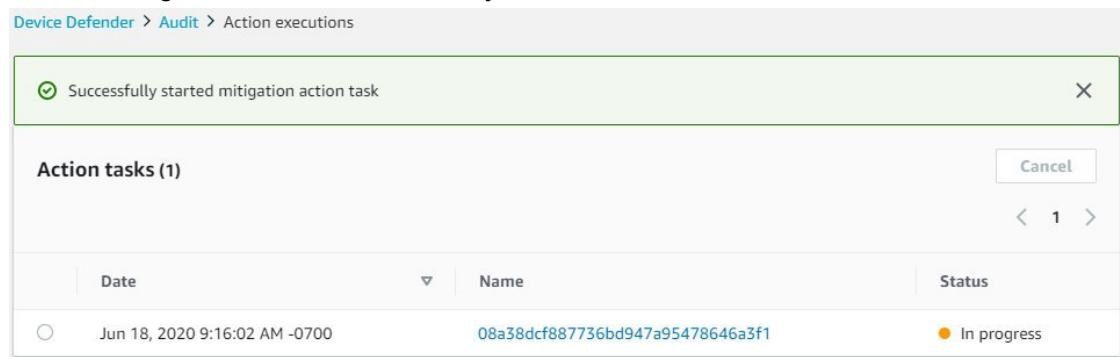
Created date	Task name	Status
--------------	-----------	--------

You don't have any Audit action tasks yet.

- Under **Select actions**, select the appropriate actions for each non-compliant finding to address the issues.



- Select Confirm.**
- Once the mitigation action is started, it may take a few minutes for it to run.



To check that the mitigation action worked

- In the AWS console, in the navigation pane, select **Settings**.
- Confirm that **Logs** are **Enabled** and the **Level of verbosity** is **Error**.

Enable SNS notifications (optional)

In the following procedure, you enable Simple Notifications Service (SNS) notifications to alert you when your audits identifies any non-compliant resources. In this tutorial you will set up notifications for the audit checks enabled in the [Enable audit checks \(p. 774\)](#) tutorial.

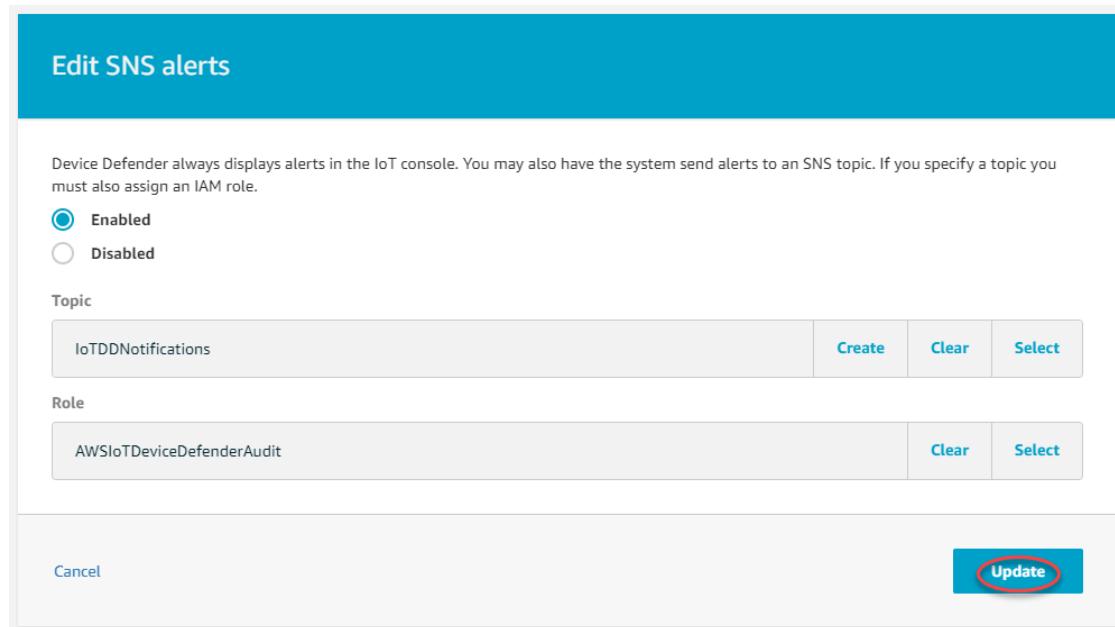
- First, you need to create an IAM policy that provides access to Amazon SNS via the AWS Management Console. You can do this by following the [Creating a AWS IoT Device Defender Audit IAM role \(optional\) \(p. 785\)](#) process, but selecting **AWSIoTDeviceDefenderPublishFindingsToSNSSMitigationAction** in step 8.
- In the [AWS IoT console](#), in the navigation pane, expand **Defend** and select **Settings**.
- Under **SNS alerts**, select **Edit**.

The screenshot shows the AWS IoT Device Defender Audit guide. On the left, a sidebar has 'Defend' selected, with 'Audit' and 'Mitigation actions (new)' listed. Under 'Audit', 'Logging disabled' is checked. A large blue button labeled 'Disable' is present. Below it, a section titled 'Disabled Audit checks' states 'All Audit checks are currently enabled'. At the bottom, a section titled 'SNS alerts' says 'SNS alerts are not configured' and features a blue 'Edit' button.

- On the **Edit SNS alerts** screen, select **Enabled**. Under **Topic**, select **Create**. Name the topic **IoTDDNotifications** and select **Create**. Under **Role**, select the role you created called **AWSIoTDeviceDefenderAudit**.

The screenshot shows the 'Edit SNS alerts' configuration screen. It includes fields for 'Enabled' (radio button selected), 'Topic' (button for 'No topic selected' with 'Create' and 'Select' buttons), 'Role' (button for 'No role selected' with 'Select' button), and a 'Cancel' button with an 'Update' button at the bottom right.

Select **Update**.



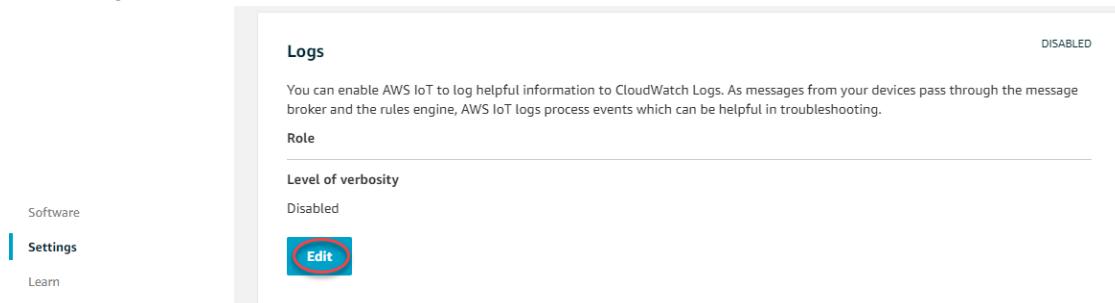
If you'd like to receive email or text in your Ops platforms through SNS, see [Using Amazon SNS for user notifications](#).

Enable logging (optional)

This procedure describes how to enable AWS IoT to log information to CloudWatch Logs. This will allow you to view your audit results. Enabling logging may result in incurred charges.

To enable logging

1. In the AWS console, in the navigation pane, select **Settings**.
2. Under **Logs**, select **Edit**.



3. Under **Level of verbosity**, select **Debug (most verbose)**.
4. Under **Set role**, select **Create Role** and name the role **AWSIoTLoggingRole**. A policy will automatically be attached.

Configure role setting

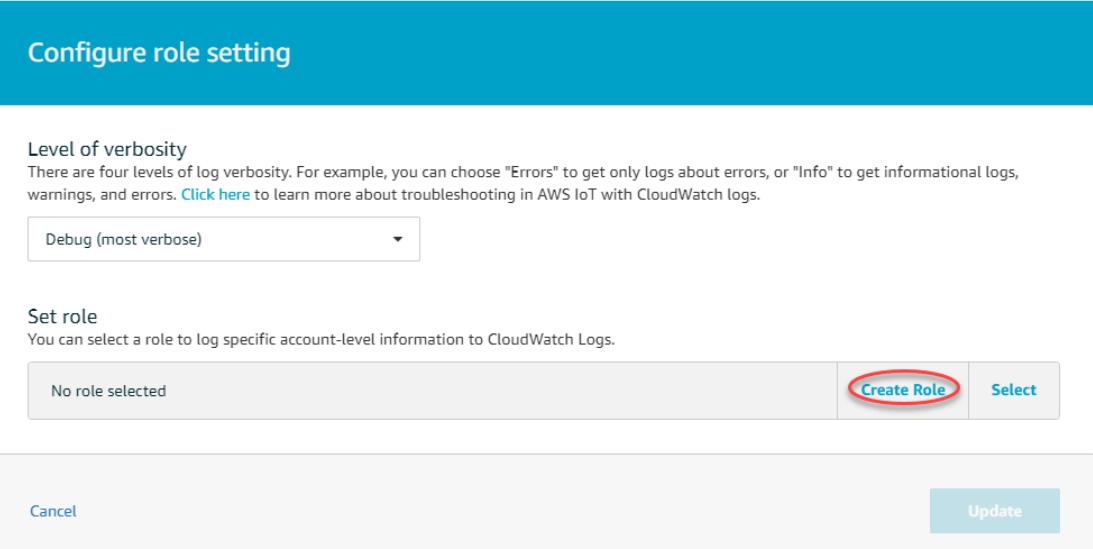
Level of verbosity
There are four levels of log verbosity. For example, you can choose "Errors" to get only logs about errors, or "Info" to get informational logs, warnings, and errors. [Click here](#) to learn more about troubleshooting in AWS IoT with CloudWatch logs.

Debug (most verbose) ▾

Set role
You can select a role to log specific account-level information to CloudWatch Logs.

No role selected [Create Role](#) [Select](#)

[Cancel](#) [Update](#)



Select **Update**.

Configure role setting

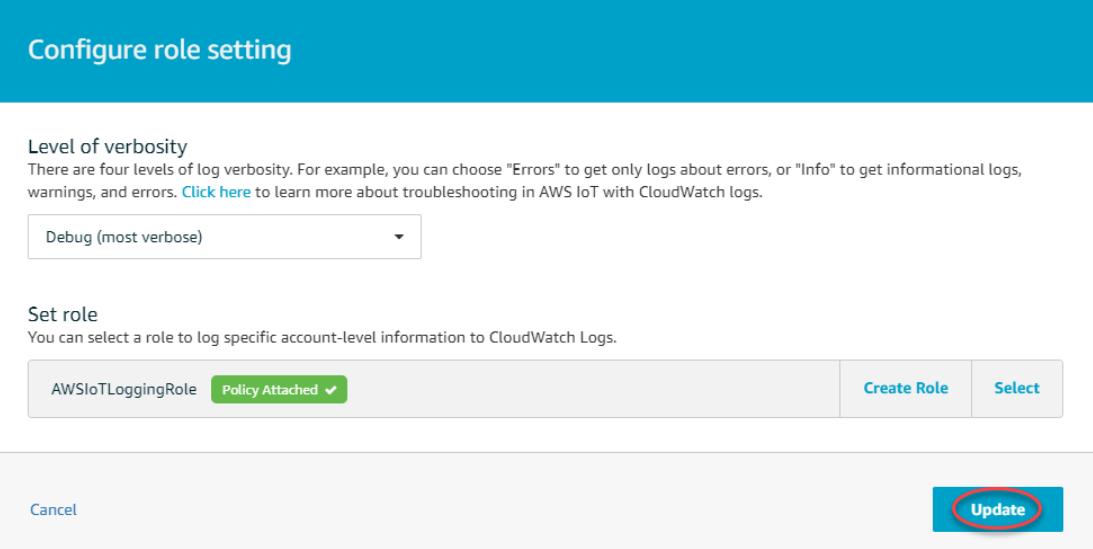
Level of verbosity
There are four levels of log verbosity. For example, you can choose "Errors" to get only logs about errors, or "Info" to get informational logs, warnings, and errors. [Click here](#) to learn more about troubleshooting in AWS IoT with CloudWatch logs.

Debug (most verbose) ▾

Set role
You can select a role to log specific account-level information to CloudWatch Logs.

AWSIoTLoggingRole Policy Attached ▾ [Create Role](#) [Select](#)

[Cancel](#) [Update](#)



Creating a AWS IoT Device Defender Audit IAM role (optional)

In the following procedure, you create a AWS IoT Device Defender Audit IAM role that provides AWS IoT Device Defender read access to AWS IoT.

1. Navigate to the IAM console at <https://console.aws.amazon.com/iam/>
2. In the navigation pane, chose **Users** and then choose **Add user**.
3. For **User name**, enter **Administrator**.
4. Select the check box next to **AWS Management Console access**. Then select **Custom password**, and then enter your new password in the text box.

5. (Optional) By default, AWS requires the new user to create a new password when first signing in. You can clear the check box next to **User must create a new password at next sign-in** to allow the new user to reset their password after they sign in.
6. Choose **Next: Permissions**.
7. Under **Set permissions**, choose **Attach existing policies directly**.
8. In the policy list, select the check box for **AWSIoTDeviceDefenderAudit**.
9. Choose **Next: Tags**.
10. Choose **Next: Review** to see the list of group memberships to be added to the new user. When you are ready to proceed, choose **Create user**.

ML Detect guide

In this Getting Started guide, you create an ML Detect Security Profile that uses machine learning (ML) to create models of expected behavior based on historical metric data from your devices. While ML Detect is creating the ML model, you can monitor its progress. After the ML model is built, you can view and investigate alarms on an ongoing basis and mitigate identified issues.

For more information about ML Detect and its API and CLI commands, see [ML Detect \(p. 888\)](#).

This chapter contains the following sections:

- [Prerequisites \(p. 786\)](#)
- [How to use ML Detect in the console \(p. 786\)](#)
- [How to use ML Detect with the CLI \(p. 799\)](#)

Prerequisites

- An AWS account. If you don't have this, see [Setting up](#).

How to use ML Detect in the console

Tutorials

- [Enable ML Detect \(p. 786\)](#)
- [Monitor your ML model status \(p. 790\)](#)
- [Review your ML Detect alarms \(p. 791\)](#)
- [Fine-tune your ML alarms \(p. 793\)](#)
- [Mark your alarm's verification state \(p. 794\)](#)
- [Mitigate identified device issues \(p. 795\)](#)

Enable ML Detect

The following procedures detail how to set up ML Detect in the console.

1. First, make sure your devices will create the minimum datapoints required as defined in [ML Detect minimum requirements \(p. 889\)](#) for ongoing training and refreshing of the model. For data collection to progress, ensure your Security Profile is attached to a target, which can be a thing or thing group.
2. In the [AWS IoT console](#), in the navigation pane, expand **Defend**. Choose **Detect**, **Security profiles**, **Create security profile**, and then **Create ML anomaly Detect profile**.
3. On the **Set basic configurations** page, do the following.

- Under **Target**, choose your target device groups.
- Under **Security profile name**, enter a name for your Security Profile.
- (Optional) Under **Description** you can write in a short description for the ML profile.
- Under **Selected metric behaviors in Security Profile**, choose the metrics you'd like to monitor.

Metric	Type	ML Detect confidence	Datapoints required to trigger alarm	Datapoints required to clear alarm	Notifications
Authorization failures	Cloud-side	High	1	1	Suppressed
Connection attempts	Cloud-side	High	1	1	Suppressed
Disconnects	Cloud-side	High	1	1	Suppressed
Message size	Cloud-side	High	1	1	Suppressed
Messages received	Cloud-side	High	1	1	Suppressed
Messages sent	Cloud-side	High	1	1	Suppressed

When you're done, choose **Next**.

4. On the **Set SNS (optional)** page, specify an SNS topic for alarm notifications when a device violates a behavior in your profile. Choose an IAM role you will use to publish to the selected SNS topic.

If you don't have an SNS role yet, use the following steps to create a role with the proper permissions and trust relationships required.

- Navigate to the [IAM console](#). In the navigation pane, choose **Roles** and then choose **Create role**.
- Under **Select type of trusted entity**, select **AWS Service**. Then, under **Choose a use case**, choose **IoT** and under **Select your use case**, choose **IoT - Device Defender Mitigation Actions**. When you're done, choose **Next: Permissions**.

- Under **Attached permissions policies**, ensure that **AWSIoTDeviceDefenderPublishFindingsToSNSSmitigationAction** is selected, and then choose **Next: Tags**.

Create role

1 2 3 4

Attached permissions policies

The type of role that you selected requires the following policy.

Filter policies ▼		Search	Showing 6 results
Policy name	Used as	Description	
▶ AWSIoTDeviceDefenderAddThingsToThingGrou...	Permissions policy (1)	Provides write access to IoT thing groups and r...	
▶ AWSIoTDeviceDefenderEnableIoTLoggingMitig...	Permissions policy (2)	Provides access for enabling IoT logging for ex...	
▶ AWSIoTDeviceDefenderPublishFindingsToSNS...	None	Provides messages publish access to SNS topi...	
▶ AWSIoTDeviceDefenderReplaceDefaultPolicyMi...	None	Provides write access to IoT policies for execut...	
▶ AWSIoTDeviceDefenderUpdateCACertMitigatio...	None	Provides write access to IoT CA certificates for ...	
▶ AWSIoTDeviceDefenderUpdateDeviceCertMitig...	None	Provides write access to IoT certificates for exe...	

Set permissions boundary

* Required Cancel Previous **Next: Tags**

- Under **Add tags (optional)**, you can add any tags you'd like to associate with your role. When you're done, choose **Next: Review**.
- Under **Review**, give your role a name and ensure that **AWSIoTDeviceDefenderPublishFindingsToSNSSmitigationAction** is listed under **Permissions** and **AWS service: iot.amazonaws.com** is listed under **Trust relationships**. When you're done, choose **Create role**.

Identity and Access Management (IAM)

Roles > Sample-SNS-role

Summary

Role ARN: arn:aws:iam::049832161882:role/Sample-SNS-role

Role description: Provides AWS IoT Device Defender write access to publish SNS notifications | Edit

Instance Profile ARNs:

Path: /

Creation time: 2020-12-21 17:13 PST

Last activity: Not accessed in the tracking period

Maximum session duration: 1 hour | Edit

Permissions **Trust relationships** **Tags** **Access Advisor** **Revoke sessions**

▶ Permissions policies (1 policy applied)

Attach policies **Add inline policy**

Policy name	Policy type
▶ AWSIoTDeviceDefenderPublishFindingsToSNSSmitigationAction	AWS managed policy

▶ Permissions boundary (not set)

The screenshot shows the AWS Identity and Access Management (IAM) service. On the left, there's a navigation menu with options like Dashboard, Access management, Roles (which is selected), Policies, Identity providers, Account settings, Access reports, Access analyzer, Archive rules, Analyzers, Settings, Credential report, Organization activity, and Service control policies (SCPs). Below the menu is a search bar labeled 'Search IAM'. The main area is titled 'Roles > Sample-SNS-role' and has a 'Summary' tab selected. It displays the following details:

- Role ARN:** arn:aws:iam::049832161882:role/Sample-SNS-role
- Role description:** Provides AWS IoT Device Defender write access to publish SNS notifications | Edit
- Instance Profile ARNs:** (empty)
- Path:** /
- Creation time:** 2020-12-21 17:13 PST
- Last activity:** Not accessed in the tracking period
- Maximum session duration:** 1 hour | Edit

Below these details are tabs for 'Permissions', 'Trust relationships' (which is selected), 'Tags', 'Access Advisor', and 'Revoke sessions'. A note says 'You can view the trusted entities that can assume the role and the access conditions for the role.' followed by a link 'Show policy document'. Under 'Edit trust relationship', it says 'Trusted entities' and lists 'The following trusted entities can assume this role.' There is also a section for 'Conditions' which states 'There are no conditions associated with this role.'

5. On the **Edit Metric behavior** page, you can customize your ML behavior settings.

The screenshot shows the 'Edit metric behaviors - optional' page. At the top, there's a breadcrumb trail: AWS IoT > Device Defender > Detect > Security Profiles > Create ML Security Profile. Below the breadcrumb, there are three steps: Step 1 (Set basic configurations), Step 2 - optional (Edit metric behaviors, which is selected), and Step 3 (Review configuration). The main content area is titled 'Edit metric behaviors' and contains three sections:

- Authorization failures:**
 - Behavior name: Authorization_failures_ML_behavior
 - Metric: Authorization failures
 - Datapoints required to trigger alarm: 1
 - Datapoints required to clear alarm: 1
 - Notifications: Suppressed
 - ML Detect confidence: High
- Bytes in:**
 - Behavior name: Bytes_in_ML_behavior
 - Metric: Bytes in
 - Datapoints required to trigger alarm: 1
 - Datapoints required to clear alarm: 1
 - Notifications: Suppressed
 - ML Detect confidence: High
- Connection attempts:**
 - Behavior name: Connection_attempts_ML_behavior
 - Metric: Connection attempts
 - Datapoints required to trigger alarm: 1
 - Datapoints required to clear alarm: 1
 - Notifications: Suppressed
 - ML Detect confidence: High

6. When you're done, choose **Next**.
7. On the **Review configuration** page, verify the behaviors you'd like machine learning to monitor, and then choose **Next**.

Behavior name	Metric	Type	ML Detect confidence	Datapoints required to trigger alarm	Datapoints required to clear alarm	No
Authorization failures_ML_behavior	Authorization failures	Cloud-side	High	1	1	Sup
Bytes_out_ML_behavior	Bytes out	Device-side	High	1	1	Sup
Connection_attempts_ML_behavior	Connection attempts	Cloud-side	High	1	1	Sup
Disconnects_ML_behavior	Disconnects	Cloud-side	High	1	1	Sup

- After you've created your Security Profile, you're redirected to the **Security Profiles** page, where the newly created Security Profile appears.

Note

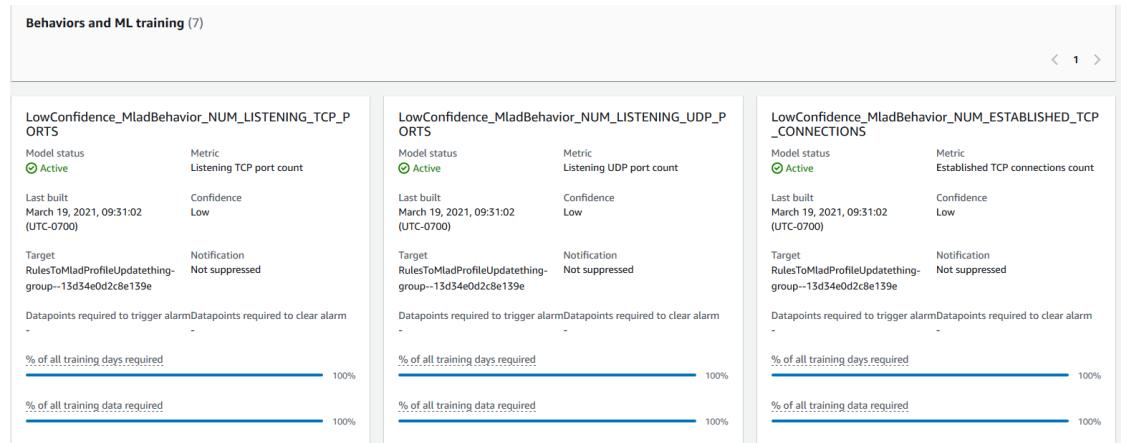
The initial ML model training and creation takes 14 days to complete. You can expect to see alarms after it's complete, if there is any anomalous activity on your devices.

Monitor your ML model status

While your ML models are in the initial training period, you can monitor their progress at any time by taking the following steps.

- In the [AWS IoT console](#), in the navigation pane, expand **Defend**, and then choose **Detect, Security profiles**.
- On the **Security Profiles** page, choose the Security Profile you'd like to review. Then, choose **Behaviors and ML training**.
- On the **Behaviors and ML training** page, check the training progress of your ML models.

After your model status is **Active**, it'll start making Detect decisions based on your usage and update the profile every day.



Note

If your model doesn't progress as expected, make sure your devices are meeting the [Minimum requirements \(p. 889\)](#).

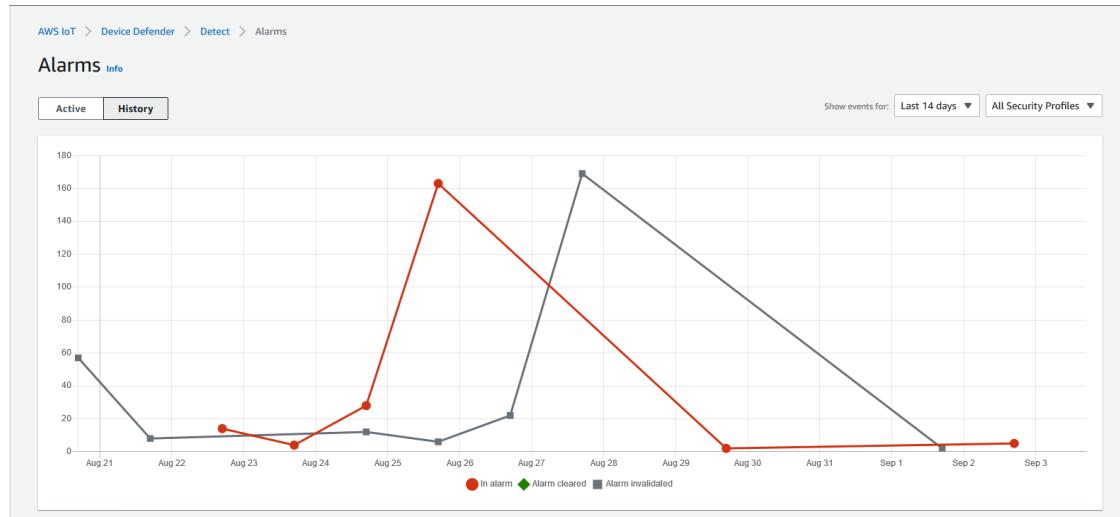
Review your ML Detect alarms

After your ML models are built and ready for data inference, you can regularly view and investigate alarms that are identified by the models.

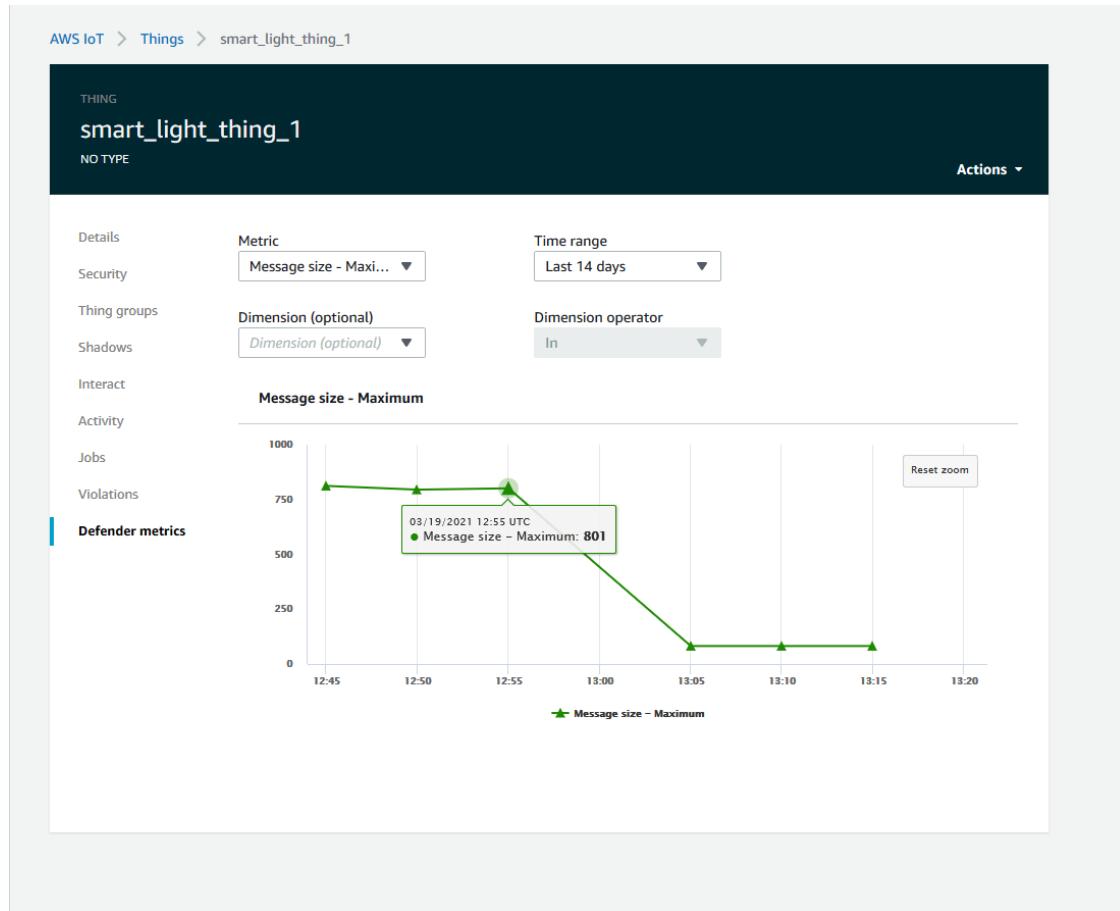
1. In the [AWS IoT console](#), in the navigation pane, expand **Defend**, and then choose **Detect Alarms**.

First event	Thing name	Security Profile	Behavior type	Behavior name	Last emitted	Verification state	Confidence
September 03, 2021, 15:50:00 (UTC-0700)	iotconsole-6f8379bc-c245-4ffe-8ef7-b2b52e78975c	fdsa	Rule-based	Authorization_failures_behavior (Notification: on)	Authorization failures: 0 failure(s)	Unknown	-
September 03, 2021, 15:50:00 (UTC-0700)	iotconsole-539d0ef0-3504-4a9c-a7a1-be53b472b850	fdsa	Rule-based	Authorization_failures_behavior (Notification: on)	Authorization failures: 0 failure(s)	Unknown	-
September 03, 2021, 15:50:00 (UTC-0700)	iotconsole-81fc61d7-9362-4c87-ad6-333891ff7349	fdsa	Rule-based	Authorization_failures_behavior (Notification: on)	Authorization failures: 0 failure(s)	Unknown	-
September 03, 2021, 15:50:00 (UTC-0700)	iotconsole-23e8ec9a-2162-456e-a8c2-302c3826f618	fdsa	Rule-based	Authorization_failures_behavior (Notification: on)	Authorization failures: 0 failure(s)	Unknown	-
September 03, 2021, 15:50:00 (UTC-0700)	iotconsole-85e0278e-29aa-4554-b3b6-111b9063228f	fdsa	Rule-based	Authorization_failures_behavior (Notification: on)	Authorization failures: 0 failure(s)	Unknown	-

2. If you navigate to the **History** tab, you can also view details about your devices that are no longer in alarms.



To get more information, under **Manage** choose **Things**, chose the thing you'd like to see more details for, and then navigate to **Defender metrics**. You can access the **Defender metrics graph** and perform your investigation on anything in alarm from the **Active** tab. In this case, the graph shows a spike in message size, which triggered the alarm. You can see the alarm subsequently cleared.



Fine-tune your ML alarms

After your ML models are built and ready for data evaluations, you can update your Security Profile's ML behavior settings to change the configuration. The following procedure shows you how to update your Security Profile's ML behavior settings in the AWS CLI.

1. In the [AWS IoT console](#), in the navigation pane, expand **Defend**, and then choose **Detect, Security profiles**.
2. On the **Security Profiles** page, select the check box next to the Security Profile you'd like to review. Then, choose **Actions, Edit**.

The screenshot shows the AWS IoT Device Defender Security Profiles page. The left sidebar lists various AWS services like Types, Thing groups, Billing groups, Jobs, Tunnels, Fleet Hub, Greengrass, Wireless connectivity, and Secure. The main content area shows a table titled "Security Profiles (30+)" with two entries:

Security Profile	Threshold type	Behaviors	Metrics retained	Target	Creation date	Notifications
Smart_lights_ML_Detect_Security_Profile	ML	9	-	All registered things	March 17, 2021, 12:58:14 (UTC-0700)	Suppressed (9)
MyEmptyGroupSP	ML	6	-	EmptyGroup	March 16, 2021, 17:52:01 (UTC-0700)	Suppressed (6)

3. Under **Set basic configurations**, you can adjust Security Profile target thing groups or change what metrics you want to monitor.

The screenshot shows the "Create ML Security Profile" wizard, Step 1: Set basic configurations. The left sidebar has three steps: Step 1 (Set basic configurations), Step 2 - optional (Edit metric behaviors), and Step 3 (Review configuration). The main content area is titled "Set basic configurations" and includes the following fields:

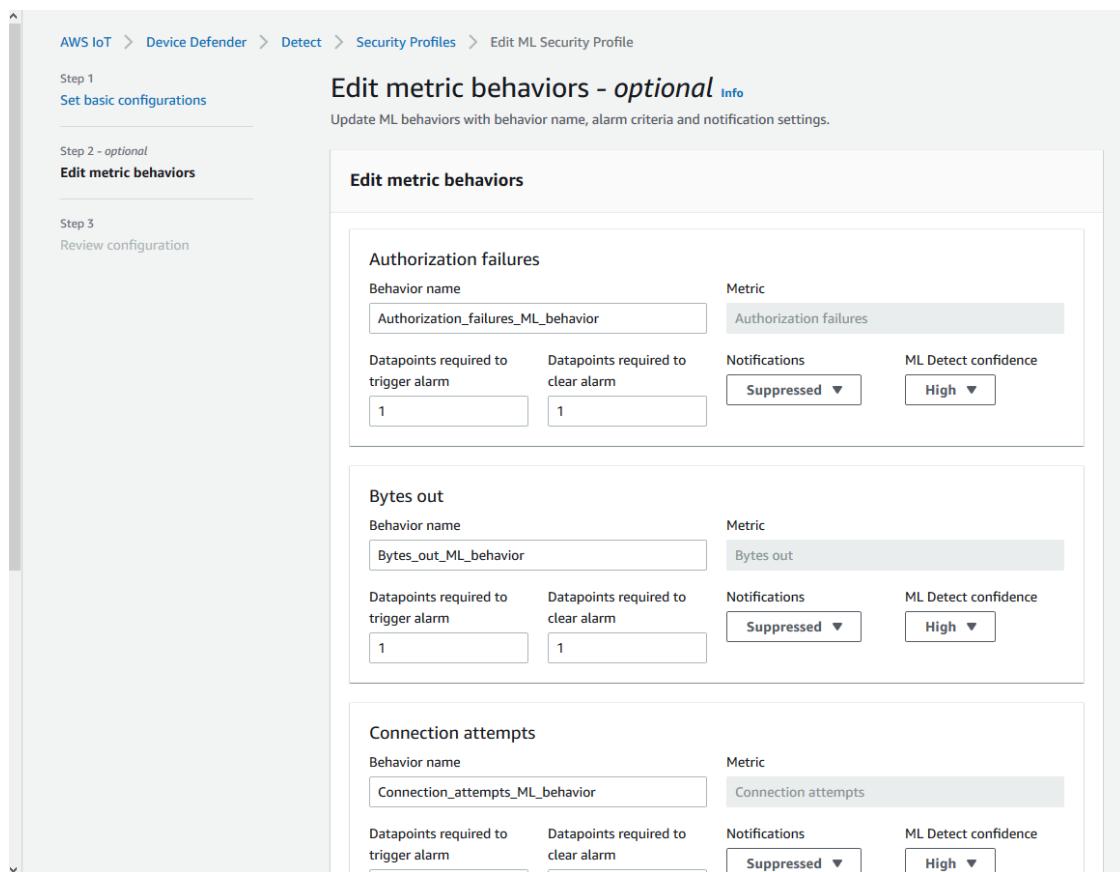
- Target:** A dropdown menu showing "All registered things" which is selected.
- Security Profile name:** A text input field containing "Smart_lights_ML_Detect_Security_Profile". Below it is a note: "Enter a unique name containing only letters, numbers, hyphens, colon, or underscores. A Security Profile name cannot contain any spaces."
- Description - optional:** A text input field containing "ML Detect security profile for monitoring smart lights".

Below this is a section titled "Selected metric behaviors in Security Profile (6)" with a note: "You can assess how your fleet of devices is operating across the following metric behaviors." It includes buttons for "Delete", "Add cloud-side metric", and "Add device-side metric". The table below lists six metric behaviors:

	Metric	Type	ML Detect confidence	Datapoints required to trigger alarm	Datapoints required to clear alarm	Notifications
<input type="checkbox"/>	Authorization failures	Cloud-side	High	1	1	Suppressed
<input type="checkbox"/>	Connection attempts	Cloud-side	High	1	1	Suppressed
<input type="checkbox"/>	Disconnects	Cloud-side	High	1	1	Suppressed
<input type="checkbox"/>	Message size	Cloud-side	High	1	1	Suppressed
<input type="checkbox"/>	Messages received	Cloud-side	High	1	1	Suppressed
<input type="checkbox"/>	Messages sent	Cloud-side	High	1	1	Suppressed

4. You can update any of the following by navigating to **Edit metric behaviors**.

- Your ML model datapoints required to trigger alarm
- Your ML model datapoints required to clear alarm
- Your ML Detect confidence level
- Your ML Detect notifications (for example, **Not suppressed**, **Suppressed**)



Mark your alarm's verification state

Mark your alarms by setting the verification state and providing a description of that verification state. This helps you and your team identify alarms that you don't have to respond to.

1. In the [AWS IoT console](#), on the navigation pane, expand **Defend**, and then choose **Detect**, **Alarms**. Select an alarm to mark its verification state.

The screenshot shows the AWS IoT Device Defender Detect Alarms interface. At the top, there are tabs for Active and History. Below is a table titled 'All alarms (1/5) Info' with columns: First event, Thing name, Security Profile, Behavior type, Behavior name, Last emitted, Verification state, and Confidence. One row is selected, showing details: September 03, 2021, at 15:50:00 UTC-0700, for thing 'iotconsole-6f8379bc-c245-4ffe-8ef7-b2b52e789750'. The behavior is Rule-based, named 'Authorization_failures_behavior [Notification: on].....'. The last emitted time was Authorization failures: 0 failure(s) at 15:50:00 UTC-0700. The verification state is Unknown. A 'Mark verification state' button is visible at the top right.

2. Choose **Mark verification state**. The verification state modal opens.
3. Choose the appropriate verification state, enter a verification description (optional), and then choose **Mark**. This action assigns a verification state and description to the chosen alarm.

The screenshot shows the 'Mark verification state' modal window overlaid on the alarms list. The modal has a title 'Mark verification state' and a sub-section 'Select verification state'. It contains a list of options: Unknown, True positive, False positive, Benign positive, and Unknown (which is currently selected). Below the list is a note about marking verification state to help improve ML and Rules Detect features. At the bottom of the modal are 'Cancel' and 'Mark' buttons. The background shows the same list of alarms as the previous screenshot.

Mitigate identified device issues

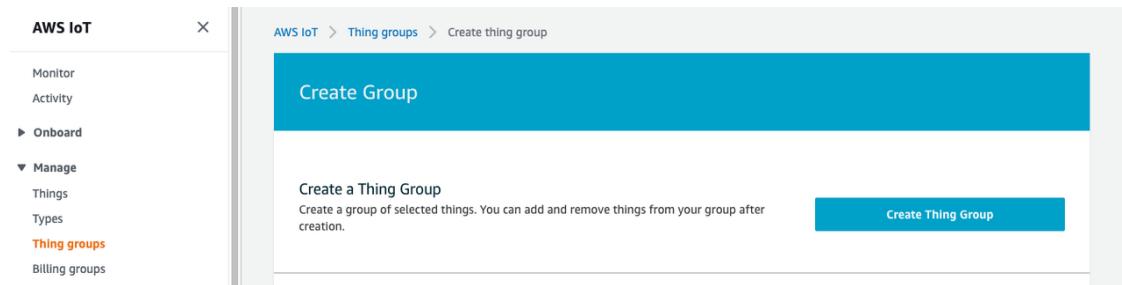
1. (*Optional*) Before setting up quarantine mitigation actions, let's set up a quarantine group where we'll move the device that's in violation to. You can also use an existing group.
2. Navigate to **Manage, Thing groups**, and then **Create Thing Group**. Name your thing group. For this tutorial, we'll name our thing group **Quarantine_group**. Under **Thing group, Security**, apply the following policy to the thing group.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Action": "iot:Connect"
    }
  ]
}
```

```

        "Action": "iot:*",
        "Resource": "*",
    }
]
}

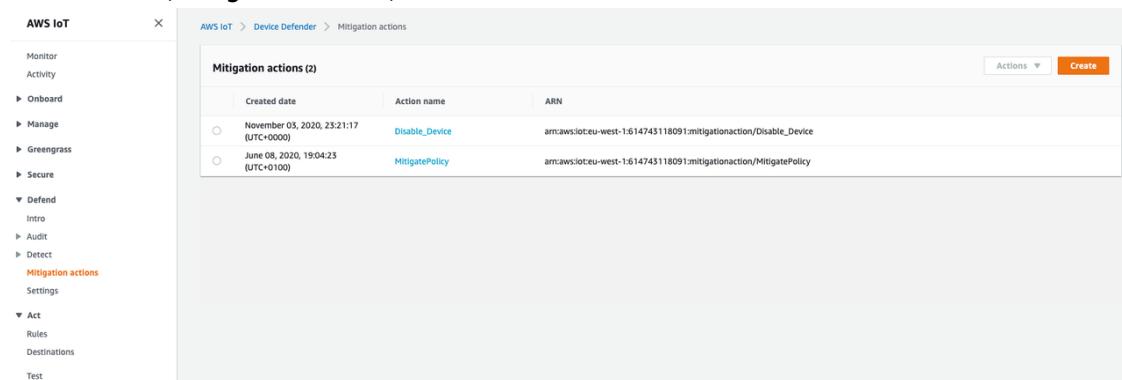
```



When you're done, choose **Create thing group**.

- Now that we've created a thing group, let's create a mitigation action that move devices that in alarm into the `Quarantine_group`.

Under **Defend, Mitigation actions**, choose **Create**.



- On the **Create a new mitigation action** page, enter the following information.
 - Action name:** Give your mitigation action a name, such as `Quarantine_action`.
 - Action type:** Choose the type of action. We'll choose **Add things to thing group (Audit or Detect mitigation)**.
 - Action execution role:** Create a role or choose an existing role if you created one earlier.
 - Parameters:** Choose a thing group. We can use `Quarantine_group`, which we created earlier.

Create a new mitigation action

You can use AWS IoT Device Defender to mitigate issues that were found during and audit or ongoing detect monitoring. There are predefined actions for the different audit checks and detect alarms to help you resolve issues quickly.

Action name [Info](#)

Action type [Info](#)

Permissions

Please create or select a role with the following mitigation action type specific permission(s) and trust relationship.

Required permissions: [Manage your service permissions ↗](#)

- ▶ Permissions
- ▶ Trust relationships

You can also attach an action specific managed policy to an existing role, or create a new role with the required managed policy attached.

Action execution role [Info](#)
 [Create Role](#) [Select](#)

Parameters

Thing groups [Info](#)

1 thing group(s) selected. [Close](#)

[Thing groups](#) [Summary](#)

Quarantine_group

When you're done, choose **Save**. You now have a mitigation action that moves devices in alarm to a quarantine thing group, and a mitigation action to isolate the device while you investigate.

5. Navigate to **Defender, Detect, Alarms**. You can see which devices are in alarm state under **Active**.

First event	Thing name	Security Profile	Behavior type	Behavior name	Last emitted	Verification state	Confidence
September 03, 2021, 15:50:00 (UTC-0700)	iotconsole-6f8379bc-c245-4ffe-8ef7-b2b52e78975c	fdsa	Rule-based	Authorization_failures_behavior (Notification: on).....	Authorization failures: 0 failure(s)	Unknown	-
September 03, 2021, 15:50:00 (UTC-0700)	iotconsole-539d0ef0-3504-4a9c-a7a1-be53b472b850	fdsa	Rule-based	Authorization_failures_behavior (Notification: on).....	Authorization failures: 0 failure(s)	Unknown	-
September 03, 2021, 15:50:00 (UTC-0700)	iotconsole-01fc61d7-9362-4c87-ad6-333891ff7349	fdsa	Rule-based	Authorization_failures_behavior (Notification: on).....	Authorization failures: 0 failure(s)	Unknown	-
September 03, 2021, 15:50:00 (UTC-0700)	iotconsole-23e8ec9a-2162-456e-a8c2-302c3826f618	fdsa	Rule-based	Authorization_failures_behavior (Notification: on).....	Authorization failures: 0 failure(s)	Unknown	-
September 03, 2021, 15:50:00 (UTC-0700)	iotconsole-85e0278e-29aa-4554-b3b6-11b906323bf	fdsa	Rule-based	Authorization_failures_behavior (Notification: on).....	Authorization failures: 0 failure(s)	Unknown	-

Select the device you want to move to the quarantine group and choose **Start Mitigation Actions**.

- Under **Start mitigation actions**, **Start Actions** select the mitigation action you created earlier. For example, we'll choose **Quarantine_action**, then choose **Start**. The Action Tasks page opens.

Start mitigation actions

Select actions for mitigation.

Things effected by the selected alarm(s)
ddml7

Select Actions
The sequence of action executions follows the order of selected action(s)

Choose actions(s) to execute

Quarantine_action

I understand that the selected mitigation action(s) may not be reversible.

Cancel **Start**

- The device is now isolated in **Quarantine_group** and you can investigate the root cause of the issue that set off the alarm. After you complete the investigation, you can move the device out of the thing group or take further actions.

Action tasks (1)						
Date	Task ID	Action name	Action type	Action parameter (1)	Action parameter (2)	Action Executions
December 02, 2020, 14:19:57 (UTCZ)	73fad2ea-9bd8-48d0-af3a-3dbc120b91e7	Quarantine_action	Add things to thing group	Thing group(s): Quarantine_group	Override dynamic groups: false	Successful

How to use ML Detect with the CLI

The following shows you how to set up ML Detect using the CLI.

Tutorials

- [Enable ML Detect \(p. 799\)](#)
- [Monitor your ML model status \(p. 800\)](#)
- [Review your ML Detect alarms \(p. 802\)](#)
- [Fine-tune your ML alarms \(p. 803\)](#)
- [Mark your alarm's verification state \(p. 805\)](#)
- [Mitigate identified device issues \(p. 805\)](#)

Enable ML Detect

The following procedure shows you how to enable ML Detect in the AWS CLI.

1. Make sure your devices will create the minimum datapoints required as defined in [ML Detect minimum requirements \(p. 889\)](#) for ongoing training and refreshing of the model. For data collection to progress, ensure your things are in a thing group attached to a Security Profile.
2. Create an ML Detect Security Profile by using the `create-security-profile` command. The following example creates a Security Profile named `security-profile-for-smart-lights` that checks for number of messages sent, number of authorization failures, number of connection attempts, and number of disconnects. The example uses `mlDetectionConfig` to establish that the metric will use the ML Detect model.

```
aws iot create-security-profile \
  --security-profile-name security-profile-for-smart-lights \
  --behaviors \
  '[{
    "name": "num-messages-sent-ml-behavior",
    "metric": "aws:num-messages-sent",
    "criteria": {
      "consecutiveDatapointsToAlarm": 1,
      "consecutiveDatapointsToClear": 1,
      "mlDetectionConfig": {
        "confidenceLevel": "HIGH"
      }
    },
    "suppressAlerts": true
  },
  {
    "name": "num-authorization-failures-ml-behavior",
    "metric": "aws:num-authorization-failures",
    "criteria": {
      "consecutiveDatapointsToAlarm": 1,
      "consecutiveDatapointsToClear": 1,
      "mlDetectionConfig": {
        "confidenceLevel": "HIGH"
      }
    }
  }
]'
```

```

        "confidenceLevel": "HIGH"
    },
},
"suppressAlerts": true
},
{
    "name": "num-connection-attempts-ml-behavior",
    "metric": "aws:num-connection-attempts",
    "criteria": {
        "consecutiveDatapointsToAlarm": 1,
        "consecutiveDatapointsToClear": 1,
        "mlDetectionConfig": {
            "confidenceLevel": "HIGH"
        }
    },
    "suppressAlerts": true
},
{
    "name": "num-disconnects-ml-behavior",
    "metric": "aws:num-disconnects",
    "criteria": {
        "consecutiveDatapointsToAlarm": 1,
        "consecutiveDatapointsToClear": 1,
        "mlDetectionConfig": {
            "confidenceLevel": "HIGH"
        }
    },
    "suppressAlerts": true
}
]'
```

Output:

```
{
    "securityProfileName": "security-profile-for-smart-lights",
    "securityProfileArn": "arn:aws:iot:eu-west-1:123456789012:securityprofile/security-
profile-for-smart-lights"
}
```

3. Next, associate your Security Profile with one or multiple thing groups. Use the [attach-security-profile](#) command to attach a thing group to your Security Profile. The following example associates a thing group named `ML_Detect_beta_static_group` with the `security-profile-for-smart-lights` Security Profile.

```
aws iot attach-security-profile \
--security-profile-name security-profile-for-smart-lights \
--security-profile-target-arn arn:aws:iot:eu-
west-1:123456789012:thinggroup/ML_Detect_beta_static_group
```

Output:

None.

4. After you've created your complete Security Profile, the ML model begins training. The initial ML model training and building takes 14 days to complete. After 14 days, if there's anomalous activity on your device, you can expect to see alarms.

Monitor your ML model status

The following procedure shows you how to monitor your ML models in-progress training.

- Use the `get-behavior-model-training-summaries` command to view your ML model's progress. The following example gets the ML model training progress summary for the `security-profile-for-smart-lights` Security Profile. `modelStatus` shows you if a model has completed training or is still pending build for a particular behavior.

```
aws iot get-behavior-model-training-summaries \
--security-profile-name security-profile-for-smart-lights
```

Output:

```
{
  "summaries": [
    {
      "securityProfileName": "security-profile-for-smart-lights",
      "behaviorName": "Messages_sent_ML_behavior",
      "trainingDataCollectionStartDate": "2020-11-30T14:00:00-08:00",
      "modelStatus": "ACTIVE",
      "datapointsCollectionPercentage": 29.408,
      "lastModelRefreshDate": "2020-12-07T14:35:19.237000-08:00"
    },
    {
      "securityProfileName": "security-profile-for-smart-lights",
      "behaviorName": "Messages_received_ML_behavior",
      "modelStatus": "PENDING_BUILD",
      "datapointsCollectionPercentage": 0.0
    },
    {
      "securityProfileName": "security-profile-for-smart-lights",
      "behaviorName": "Authorization_failures_ML_behavior",
      "trainingDataCollectionStartDate": "2020-11-30T14:00:00-08:00",
      "modelStatus": "ACTIVE",
      "datapointsCollectionPercentage": 35.464,
      "lastModelRefreshDate": "2020-12-07T14:29:44.396000-08:00"
    },
    {
      "securityProfileName": "security-profile-for-smart-lights",
      "behaviorName": "Message_size_ML_behavior",
      "trainingDataCollectionStartDate": "2020-11-30T14:00:00-08:00",
      "modelStatus": "ACTIVE",
      "datapointsCollectionPercentage": 29.332,
      "lastModelRefreshDate": "2020-12-07T14:30:44.113000-08:00"
    },
    {
      "securityProfileName": "security-profile-for-smart-lights",
      "behaviorName": "Connection_attempts_ML_behavior",
      "trainingDataCollectionStartDate": "2020-11-30T14:00:00-08:00",
      "modelStatus": "ACTIVE",
      "datapointsCollectionPercentage": 32.891999999999996,
      "lastModelRefreshDate": "2020-12-07T14:29:43.121000-08:00"
    },
    {
      "securityProfileName": "security-profile-for-smart-lights",
      "behaviorName": "Disconnects_ML_behavior",
      "trainingDataCollectionStartDate": "2020-11-30T14:00:00-08:00",
      "modelStatus": "ACTIVE",
      "datapointsCollectionPercentage": 35.46,
      "lastModelRefreshDate": "2020-12-07T14:29:55.556000-08:00"
    }
  ]
}
```

Note

If your model doesn't progress as expected, make sure your devices are meeting the [Minimum requirements \(p. 889\)](#).

Review your ML Detect alarms

After your ML models are built and ready for data evaluations, you can regularly view any alarms that are inferred by the models. The following procedure shows you how to view your alarms in the AWS CLI.

- To see all active alarms, use the [list-active-violations](#) command.

```
aws iot list-active-violations \
--max-results 2
```

Output:

```
{  
    "activeViolations": []  
}
```

Alternatively, you can view all violations discovered during a given time period by using the [list-violation-events](#) command. The following example lists violation events from September 22, 2020 5:42:13 GMT to October 26, 2020 5:42:13 GMT.

```
aws iot list-violation-events \
--start-time 1599500533 \
--end-time 1600796533 \
--max-results 2
```

Output:

```
{  
    "violationEvents": [  
        {  
            "violationId": "1448be98c09c3d4ab7cb9b6f3ece65d6",  
            "thingName": "lightbulb-1",  
            "securityProfileName": "security-profile-for-smart-lights",  
            "behavior": {  
                "name": "LowConfidence_MladBehavior_MessagesSent",  
                "metric": "aws:num-messages-sent",  
                "criteria": {  
                    "consecutiveDatapointsToAlarm": 1,  
                    "consecutiveDatapointsToClear": 1,  
                    "mlDetectionConfig": {  
                        "confidenceLevel": "HIGH"  
                    }  
                },  
                "suppressAlerts": true  
            },  
            "violationEventType": "alarm-invalidated",  
            "violationEventTime": 1600780245.29  
        },  
        {  
            "violationId": "df4537569ef23efb1c029a433ae84b52",  
            "thingName": "lightbulb-2",  
            "securityProfileName": "security-profile-for-smart-lights",  
            "behavior": {  
                "name": "LowConfidence_MladBehavior_MessagesSent",  
                "metric": "aws:num-messages-sent",  
                "criteria": {  
                    "consecutiveDatapointsToAlarm": 1,  
                    "consecutiveDatapointsToClear": 1,  
                    "mlDetectionConfig": {  
                        "confidenceLevel": "HIGH"  
                    }  
                },  
                "suppressAlerts": true  
            },  
            "violationEventType": "alarm-invalidated",  
            "violationEventTime": 1600780245.29  
        }  
    ]  
}
```

```

        "consecutiveDatapointsToAlarm": 1,
        "consecutiveDatapointsToClear": 1,
        "mlDetectionConfig": {
            "confidenceLevel": "HIGH"
        }
    },
    "suppressAlerts": true
},
"violationEventType": "alarm-invalidated",
"violationEventTime": 1600780245.281
]
],
"nextToken":
"Amo6XIUrsoohsojuIG6TuwSR3X9iUvH2OcksBZg6bed2j21VSnD1uP1pfIxKX1+a3cvBRSosIB0xFv40kM6RYBknZ/vxabMe/ZW31Ps/WiZHlr9Wg7R7eEGli59IJ/U0iBQ1McP/ht0E2XA2TTIvYeMmKQOPsRj/eov9j7P/wveu7skNGepU/mvpV002Ap7hnV5U+Prx/9+iJA/341va+pQww7jpUeHmJN9Hw4MqW0ysw0Ry3w38h0QWEpz2xwFWAxAARxeIxCxt5c37RK/lRZBlhYqoB+w2PZ74730h8pICGY4gktJxkwHyyRabpSM/G/f5DFrD905v8idkTZZBxW2jrbzSUIdafPtsZHL/yAMKr3HAKtaAbZ2nTsOBNrre7X2d/jIjjarhon0Dh91+8I9Y5Ey+DIFBcqFTvhbKAafQt3gs6CUiqHdWiCenfJyb8whmDE2qxvdxGE1GmRb+k6kuN5jrZxxw95gzfYDgRHv11iEn8h1qZLD0czkIFBpMppHj9cetHPvM+qffXGAzKi8tL6eQuCdMLxmVE3jbqcJcjk9ItnaYJi5zKDz9FVbrz9qZZPtZJFHp"
}

```

Fine-tune your ML alarms

Once your ML models are built and ready for data evaluations, you can update your Security Profile's ML behavior settings to change the configuration. The following procedure shows you how to update your Security Profile's ML behavior settings in the AWS CLI.

- To change your Security Profile's ML behavior settings, use the [update-security-profile](#) command. The following example updates the ***security-profile-for-smart-lights*** Security Profile's behaviors by changing the confidenceLevel of a few of the behaviors and unsuppresses notifications for all behaviors.

```

aws iot update-security-profile \
--security-profile-name security-profile-for-smart-lights \
--behaviors \
'[
{
    "name": "num-messages-sent-ml-behavior",
    "metric": "aws:num-messages-sent",
    "criteria": {
        "mlDetectionConfig": {
            "confidenceLevel" : "HIGH"
        }
    },
    "suppressAlerts": false
},
{
    "name": "num-authorization-failures-ml-behavior",
    "metric": "aws:num-authorization-failures",
    "criteria": {
        "mlDetectionConfig": {
            "confidenceLevel" : "HIGH"
        }
    },
    "suppressAlerts": false
},
{
    "name": "num-connection-attempts-ml-behavior",
    "metric": "aws:num-connection-attempts",
    "criteria": {

```

```

        "mlDetectionConfig": {
            "confidenceLevel" : "HIGH"
        }
    },
    "suppressAlerts": false
},
{
    "name": "num-disconnects-ml-behavior",
    "metric": "aws:num-disconnects",
    "criteria": {
        "mlDetectionConfig": {
            "confidenceLevel" : "LOW"
        }
    },
    "suppressAlerts": false
}
]'
```

Output:

```
{
    "securityProfileName": "security-profile-for-smart-lights",
    "securityProfileArn": "arn:aws:iot:eu-west-1:123456789012:securityprofile/security-
profile-for-smart-lights",
    "behaviors": [
        {
            "name": "num-messages-sent-ml-behavior",
            "metric": "aws:num-messages-sent",
            "criteria": {
                "mlDetectionConfig": {
                    "confidenceLevel": "HIGH"
                }
            }
        },
        {
            "name": "num-authorization-failures-ml-behavior",
            "metric": "aws:num-authorization-failures",
            "criteria": {
                "mlDetectionConfig": {
                    "confidenceLevel": "HIGH"
                }
            }
        },
        {
            "name": "num-connection-attempts-ml-behavior",
            "metric": "aws:num-connection-attempts",
            "criteria": {
                "mlDetectionConfig": {
                    "confidenceLevel": "HIGH"
                }
            },
            "suppressAlerts": false
        },
        {
            "name": "num-disconnects-ml-behavior",
            "metric": "aws:num-disconnects",
            "criteria": {
                "mlDetectionConfig": {
                    "confidenceLevel": "LOW"
                }
            },
            "suppressAlerts": true
        }
    ],
}
```

```
        "version": 2,  
        "creationDate": 1600799559.249,  
        "lastModifiedDate": 1600800516.856  
    }
```

Mark your alarm's verification state

You can mark your alarms with verification states to help classify alarms and investigate anomalies.

- Mark your alarms with a verification state and a description of that state. For example to set an alarm's verification state to False positive, use the following command:

```
aws iot put-verification-state-on-violation --violation-id 12345 --verification-state FALSE_POSITIVE --verification-state-description "This is dummy description" --endpoint https://us-east-1.iot.amazonaws.com --region us-east-1
```

Output:

None.

Mitigate identified device issues

1. Use the `create-thing-group` command to create a thing group for the mitigation action. In the following example, we create a thing group called **ThingGroupForDetectMitigationAction**.

```
aws iot create-thing-group --thing-group-name ThingGroupForDetectMitigationAction
```

Output:

```
{  
    "thingGroupName": "ThingGroupForDetectMitigationAction",  
    "thingGroupArn": "arn:aws:iot:us-  
east-1:123456789012:thinggroup/ThingGroupForDetectMitigationAction",  
    "thingGroupId": "4139cd61-10fa-4c40-b867-0fc6209dca4d"  
}
```

2. Next, use the `create-mitigation-action` command to create a mitigation action. In the following example, we create a mitigation action called **detect_mitigation_action** with the ARN of the IAM role that is used to apply the mitigation action. We also define the type of action and the parameters for that action. In this case, our mitigation will move things to our previously created thing group called **ThingGroupForDetectMitigationAction**.

```
aws iot create-mitigation-action --action-name detect_mitigation_action \  
--role-arn arn:aws:iam::123456789012:role/MitigationActionValidRole \  
--action-params \  
'{  
    "addThingsToThingGroupParams": {  
        "thingGroupNames": ["ThingGroupForDetectMitigationAction"],  
        "overrideDynamicGroups": false  
    }  
'
```

Output:

```
{
```

```

    "actionArn": "arn:aws:iot:us-
east-1:123456789012:mitigationaction/detect_mitigation_action",
    "actionId": "5939e3a0-bf4c-44bb-a547-1ab59ffe67c3"
}

```

3. Use the [start-detect-mitigation-actions-task](#) command to start your mitigation actions task. task-id, target and actions are required parameters.

```

aws iot start-detect-mitigation-actions-task \
--task-id taskIdForMitigationAction \
--target '{ "violationIds" : [ "violationId-1", "violationId-2" ] }' \
--actions "detect_mitigation_action" \
--include-only-active-violations \
--include-suppressed-alerts

```

Output:

```
{
    "taskId": "taskIdForMitigationAction"
}
```

4. (Optional)To view mitigation action executions included in a task, use the [list-detect-mitigation-actions-executions](#) command.

```

aws iot list-detect-mitigation-actions-executions \
--task-id taskIdForMitigationAction \
--max-items 5 \
--page-size 4

```

Output:

```
{
    "actionsExecutions": [
        {
            "taskId": "e56ee95e - f4e7 - 459 c - b60a - 2701784290 af",
            "violationId": "214_fe0d92d21ee8112a6cf1724049d80",
            "actionName": "underTest_MATHingGroup71232127",
            "thingName": "cancelDetectMitigationActionsTaskd143821b",
            "executionStartDate": "Thu Jan 07 18: 35: 21 UTC 2021",
            "executionEndDate": "Thu Jan 07 18: 35: 21 UTC 2021",
            "status": "SUCCESSFUL",
        }
    ]
}
```

5. (Optional) Use the [describe-detect-mitigation-actions-task](#) command to get information about a mitigation action task.

```

aws iot describe-detect-mitigation-actions-task \
--task-id taskIdForMitigationAction

```

Output:

```
{
    "taskSummary": {
        "taskId": "taskIdForMitigationAction",
        "taskStatus": "SUCCESSFUL",
        "taskStartTime": 1609988361.224,
        "taskEndTime": 1609988362.281,
    }
}
```

```

    "target": {
        "securityProfileName": "security-profile-for-smart-lights",
        "behaviorName": "num-messages-sent-ml-behavior"
    },
    "violationEventOccurrenceRange": {
        "startTime": 1609986633.0,
        "endTime": 1609987833.0
    },
    "onlyActiveViolationsIncluded": true,
    "suppressedAlertsIncluded": true,
    "actionsDefinition": [
        {
            "name": "detect_mitigation_action",
            "id": "5939e3a0-bf4c-44bb-a547-1ab59ffe67c3",
            "roleArn": "arn:aws:iam::123456789012:role/MitigationActionValidRole",
            "actionParams": {
                "addThingsToThingGroupParams": {
                    "thingGroupNames": [
                        "ThingGroupForDetectMitigationAction"
                    ],
                    "overrideDynamicGroups": false
                }
            }
        }
    ],
    "taskStatistics": {
        "actionsExecuted": 0,
        "actionsSkipped": 0,
        "actionsFailed": 0
    }
}
}
}

```

- To get a list of your mitigation actions tasks, use the [list-detect-mitigation-actions-tasks](#) command.

```

aws iot list-detect-mitigation-actions-tasks \
--start-time 1609985315 \
--end-time 1609988915 \
--max-items 5 \
--page-size 4

```

Output:

```

{
    "tasks": [
        {
            "taskId": "taskIdForMitigationAction",
            "taskStatus": "SUCCESSFUL",
            "taskStartTime": 1609988361.224,
            "taskEndTime": 1609988362.281,
            "target": {
                "securityProfileName": "security-profile-for-smart-lights",
                "behaviorName": "num-messages-sent-ml-behavior"
            },
            "violationEventOccurrenceRange": {
                "startTime": 1609986633.0,
                "endTime": 1609987833.0
            },
            "onlyActiveViolationsIncluded": true,
            "suppressedAlertsIncluded": true,
            "actionsDefinition": [
                {

```

```
        "name": "detect_mitigation_action",
        "id": "5939e3a0-bf4c-44bb-a547-1ab59ffe67c3",
        "roleArn": "arn:aws:iam::123456789012:role/
MitigationActionValidRole",
        "actionParams": {
            "addThingsToThingGroupParams": {
                "thingGroupNames": [
                    "ThingGroupForDetectMitigationAction"
                ],
                "overrideDynamicGroups": false
            }
        }
    ],
    "taskStatistics": {
        "actionsExecuted": 0,
        "actionsSkipped": 0,
        "actionsFailed": 0
    }
}
]
```

7. (Optional) To cancel a mitigation actions task, use the [cancel-detect-mitigation-actions-task](#) command.

```
aws iot cancel-detect-mitigation-actions-task \
--task-id taskIdForMitigationAction
```

Output:

None.

Customize when and how you view AWS IoT Device Defender audit results

AWS IoT Device Defender audit provides periodic security checks to confirm AWS IoT devices and resources are following best practices. For each check, the audit results are categorized as compliant or non-compliant, where non-compliant results in console warning icons. To reduce noise from repeating known issues, the audit finding suppression feature allows you to temporarily silence these non-compliance notifications.

You can suppress select audit checks for a specific resource or account for a predetermined time period. An audit check result that has been suppressed is categorized as a suppressed finding, separate from the compliant and non-compliant categories. This new category doesn't trigger an alarm like a non-compliant result. This allows you to reduce non-compliance notification disturbances during known maintenance periods or until an update is scheduled to be completed.

Getting started

The following sections detail how you can use audit finding suppressions to suppress a Device certificate expiring check in the console and CLI. If you'd like to follow either of the demonstrations, you must first create two expiring certificates for Device Defender to detect.

Use the following to create your certificates.

- [Create and register a CA certificate \(p. 286\)](#)
- [Create a client certificate using your CA certificate. In step 3, set your days parameter to 1.](#)

If you're using the CLI to create your certificates, enter the following command.

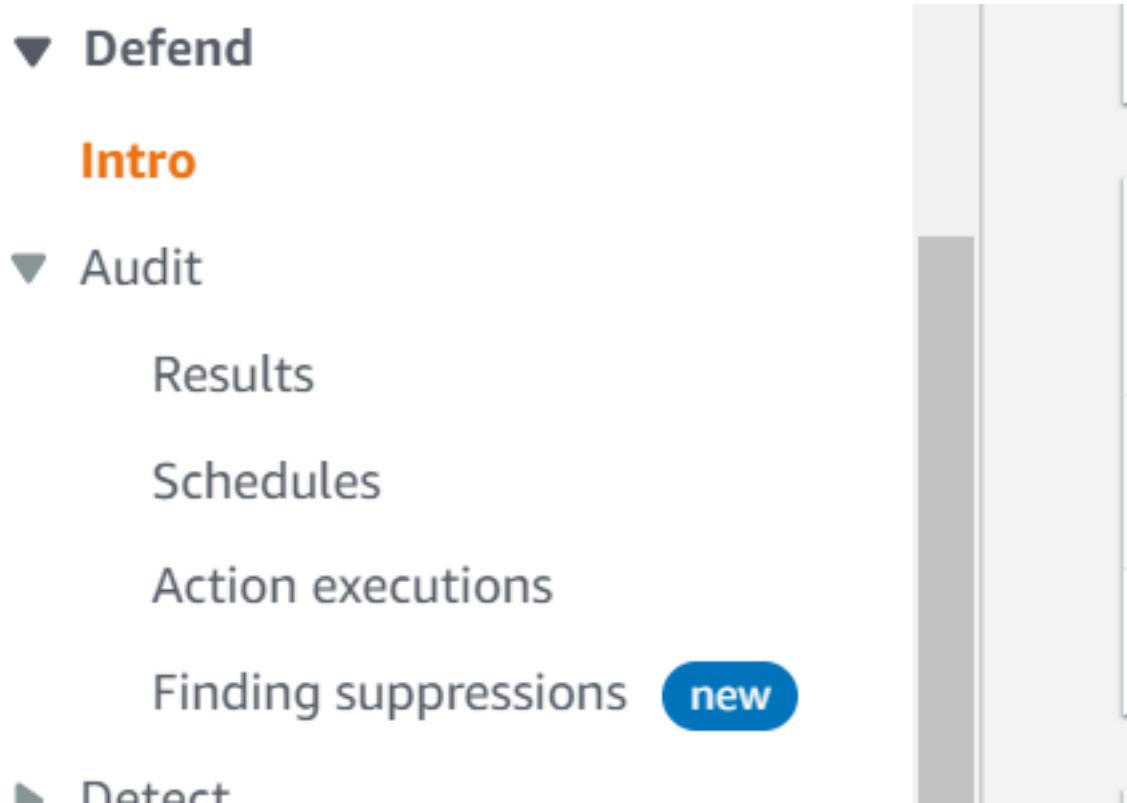
```
openssl x509 -req \
-in device_cert_csr_filename \
-CA root_ca_pem_filename \
-CAkey root_ca_key_filename \
-CAcreateserial \
-out device_cert_pem_filename \
-days 1 -sha256
```

Customize your audit findings in the console

The following walkthrough uses an account with two expired device certificates that trigger a non-compliant audit check. In this scenario, we want to disable the warning because our developers are testing a new feature that'll address the problem. We create an audit finding suppression for each certificate to stop the audit result from being non-compliant for the next week.

1. We will first run an on-demand audit to show that the expired device certificate check is non-compliant.

From the [AWS IoT console](#), choose **Defend** from the left sidebar, then **Audit**, and then **Results**. On the **Audit Results** page, choose **Create**. The **Create a new audit** window opens. Choose **Create**.



From the on-demand audit results, we can see that "Device certificate expiring" is non-compliant for two resources.

2. Now, we'd like to disable the "Device certificate expiring" non-compliant check warning because our developers are testing new features that will fix the warning.

From the left sidebar under **Defend**, choose **Audit**, and then choose **Finding suppressions**. On the **Audit finding suppressions** page, choose **Create**.

POLICIES

CAs

Role Aliases

Authorizers

▼ Defend

Intro

▼ Audit

Results

Schedules

Action executions

Finding suppressions

3. On the **Create an audit finding suppression** window, we need to fill out the following.
 - **Audit check:** We select `Device certificate expiring`, because that is the audit check we'd like to suppress.
 - **Resource identifier:** We input the device certificate ID of one of the certificates we'd like to suppress audit findings for.
 - **Suppression duration:** We select `1 week`, because that's how long we'd like to suppress the `Device certificate expiring` audit check for.
 - **Description (optional):** We add a note that describes why we're suppressing this audit finding.

Create an audit finding suppression

Suppressing an audit finding on a specified resource means that the audit check will no longer be run on the resource. Supressing an audit finding on a specified resource for the specified audit check will no longer make the device compliant.

Audit check

Device certificate expiring

Resource identifier

Device certificate id

b4490bd64c5cf85182f3182f1c03e70017e483f17b

Suppression duration

1 week

Description (optional)

Developer updates

After we've filled out the fields, choose **Create**. We see a success banner after the audit finding suppression has been created.

4. We've suppressed an audit finding for one of the certificates and now we need to suppress the audit finding for the second certificate. We could use the same suppression method that we used in step 3, but we will be using a different method for demonstration purposes.

From the left sidebar under **Defend**, choose **Audit**, and then choose **Results**. On the **Audit results** page, choose the audit with the non-compliant resource. Then, select the resource under **Non-compliant checks**. In our case, we select "Device certificate expiring".

5. On the **Device certificate expiring** page, under **Non-compliant policy** choose the option button next to the finding that needs to be suppressed. Next, choose the **Actions** dropdown menu, and then choose the duration for which you'd like finding to be suppressed. In our case, we choose 1 week as we did for the other certificate. On the **Confirm suppression** window, choose **Enable suppression**.

2 OUT OF 195 device certificates non-compliant

Mitigation

Consult your security best practices for how to proceed.

1. Provision a new certificate and attach it to the device.
2. Verify that the new certificate is valid and the device can connect.
3. Mark the old certificate as "INACTIVE" in the AWS IoT Device Defender audit results.
4. Detach the old certificate from the device. (See [Detaching a certificate](#))

Non-compliant certificate (2)

Finding



28022a890964e991852c79a28a83eb89



dc9b109c705ed7e68588bc54eef86f1c

We see a success banner after the audit finding suppression has been created. Now, both audit findings have been suppressed for 1 week while our developers work on a solution to address the warning.

Customize your audit findings in the CLI

The following walkthrough uses an account with an expired device certificate that trigger a non-compliant audit check. In this scenario, we want to disable the warning because our developers are testing a new feature that'll address the problem. We create an audit finding suppression for the certificate to stop the audit result from being non-compliant for the next week.

We use the following CLI commands.

- [create-audit-suppression](#)
- [describe-audit-suppression](#)
- [update-audit-suppression](#)
- [delete-audit-suppression](#)
- [list-audit-suppressions](#)

1. Use the following command to enable the audit.

```
aws iot update-account-audit-configuration \
  --audit-check-configurations "{\"DEVICE_CERTIFICATE_EXPIRING_CHECK\":{\"enabled \
  \":true}}"
```

Output:

None.

2. Use the following command to run an on-demand audit that targets the `DEVICE_CERTIFICATE_EXPIRING_CHECK` audit check.

```
aws iot start-on-demand-audit-task \
  --target-check-names DEVICE_CERTIFICATE_EXPIRING_CHECK
```

Output:

```
{ \
  "taskId": "787ed873b69cb4d6cdbae6ddd06996c5"
}
```

3. Use the [describe-account-audit-configuration](#) command to describe the audit configuration. We want to confirm that we've turned on the audit check for `DEVICE_CERTIFICATE_EXPIRING_CHECK`.

```
aws iot describe-account-audit-configuration
```

Output:

```
{
  "roleArn": "arn:aws:iam::<accountid>:role/service-role/project",
  "auditNotificationTargetConfigurations": {
    "SNS": {
```

```
"targetArn": "arn:aws:sns:us-east-1:<accountid>:project_sns",
"roleArn": "arn:aws:iam::<accountid>:role/service-role/project",
"enabled": true
},
"auditCheckConfigurations": {
    "AUTHTENTICATED_COGNITO_ROLE_OVERLY_PERMISSIVE_CHECK": {
        "enabled": false
    },
    "CA_CERTIFICATE_EXPIRING_CHECK": {
        "enabled": false
    },
    "CA_CERTIFICATE_KEY_QUALITY_CHECK": {
        "enabled": false
    },
    "CONFLICTING_CLIENT_IDS_CHECK": {
        "enabled": false
    },
    "DEVICE_CERTIFICATE_EXPIRING_CHECK": {
        "enabled": true
    },
    "DEVICE_CERTIFICATE_KEY_QUALITY_CHECK": {
        "enabled": false
    },
    "DEVICE_CERTIFICATE_SHARED_CHECK": {
        "enabled": false
    },
    "IOT_POLICY_OVERLY_PERMISSIVE_CHECK": {
        "enabled": true
    },
    "IOT_ROLE_ALIAS_ALLows_ACCESS_TO_UNUSED_SERVICES_CHECK": {
        "enabled": false
    },
    "IOT_ROLE_ALIAS_OVERLY_PERMISSIVE_CHECK": {
        "enabled": false
    },
    "LOGGING_DISABLED_CHECK": {
        "enabled": false
    },
    "REVOKED_CA_CERTIFICATE_STILL_ACTIVE_CHECK": {
        "enabled": false
    },
    "REVOKED_DEVICE_CERTIFICATE_STILL_ACTIVE_CHECK": {
        "enabled": false
    },
    "UNAUTHENTICATED_COGNITO_ROLE_OVERLY_PERMISSIVE_CHECK": {
        "enabled": false
    }
}
}
```

DEVICE_CERTIFICATE_EXPIRING_CHECK should have a value of true.

4. Use the [list-audit-task](#) command to identify the completed audit tasks.

```
aws iot list-audit-tasks \
--task-status "COMPLETED" \
--start-time 2020-07-31 \
--end-time 2020-08-01
```

Output:

```
{
```

```
"tasks": [
  {
    "taskId": "787ed873b69cb4d6cdbae6ddd06996c5",
    "taskStatus": "COMPLETED",
    "taskType": "SCHEDULED_AUDIT_TASK"
  }
]
```

The `taskId` of the audit you ran in step 1 should have a `taskStatus` of `COMPLETED`.

5. Use the [describe-audit-task](#) command to get details about the completed audit using the `taskId` output from the previous step. This command lists details about your audit.

```
aws iot describe-audit-task \
--task-id "787ed873b69cb4d6cdbae6ddd06996c5"
```

Output:

```
{
  "taskStatus": "COMPLETED",
  "taskType": "SCHEDULED_AUDIT_TASK",
  "taskStartTime": 1596168096.157,
  "taskStatistics": {
    "totalChecks": 1,
    "inProgressChecks": 0,
    "waitingForDataCollectionChecks": 0,
    "compliantChecks": 0,
    "nonCompliantChecks": 1,
    "failedChecks": 0,
    "canceledChecks": 0
  },
  "scheduledAuditName": "AWSIoTDeviceDefenderDailyAudit",
  "auditDetails": {
    "DEVICE_CERTIFICATE_EXPIRING_CHECK": {
      "checkRunStatus": "COMPLETED_NON_COMPLIANT",
      "checkCompliant": false,
      "totalResourcesCount": 195,
      "nonCompliantResourcesCount": 2
    }
  }
}
```

6. Use the [list-audit-findings](#) command to find the non-compliant certificate ID so that we can suspend the audit alerts for this resource.

```
aws iot list-audit-findings \
--start-time 2020-07-31 \
--end-time 2020-08-01
```

Output:

```
{
  "findings": [
    {
      "findingId": "296cccd39f806bf9d8f8de20d0ceb33a1",
      "taskId": "787ed873b69cb4d6cdbae6ddd06996c5",
      "checkName": "DEVICE_CERTIFICATE_EXPIRING_CHECK",
      "taskStartTime": 1596168096.157,
      "findingTime": 1596168096.651,
      "severity": "MEDIUM",
      "resourceArn": "arn:aws:iot:us-east-1:123456789012:cert/12345678901234567890123456789012"
    }
  ]
}
```

```

    "nonCompliantResource": {
        "resourceType": "DEVICE_CERTIFICATE",
        "resourceIdentifier": {
            "deviceCertificateId": "b4490<shortened>"
        },
        "additionalInfo": {
            "EXPIRATION_TIME": "1582862626000"
        }
    },
    "reasonForNonCompliance": "Certificate is past its expiration.",
    "reasonForNonComplianceCode": "CERTIFICATE_PAST_EXPIRATION",
    "isSuppressed": false
},
{
    "findingId": "37ecb79b7afb53deb328ec78e647631c",
    "taskId": "787ed873b69cb4d6cdbae6ddd06996c5",
    "checkName": "DEVICE_CERTIFICATE_EXPIRING_CHECK",
    "taskStartTime": 1596168096.157,
    "findingTime": 1596168096.651,
    "severity": "MEDIUM",
    "nonCompliantResource": {
        "resourceType": "DEVICE_CERTIFICATE",
        "resourceIdentifier": {
            "deviceCertificateId": "c7691<shortened>"
        },
        "additionalInfo": {
            "EXPIRATION_TIME": "1583424717000"
        }
    },
    "reasonForNonCompliance": "Certificate is past its expiration.",
    "reasonForNonComplianceCode": "CERTIFICATE_PAST_EXPIRATION",
    "isSuppressed": false
}
]
}

```

7. Use the [create-audit-suppression](#) command to suppress notifications for the `DEVICE_CERTIFICATE_EXPIRING_CHECK` audit check for a device certificate with the id `c7691e<shortened>` until `2020-08-20`.

```
aws iot create-audit-suppression \
--check-name DEVICE_CERTIFICATE_EXPIRING_CHECK \
--resource-identifier deviceCertificateId="c7691e<shortened>" \
--no-suppress-indefinitely \
--expiration-date 2020-08-20
```

8. Use the [list-audit-suppression](#) command to confirm the audit suppression setting and get details about the suppression.

```
aws iot list-audit-suppressions
```

Output:

```
{
    "suppressions": [
        {
            "checkName": "DEVICE_CERTIFICATE_EXPIRING_CHECK",
            "resourceIdentifier": {
                "deviceCertificateId": "c7691e<shortened>"
            },
            "expirationDate": 1597881600.0,
            "suppressIndefinitely": false
        }
    ]
}
```

```
    ]  
}
```

9. The [update-audit-suppression](#) command can be used to update the audit finding suppression. The example below updates the expiration-date to 08/21/20.

```
aws iot update-audit-suppression \  
  --check-name DEVICE_CERTIFICATE_EXPIRING_CHECK \  
  --resource-identifier deviceCertificateId=c7691e<shortened> \  
  --no-suppress-indefinitely \  
  --expiration-date 2020-08-21
```

10. The [delete-audit-suppression](#) command can be used to remove an audit finding suppression.

```
aws iot delete-audit-suppression \  
  --check-name DEVICE_CERTIFICATE_EXPIRING_CHECK \  
  --resource-identifier deviceCertificateId="c7691e<shortened>"
```

To confirm deletion, use the [list-audit-suppressions](#) command.

```
aws iot list-audit-suppressions
```

Output:

```
{  
  "suppressions": []  
}
```

In this tutorial, we showed you how to suppress a Device certificate expiring check in the console and CLI. For more information about audit finding suppressions, see [Audit finding suppressions \(p. 870\)](#)

Audit

An AWS IoT Device Defender audit looks at account- and device-related settings and policies to ensure security measures are in place. An audit can help you detect any drifts from security best practices or access policies (for example, multiple devices using the same identity, or overly permissive policies that allow one device to read and update data for many other devices). You can run audits as needed (*on-demand audits*) or schedule them to be run periodically (*scheduled audits*).

An AWS IoT Device Defender audit runs a set of predefined checks for common IoT security best practices and device vulnerabilities. Examples of predefined checks include policies that grant permission to read or update data on multiple devices, devices that share an identity (X.509 certificate), or certificates that are expiring or have been revoked but are still active.

Issue severity

Issue severity indicates the level of concern associated with each identified instance of noncompliance and the recommended time to remediation.

Critical

Non compliant audit checks with this severity identify issues that require urgent attention. Critical issues often allow bad actors with little sophistication and no insider knowledge or special credentials to easily gain access to or control of your assets.

High

Non compliant audit checks with this severity require urgent investigation and remediation planning after critical issues are addressed. Like critical issues, high severity issues often provide bad actors with access to or control of your assets. However, high severity issues are often more difficult to exploit. They might require special tools, insider knowledge, or specific setups.

Medium

Non compliant audit checks with this severity present issues that need attention as part of your continuous security posture maintenance. Medium severity issues might cause negative operational impact, such as unplanned outages due to malfunction of security controls. These issues might also provide bad actors with limited access to or control of your assets, or might facilitate parts of their malicious actions.

Low

Non compliant audit checks with this severity often indicate security best practices were overlooked or bypassed. Although they might not cause an immediate security impact on their own, these lapses can be exploited by bad actors. Like medium severity issues, low severity issues require attention as part of your continuous security posture maintenance.

Next steps

To understand the types of audit checks that can be performed, see [Audit checks \(p. 819\)](#). For information about service quotas that apply to audits, see [Service Quotas](#).

Audit checks

Note

When you enable a check, data collection starts immediately. If there is a large amount of data in your account to collect, results of the check might not be available for some time after you enabled it.

The following audit checks are supported:

- CA certificate revoked but device certificates still active (p. 819)
- Device certificate shared (p. 820)
- Device certificate key quality (p. 821)
- CA certificate key quality (p. 822)
- Unauthenticated Cognito role overly permissive (p. 823)
- Authenticated Cognito role overly permissive (p. 829)
- AWS IoT policies overly permissive (p. 835)
- Role alias overly permissive (p. 839)
- Role alias allows access to unused services (p. 840)
- CA certificate expiring (p. 840)
- Conflicting MQTT client IDs (p. 841)
- Device certificate expiring (p. 842)
- Revoked device certificate still active (p. 843)
- Logging disabled (p. 843)

CA certificate revoked but device certificates still active

A CA certificate was revoked, but is still active in AWS IoT.

This check appears as `REVOKED_CA_CERTIFICATE_STILL_ACTIVE_CHECK` in the CLI and API.

Severity: **Critical**

Details

A CA certificate is marked as revoked in the certificate revocation list maintained by the issuing authority, but is still marked as ACTIVE or PENDING_TRANSFER in AWS IoT.

The following reason codes are returned when this check finds a noncompliant CA certificate:

- `CERTIFICATE_REVOKED_BY_ISSUER`

Why it matters

A revoked CA certificate should no longer be used to sign device certificates. It might have been revoked because it was compromised. Newly added devices with certificates signed using this CA certificate might pose a security threat.

How to fix it

1. Use [UpdateCACertificate](#) to mark the CA certificate as INACTIVE in AWS IoT. You can also use mitigation actions to:
 - Apply the `UPDATE_CA_CERTIFICATE` mitigation action on your audit findings to make this change.
 - Apply the `PUBLISH_FINDINGS_TO_SNS` mitigation action to implement a custom response in response to the Amazon SNS message.For more information, see [Mitigation actions \(p. 929\)](#).
2. Review the device certificate registration activity for the time after the CA certificate was revoked and consider revoking any device certificates that might have been issued with it during this time. You can use [ListCertificatesByCA](#) to list the device certificates signed by the CA certificate and [UpdateCertificate](#) to revoke a device certificate.

Device certificate shared

Multiple, concurrent connections use the same X.509 certificate to authenticate with AWS IoT.

This check appears as `DEVICE_CERTIFICATE_SHARED_CHECK` in the CLI and API.

Severity: **Critical**

Details

When performed as part of an on-demand audit, this check looks at the certificates and client IDs that were used by devices to connect during the 31 days before the start of the audit up to 2 hours before the check is run. For scheduled audits, this check looks at data from 2 hours before the last time the audit was run to 2 hours before the time this instance of the audit started. If you have taken steps to mitigate this condition during the time checked, note when the concurrent connections were made to determine if the problem persists.

The following reason codes are returned when this check finds a noncompliant certificate:

- `CERTIFICATE_SHARED_BY_MULTIPLE_DEVICES`

In addition, the findings returned by this check include the ID of the shared certificate, the IDs of the clients using the certificate to connect, and the connect/disconnect times. Most recent results are listed first.

Why it matters

Each device should have a unique certificate to authenticate with AWS IoT. When multiple devices use the same certificate, this might indicate that a device has been compromised. Its identity might have been cloned to further compromise the system.

How to fix it

Verify that the device certificate has not been compromised. If it has, follow your security best practices to mitigate the situation.

If you use the same certificate on multiple devices, you might want to:

1. Provision new, unique certificates and attach them to each device.
2. Verify that the new certificates are valid and the devices can use them to connect.
3. Use [UpdateCertificate](#) to mark the old certificate as REVOKED in AWS IoT. You can also use mitigation actions to do the following:
 - Apply the `UPDATE_DEVICE_CERTIFICATE` mitigation action on your audit findings to make this change.
 - Apply the `ADD_THINGS_TO_THING_GROUP` mitigation action to add the device to a group where you can take action on it.
 - Apply the `PUBLISH_FINDINGS_TO_SNS` mitigation action if you want to implement a custom response in response to the Amazon SNS message.

For more information, see [Mitigation actions \(p. 929\)](#).

4. Detach the old certificate from each of the devices.

Device certificate key quality

AWS IoT customers often rely on TLS mutual authentication using X.509 certificates for authenticating to AWS IoT message broker. These certificates and their certificate authority certificates must be registered in their AWS IoT account before they are used. AWS IoT performs basic sanity checks on these certificates when they are registered. These checks include:

- They must be in a valid format.
- They must be signed by a registered certificate authority.
- They must still be within their validity period (in other words, they haven't expired).
- Their cryptographic key sizes must meet a minimum required size (for RSA keys, they must be 2048 bits or larger).

This audit check provides the following additional tests of the quality of your cryptographic key:

- CVE-2008-0166 – Check whether the key was generated using OpenSSL 0.9.8c-1 up to versions before 0.9.8g-9 on a Debian-based operating system. Those versions of OpenSSL use a random number generator that generates predictable numbers, making it easier for remote attackers to conduct brute force guessing attacks against cryptographic keys.
- CVE-2017-15361 – Check whether the key was generated by the Infineon RSA library 1.02.013 in Infineon Trusted Platform Module (TPM) firmware, such as versions before 000000000000422 – 4.34, before 000000000000062b – 6.43, and before 0000000000008521 – 133.33. That library mishandles RSA key generation, making it easier for attackers to defeat some cryptographic protection mechanisms through targeted attacks. Examples of affected technologies include BitLocker with TPM 1.2, YubiKey 4 (before 4.3.5) PGP key generation, and the Cached User Data encryption feature in Chrome OS.

AWS IoT Device Defender reports certificates as noncompliant if they fail these tests.

This check appears as `DEVICE_CERTIFICATE_KEY_QUALITY_CHECK` in the CLI and API.

Severity: **Critical**

Details

This check applies to device certificates that are ACTIVE or PENDING_TRANSFER.

The following reason codes are returned when this check finds a noncompliant certificate:

- `CERTIFICATE_KEY_VULNERABILITY CVE-2017-15361`
- `CERTIFICATE_KEY_VULNERABILITY CVE-2008-0166`

Why it matters

When a device uses a vulnerable certificate, attackers can more easily compromise that device.

How to fix it

Update your device certificates to replace those with known vulnerabilities.

If you are using the same certificate on multiple devices, you might want to:

1. Provision new, unique certificates and attach them to each device.
2. Verify that the new certificates are valid and the devices can use them to connect.
3. Use [UpdateCertificate](#) to mark the old certificate as REVOKED in AWS IoT. You can also use mitigation actions to:
 - Apply the `UPDATE_DEVICE_CERTIFICATE` mitigation action on your audit findings to make this change.
 - Apply the `ADD_THINGS_TO_THING_GROUP` mitigation action to add the device to a group where you can take action on it.
 - Apply the `PUBLISH_FINDINGS_TO_SNS` mitigation action if you want to implement a custom response in response to the Amazon SNS message.

For more information, see [Mitigation actions \(p. 929\)](#).

4. Detach the old certificate from each of the devices.

CA certificate key quality

AWS IoT customers often rely on TLS mutual authentication using X.509 certificates for authenticating to AWS IoT message broker. These certificates and their certificate authority certificates must be registered in their AWS IoT account before they are used. AWS IoT performs basic sanity checks on these certificates when they are registered, including:

- The certificates are in a valid format.
- The certificates are within their validity period (in other words, not expired).
- Their cryptographic key sizes meet a minimum required size (for RSA keys, they must be 2048 bits or larger).

This audit check provides the following additional tests of the quality of your cryptographic key:

- CVE-2008-0166 – Check whether the key was generated using OpenSSL 0.9.8c-1 up to versions before 0.9.8g-9 on a Debian-based operating system. Those versions of OpenSSL use a random number generator that generates predictable numbers, making it easier for remote attackers to conduct brute force guessing attacks against cryptographic keys.
- CVE-2017-15361 – Check whether the key was generated by the Infineon RSA library 1.02.013 in Infineon Trusted Platform Module (TPM) firmware, such as versions before 0000000000000422 – 4.34, before 000000000000062b – 6.43, and before 00000000000008521 – 133.33. That library mishandles RSA key generation, making it easier for attackers to defeat some cryptographic protection mechanisms through targeted attacks. Examples of affected technologies include BitLocker with TPM 1.2, YubiKey 4 (before 4.3.5) PGP key generation, and the Cached User Data encryption feature in Chrome OS.

AWS IoT Device Defender reports certificates as noncompliant if they fail these tests.

This check appears as `CA_CERTIFICATE_KEY_QUALITY_CHECK` in the CLI and API.

Severity: **Critical**

Details

This check applies to CA certificates that are ACTIVE or PENDING_TRANSFER.

The following reason codes are returned when this check finds a noncompliant certificate:

- `CERTIFICATE_KEY_VULNERABILITY_CVE-2017-15361`
- `CERTIFICATE_KEY_VULNERABILITY_CVE-2008-0166`

Why it matters

Newly added devices signed using this CA certificate might pose a security threat.

How to fix it

1. Use [UpdateCACertificate](#) to mark the CA certificate as INACTIVE in AWS IoT. You can also use mitigation actions to:
 - Apply the `UPDATE_CA_CERTIFICATE` mitigation action on your audit findings to make this change.
 - Apply the `PUBLISH_FINDINGS_TO_SNS` mitigation action if you want to implement a custom response in response to the Amazon SNS message.For more information, see [Mitigation actions \(p. 929\)](#).
2. Review the device certificate registration activity for the time after the CA certificate was revoked and consider revoking any device certificates that might have been issued with it during this time. (Use [ListCertificatesByCA](#) to list the device certificates signed by the CA certificate and [UpdateCertificate](#) to revoke a device certificate.)

Unauthenticated Cognito role overly permissive

A policy attached to an unauthenticated Amazon Cognito identity pool role is considered too permissive because it grants permission to perform any of the following AWS IoT actions:

- Manage or modify things.
- Read thing administrative data.
- Manage non-thing related data or resources.

Or, because it grants permission to perform the following AWS IoT actions on a broad set of devices:

- Use MQTT to connect, publish, or subscribe to reserved topics (including shadow or job execution data).
- Use API commands to read or modify shadow or job execution data.

In general, devices that connect using an unauthenticated Amazon Cognito identity pool role should have only limited permission to publish and subscribe to thing-specific MQTT topics or use the API commands to read and modify thing-specific data related to shadow or job execution data.

This check appears as `UNAUTHENTICATED_COGNITO_ROLE_OVERLY_PERMISSIVE_CHECK` in the CLI and API.

Severity: **Critical**

Details

For this check, AWS IoT Device Defender audits all Amazon Cognito identity pools that have been used to connect to the AWS IoT message broker during the 31 days before the audit execution. All Amazon Cognito identity pools from which either an authenticated or unauthenticated Amazon Cognito identity connected are included in the audit.

The following reason codes are returned when this check finds a noncompliant unauthenticated Amazon Cognito identity pool role:

- `ALLOWS_ACCESS_TO_IOT_ADMIN_ACTIONS`
- `ALLOWS_BROAD_ACCESS_TO_IOT_DATA_PLANE_ACTIONS`

Why it matters

Because unauthenticated identities are never authenticated by the user, they pose a much greater risk than authenticated Amazon Cognito identities. If an unauthenticated identity is compromised, it can use administrative actions to modify account settings, delete resources, or gain access to sensitive data. Or, with broad access to device settings, it can access or modify shadows and jobs for all devices in your account. A guest user might use the permissions to compromise your entire fleet or launch a DDOS attack with messages.

How to fix it

A policy attached to an unauthenticated Amazon Cognito identity pool role should grant only those permissions required for a device to do its job. We recommend the following steps:

1. Create a new compliant role.
2. Create a Amazon Cognito identity pool and attach the compliant role to it.
3. Verify that your identities can access AWS IoT using the new pool.
4. After verification is complete, attach the compliant role to the Amazon Cognito identity pool that was flagged as noncompliant.

You can also use mitigation actions to:

- Apply the `PUBLISH_FINDINGS_TO_SNS` mitigation action to implement a custom response in response to the Amazon SNS message.

For more information, see [Mitigation actions \(p. 929\)](#).

Manage or modify things

The following AWS IoT API actions are used to manage or modify things. Permission to perform these actions should not be granted to devices that connect through an unauthenticated Amazon Cognito identity pool.

- `AddThingToThingGroup`
- `AttachThingPrincipal`
- `CreateThing`
- `DeleteThing`
- `DetachThingPrincipal`
- `ListThings`
- `ListThingsInThingGroup`
- `RegisterThing`
- `RemoveThingFromThingGroup`
- `UpdateThing`
- `UpdateThingGroupsForThing`

Any role that grants permission to perform these actions on even a single resource is considered noncompliant.

Read thing administrative data

The following AWS IoT API actions are used to read or modify thing data. Devices that connect through an unauthenticated Amazon Cognito identity pool should not be given permission to perform these actions.

- `DescribeThing`
- `ListJobExecutionsForThing`
- `ListThingGroupsForThing`
- `ListThingPrincipals`

Example

- noncompliant:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:DescribeThing",
                "iot>ListJobExecutionsForThing",
                "iot>ListThingGroupsForThing",
                "iot>ListThingPrincipals"
            ],
            "Resource": [
                "arn:aws:iot:region:account-id:thing/MyThing"
            ]
        }
    ]
}
```

This allows the device to perform the specified actions even though it is granted for one thing only.

Manage non-things

Devices that connect through an unauthenticated Amazon Cognito identity pool should not be given permission to perform AWS IoT API actions other than those discussed in these sections. You can manage your account with an application that connects through an unauthenticated Amazon Cognito identity pool by creating a separate identity pool not used by devices.

Subscribe/publish to MQTT topics

MQTT messages are sent through the AWS IoT message broker and are used by devices to perform many actions, including accessing and modifying shadow state and job execution state. A policy that grants permission to a device to connect, publish, or subscribe to MQTT messages should restrict these actions to specific resources as follows:

Connect

- noncompliant:

```
arn:aws:iot:region:account-id:client/*
```

The wildcard * allows any device to connect to AWS IoT.

```
arn:aws:iot:region:account-id:client/${iot:ClientId}
```

Unless iot:Connection.Thing.IsAttached is set to true in the condition keys, this is equivalent to the wildcard * in the previous example.

- compliant:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [ "iot:Connect" ],  
            "Resource": [  
                "arn:aws:iot:region:account-id:client/${iot:Connection.Thing.ThingName}"  
            ],  
            "Condition": {  
                "Bool": { "iot:Connection.Thing.IsAttached": "true" }  
            }  
        }  
    ]  
}
```

The resource specification contains a variable that matches the device name used to connect. The condition statement further restricts the permission by checking that the certificate used by the MQTT client matches that attached to the thing with the name used.

Publish

- noncompliant:

```
arn:aws:iot:region:account-id:topic/$aws/things/*/shadow/update
```

This allows the device to update the shadow of any device (* = all devices).

```
arn:aws:iot:region:account-id:topic/$aws/things/*
```

This allows the device to read, update, or delete the shadow of any device.

- compliant:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [ "iot:Publish" ],
            "Resource": [
                "arn:aws:iot:region:account-id:topic/$aws/things/
${iot:Connection.Thing.ThingName}/shadow/*"
            ],
        }
    ]
}
```

The resource specification contains a wildcard, but it only matches any shadow-related topic for the device whose thing name is used to connect.

Subscribe

- noncompliant:

```
arn:aws:iot:region:account-id:topicfilter/$aws/things/*
```

This allows the device to subscribe to reserved shadow or job topics for all devices.

```
arn:aws:iot:region:account-id:topicfilter/$aws/things/*
```

The same as the previous example, but using the # wildcard.

```
arn:aws:iot:region:account-id:topicfilter/$aws/things/#/shadow/update
```

This allows the device to see shadow updates on any device (+ = all devices).

- compliant:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [ "iot:Subscribe" ],
            "Resource": [
                "arn:aws:iot:region:account-id:topicfilter/$aws/things/
${iot:Connection.Thing.ThingName}/shadow/*"
                "arn:aws:iot:region:account-id:topicfilter/$aws/things/
${iot:Connection.Thing.ThingName}/jobs/*"
            ],
        }
    ]
}
```

The resource specifications contain wildcards, but they only match any shadow-related topic and any job-related topic for the device whose thing name is used to connect.

Receive

- compliant:

```
arn:aws:iot:region:account-id:topicfilter/$aws/things/*
```

This is allowed because the device can receive messages only from topics on which it has permission to subscribe.

Read/modify shadow or job data

A policy that grants permission to a device to perform an API action to access or modify device shadows or job execution data should restrict these actions to specific resources. The following are the API actions:

- DeleteThingShadow
- GetThingShadow
- UpdateThingShadow
- DescribeJobExecution
- GetPendingJobExecutions
- StartNextPendingJobExecution
- UpdateJobExecution

Example

- noncompliant:

```
arn:aws:iot:region:account-id:thing/*
```

This allows the device to perform the specified action on any thing.

- compliant:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:DeleteThingShadow",
                "iot:GetThingShadow",
                "iot:UpdateThingShadow",
                "iot:DescribeJobExecution",
                "iot:GetPendingJobExecutions",
                "iot:StartNextPendingJobExecution",
                "iot:UpdateJobExecution"
            ],
            "Resource": [
                "arn:aws:iot:region:account-id:thing/MyThing1",
                "arn:aws:iot:region:account-id:thing/MyThing2"
            ]
        }
    ]
}
```

This allows the device to perform the specified actions on two things only.

Authenticated Cognito role overly permissive

A policy attached to an authenticated Amazon Cognito identity pool role is considered too permissive because it grants permission to perform the following AWS IoT actions:

- Manage or modify things.
- Manage non-thing related data or resources.

Or, because it grants permission to perform the following AWS IoT actions on a broad set of devices:

- Read thing administrative data.
- Use MQTT to connect/publish/subscribe to reserved topics (including shadow or job execution data).
- Use API commands to read or modify shadow or job execution data.

In general, devices that connect using an authenticated Amazon Cognito identity pool role should have only limited permission to read thing-specific administrative data, publish and subscribe to thing-specific MQTT topics, or use the API commands to read and modify thing-specific data related to shadow or job execution data.

This check appears as `AUTHENTICATED_COGNITO_ROLE_OVERLY_PERMISSIVE_CHECK` in the CLI and API.

Severity: **Critical**

Details

For this check, AWS IoT Device Defender audits all Amazon Cognito identity pools that have been used to connect to the AWS IoT message broker during the 31 days before the audit execution. All Amazon Cognito identity pools from which either an authenticated or unauthenticated Amazon Cognito identity connected are included in the audit.

The following reason codes are returned when this check finds a noncompliant authenticated Amazon Cognito identity pool role:

- `ALLOWS_BROAD_ACCESS_TO_IOT_THING_ADMIN_READ_ACTIONS`
- `ALLOWS_ACCESS_TO_IOT_NON_THING_ADMIN_ACTIONS`
- `ALLOWS_ACCESS_TO_IOT_THING_ADMIN_WRITE_ACTIONS`

Why it matters

If an authenticated identity is compromised, it can use administrative actions to modify account settings, delete resources, or gain access to sensitive data.

How to fix it

A policy attached to an authenticated Amazon Cognito identity pool role should grant only those permissions required for a device to do its job. We recommend the following steps:

1. Create a new compliant role.
2. Create a Amazon Cognito identity pool and attach the compliant role to it.
3. Verify that your identities can access AWS IoT using the new pool.
4. After verification is complete, attach the role to the Amazon Cognito identity pool that was flagged as noncompliant.

You can also use mitigation actions to:

- Apply the `PUBLISH_FINDINGS_TO_SNS` mitigation action to implement a custom response in response to the Amazon SNS message.

For more information, see [Mitigation actions \(p. 929\)](#).

Manage or modify things

The following AWS IoT API actions are used to manage or modify things so permission to perform these should not be granted to devices connecting through an authenticated Amazon Cognito identity pool:

- `AddThingToThingGroup`
- `AttachThingPrincipal`
- `CreateThing`
- `DeleteThing`
- `DetachThingPrincipal`
- `ListThings`
- `ListThingsInThingGroup`
- `RegisterThing`
- `RemoveThingFromThingGroup`
- `UpdateThing`
- `UpdateThingGroupsForThing`

Any role that grants permission to perform these actions on even a single resource is considered noncompliant.

Manage non-things

Devices that connect through an authenticated Amazon Cognito identity pool should not be given permission to perform AWS IoT API actions other than those discussed in these sections. To manage your account with an application that connects through an authenticated Amazon Cognito identity pool, create a separate identity pool not used by devices.

Read thing administrative data

The following AWS IoT API actions are used to read thing data, so devices that connect through an authenticated Amazon Cognito identity pool should be given permission to perform these on a limited set of things only:

- `DescribeThing`
 - `ListJobExecutionsForThing`
 - `ListThingGroupsForThing`
 - `ListThingPrincipals`
- noncompliant:

```
arn:aws:iot:region:account-id:thing/*
```

This allows the device to perform the specified action on any thing.

- compliant:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:DescribeThing",
        "iot>ListJobExecutionsForThing",
        "iot>ListThingGroupsForThing",
        "iot>ListThingPrincipals"
      ],
      "Resource": [
        "arn:aws:iot:region:account-id:thing/MyThing"
      ]
    }
  ]
}
```

This allows the device to perform the specified actions on only one thing.

- compliant:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:DescribeThing",
        "iot>ListJobExecutionsForThing",
        "iot>ListThingGroupsForThing",
        "iot>ListThingPrincipals"
      ],
      "Resource": [
        "arn:aws:iot:region:account-id:thing/MyThing*"
      ]
    }
  ]
}
```

This is compliant because, although the resource is specified with a wildcard (*), it is preceded by a specific string, and that limits the set of things accessed to those with names that have the given prefix.

- noncompliant:

```
arn:aws:iot:region:account-id:thing/*
```

This allows the device to perform the specified action on any thing.

- compliant:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:DescribeThing",
        "iot>ListJobExecutionsForThing",
        "iot>ListThingGroupsForThing",
        "iot>ListThingPrincipals"
      ],
      "Resource": [
        "arn:aws:iot:region:account-id:thing/*"
      ]
    }
  ]
}
```

```

        "iot>ListJobExecutionsForThing",
        "iot>ListThingGroupsForThing",
        "iot>ListThingPrincipals"
    ],
    "Resource": [
        "arn:aws:iot:region:account-id:/thing/MyThing"
    ]
}
]
}
}

```

This allows the device to perform the specified actions on only one thing.

- compliant:

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:DescribeThing",
                "iot>ListJobExecutionsForThing",
                "iot>ListThingGroupsForThing",
                "iot>ListThingPrincipals"
            ],
            "Resource": [
                "arn:aws:iot:region:account-id:/thing/MyThing*"
            ]
        }
    ]
}

```

This is compliant because, although the resource is specified with a wildcard (*), it is preceded by a specific string, and that limits the set of things accessed to those with names that have the given prefix.

Subscribe/publish to MQTT topics

MQTT messages are sent through the AWS IoT message broker and are used by devices to perform many different actions, including accessing and modifying shadow state and job execution state. A policy that grants permission to a device to connect, publish, or subscribe to MQTT messages should restrict these actions to specific resources as follows:

Connect

- noncompliant:

```
arn:aws:iot:region:account-id:client/*
```

The wildcard * allows any device to connect to AWS IoT.

```
arn:aws:iot:region:account-id:client/${iot:ClientId}
```

Unless `iot:Connection.Thing.IsAttached` is set to true in the condition keys, this is equivalent to the wildcard * in the previous example.

- compliant:

```
{
```

```

    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [ "iot:Connect" ],
            "Resource": [
                "arn:aws:iot:region:account-id:client/${iot:Connection.Thing.ThingName}"
            ],
            "Condition": {
                "Bool": { "iot:Connection.Thing.IsAttached": "true" }
            }
        }
    ]
}

```

The resource specification contains a variable that matches the device name used to connect, and the condition statement further restricts the permission by checking that the certificate used by the MQTT client matches that attached to the thing with the name used.

Publish

- noncompliant:

```
arn:aws:iot:region:account-id:topic/$aws/things/*shadow/update
```

This allows the device to update the shadow of any device (* = all devices).

```
arn:aws:iot:region:account-id:topic/$aws/things/*
```

This allows the device to read/update/delete the shadow of any device.

- compliant:

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [ "iot:Publish" ],
            "Resource": [
                "arn:aws:iot:region:account-id:topic/$aws/things/
${iot:Connection.Thing.ThingName}/shadow/*"
            ],
        }
    ]
}

```

The resource specification contains a wildcard, but it only matches any shadow-related topic for the device whose thing name is used to connect.

Subscribe

- noncompliant:

```
arn:aws:iot:region:account-id:topicfilter/$aws/things/*
```

This allows the device to subscribe to reserved shadow or job topics for all devices.

```
arn:aws:iot:region:account-id:topicfilter/$aws/things/#
```

The same as the previous example, but using the # wildcard.

```
arn:aws:iot:region:account-id:topicfilter/$aws/things/+shadow/update
```

This allows the device to see shadow updates on any device (+ = all devices).

- compliant:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [ "iot:Subscribe" ],  
            "Resource": [  
                "arn:aws:iot:region:account-id:topicfilter/$aws/things/  
${iot:Connection.ThingName}/shadow/*"  
                "arn:aws:iot:region:account-id:topicfilter/$aws/things/  
${iot:Connection.ThingName}/jobs/*"  
            ],  
        }  
    ]  
}
```

The resource specifications contain wildcards, but they only match any shadow-related topic and any job-related topic for the device whose thing name is used to connect.

Receive

- compliant:

```
arn:aws:iot:region:account-id:topicfilter/$aws/things/*
```

This is compliant because the device can receive messages only from topics on which it has permission to subscribe.

Read or modify shadow or job data

A policy that grants permission to a device to perform an API action to access or modify device shadows or job execution data should restrict these actions to specific resources. The following are the API actions:

- DeleteThingShadow
- GetThingShadow
- UpdateThingShadow
- DescribeJobExecution
- GetPendingJobExecutions
- StartNextPendingJobExecution
- UpdateJobExecution

Examples

- noncompliant:

```
arn:aws:iot:region:account-id:thing/*
```

This allows the device to perform the specified action on any thing.

- compliant:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:DeleteThingShadow",  
                "iot:GetThingShadow",  
                "iot:UpdateThingShadow",  
                "iot:DescribeJobExecution",  
                "iot:GetPendingJobExecutions",  
                "iot:StartNextPendingJobExecution",  
                "iot:UpdateJobExecution"  
            ],  
            "Resource": [  
                "arn:aws:iot:region:account-id:/thing/MyThing1",  
                "arn:aws:iot:region:account-id:/thing/MyThing2"  
            ]  
        }  
    ]  
}
```

This allows the device to perform the specified actions on only two things.

AWS IoT policies overly permissive

An AWS IoT policy gives permissions that are too broad or unrestricted. It grants permission to send or receive MQTT messages for a broad set of devices, or grants permission to access or modify shadow and job execution data for a broad set of devices.

In general, a policy for a device should grant access to resources associated with just that device and no or very few other devices. With some exceptions, using a wildcard (for example, "*") to specify resources in such a policy is considered too broad or unrestricted.

This check appears as `IOT_POLICY_OVERLY_PERMISSIVE_CHECK` in the CLI and API.

Severity: **Critical**

Details

The following reason code is returned when this check finds a noncompliant AWS IoT policy:

- `ALLOWS_BROAD_ACCESS_TO_IOT_DATA_PLANE_ACTIONS`

Why it matters

A certificate, Amazon Cognito identity, or thing group with an overly permissive policy can, if compromised, impact the security of your entire account. An attacker could use such broad access to read or modify shadows, jobs, or job executions for all your devices. Or an attacker could use a compromised certificate to connect malicious devices or launch a DDOS attack on your network.

How to fix it

Follow these steps to fix any noncompliant policies attached to things, thing groups, or other entities:

1. Use [CreatePolicyVersion](#) to create a new, compliant version of the policy. Set the `setAsDefault` flag to true. (This makes this new version operative for all entities that use the policy.)

2. Use [ListTargetsForPolicy](#) to get a list of targets (certificates, thing groups) that the policy is attached to and determine which devices are included in the groups or which use the certificates to connect.
3. Verify that all associated devices are able to connect to AWS IoT. If a device is unable to connect, use [SetPolicyVersion](#) to roll back the default policy to the previous version, revise the policy, and try again.

You can use mitigation actions to:

- Apply the `REPLACE_DEFAULT_POLICY_VERSION` mitigation action on your audit findings to make this change.
- Apply the `PUBLISH_FINDINGS_TO_SNS` mitigation action if you want to implement a custom response in response to the Amazon SNS message.

For more information, see [Mitigation actions \(p. 929\)](#).

Use [AWS IoT policy variables \(p. 318\)](#) to dynamically reference AWS IoT resources in your policies.

MQTT permissions

MQTT messages are sent through the AWS IoT message broker and are used by devices to perform many actions, including accessing and modifying shadow state and job execution state. A policy that grants permission to a device to connect, publish, or subscribe to MQTT messages should restrict these actions to specific resources as follows:

Connect

- noncompliant:

```
arn:aws:iot:region:account-id:client/*
```

The wildcard * allows any device to connect to AWS IoT.

```
arn:aws:iot:region:account-id:client/${iot:ClientId}
```

Unless `iot:Connection.Thing.IsAttached` is set to true in the condition keys, this is equivalent to the wildcard * as in the previous example.

- compliant:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [ "iot:Connect" ],
      "Resource": [
        "arn:aws:iot:region:account-id:client/${iot:Connection.Thing.ThingName}"
      ],
      "Condition": {
        "Bool": { "iot:Connection.Thing.IsAttached": "true" }
      }
    }
  ]
}
```

The resource specification contains a variable that matches the device name used to connect. The condition statement further restricts the permission by checking that the certificate used by the MQTT client matches that attached to the thing with the name used.

Publish

- noncompliant:

```
arn:aws:iot:region:account-id:topic/$aws/things/*shadow/update
```

This allows the device to update the shadow of any device (* = all devices).

```
arn:aws:iot:region:account-id:topic/$aws/things/*
```

This allows the device to read, update, or delete the shadow of any device.

- compliant:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [ "iot:Publish" ],
      "Resource": [
        "arn:aws:iot:region:account-id:topic/$aws/things/
${iot:Connection.Thing.ThingName}/shadow/*"
      ],
    }
  ]
}
```

The resource specification contains a wildcard, but it only matches any shadow-related topic for the device whose thing name is used to connect.

Subscribe

- noncompliant:

```
arn:aws:iot:region:account-id:topicfilter/$aws/things/*
```

This allows the device to subscribe to reserved shadow or job topics for all devices.

```
arn:aws:iot:region:account-id:topicfilter/$aws/things/*
```

The same as the previous example, but using the # wildcard.

```
arn:aws:iot:region:account-id:topicfilter/$aws/things/#shadow/update
```

This allows the device to see shadow updates on any device (+ = all devices).

- compliant:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [ "iot:Subscribe" ],
      "Resource": [
        "arn:aws:iot:region:account-id:topicfilter/$aws/things/
${iot:Connection.Thing.ThingName}/shadow/*"
        "arn:aws:iot:region:account-id:topicfilter/$aws/things/
${iot:Connection.Thing.ThingName}/jobs/*"
      ],
    }
  ]
}
```

```
        ],
    }
}
```

The resource specifications contain wildcards, but they only match any shadow-related topic and any job-related topic for the device whose thing name is used to connect.

Receive

- compliant:

```
arn:aws:iot:region:account-id:topic/$aws/things/*
```

This is compliant because the device can only receive messages from topics on which it has permission to subscribe.

Shadow and job permissions

A policy that grants permission to a device to perform an API action to access or modify device shadows or job execution data should restrict these actions to specific resources. The following are the API actions:

- DeleteThingShadow
- GetThingShadow
- UpdateThingShadow
- DescribeJobExecution
- GetPendingJobExecutions
- StartNextPendingJobExecution
- UpdateJobExecution

Examples

- noncompliant:

```
arn:aws:iot:region:account-id:thing/*
```

This allows the device to perform the specified action on any thing.

- compliant:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:DeleteThingShadow",
        "iot:GetThingShadow",
        "iot:UpdateThingShadow",
        "iot:DescribeJobExecution",
        "iot:GetPendingJobExecutions",
        "iot:StartNextPendingJobExecution",
        "iot:UpdateJobExecution"
      ],
      "Resource": [
        "arn:aws:iot:region:account-id:/thing/MyThing1",
        "arn:aws:iot:region:account-id:/job/MyJob1"
      ]
    }
}
```

```
        "arn:aws:iot:region:account-id:/thing/MyThing2"  
    }  
}  
}
```

This allows the device to perform the specified actions on only two things.

Role alias overly permissive

AWS IoT role alias provides a mechanism for connected devices to authenticate to AWS IoT using X.509 certificates and then obtain short-lived AWS credentials from an IAM role that is associated with an AWS IoT role alias. The permissions for these credentials must be scoped down using access policies with authentication context variables. If your policies are not configured correctly, you could leave yourself exposed to an escalation of privilege attack. This audit check ensures that the temporary credentials provided by AWS IoT role aliases are not overly permissive.

This check is triggered if one of the following conditions are found:

- The policy provides administrative permissions to any services used in the past year by this role alias (for example, "iot:*", "dynamodb:*", "iam:*", and so on).
- The policy provides broad access to thing metadata actions, access to restricted AWS IoT actions, or broad access to AWS IoT data plane actions.
- The policy provides access to security auditing services such as "iam", "cloudtrail", "guardduty", "inspector", or "trustedadvisor".

This check appears as `IOT_ROLE_ALIAS_OVERLY_PERMISSIVE_CHECK` in the CLI and API.

Severity: **Critical**

Details

The following reason codes are returned when this check finds a noncompliant IoT policy:

- `ALLOWS_BROAD_ACCESS_TO_USED_SERVICES`
- `ALLOWS_ACCESS_TO_SECURITY_AUDITING_SERVICES`
- `ALLOWS_BROAD_ACCESS_TO_IOT_THING_ADMIN_READ_ACTIONS`
- `ALLOWS_ACCESS_TO_IOT_NON_THING_ADMIN_ACTIONS`
- `ALLOWS_ACCESS_TO_IOT_THING_ADMIN_WRITE_ACTIONS`
- `ALLOWS_BROAD_ACCESS_TO_IOT_DATA_PLANE_ACTIONS`

Why it matters

By limiting permissions to those that are required for a device to perform its normal operations, you reduce the risks to your account if a device is compromised.

How to fix it

Follow these steps to fix any noncompliant policies attached to things, thing groups, or other entities:

1. Follow the steps in [Authorizing direct calls to AWS services using AWS IoT Core credential provider \(p. 355\)](#) to apply a more restrictive policy to your role alias.

You can use mitigation actions to:

- Apply the `PUBLISH_FINDINGS_TO_SNS` mitigation action if you want to implement a custom action in response to the Amazon SNS message.

For more information, see [Mitigation actions \(p. 929\)](#).

Role alias allows access to unused services

AWS IoT role alias provides a mechanism for connected devices to authenticate to AWS IoT using X.509 certificates and then obtain short-lived AWS credentials from an IAM role that is associated with an AWS IoT role alias. The permissions for these credentials must be scoped down using access policies with authentication context variables. If your policies are not configured correctly, you could leave yourself exposed to an escalation of privilege attack. This audit check ensures that the temporary credentials provided by AWS IoT role aliases are not overly permissive.

This check is triggered if the role alias has access to services that haven't been used for the AWS IoT device in the last year. For example, the audit reports if you have an IAM role linked to the role alias that has only used AWS IoT in the past year but the policy attached to the role also grants permission to `"iam:getRole"` and `"dynamodb:PutItem"`.

This check appears as `IOT_ROLE_ALIAS_ALLOWS_ACCESS_TO_UNUSED_SERVICES_CHECK` in the CLI and API.

Severity: **Medium**

Details

The following reason codes are returned when this check finds a noncompliant AWS IoT policy:

- `ALLows_ACCESS_TO_UNUSED_SERVICES`

Why it matters

By limiting permissions to those services that are required for a device to perform its normal operations, you reduce the risks to your account if a device is compromised.

How to fix it

Follow these steps to fix any noncompliant policies attached to things, thing groups, or other entities:

1. Follow the steps in [Authorizing direct calls to AWS services using AWS IoT Core credential provider \(p. 355\)](#) to apply a more restrictive policy to your role alias.

You can use mitigation actions to:

- Apply the `PUBLISH_FINDINGS_TO_SNS` mitigation action if you want to implement a custom action in response to the Amazon SNS message.

For more information, see [Mitigation actions \(p. 929\)](#).

CA certificate expiring

A CA certificate is expiring within 30 days or has expired.

This check appears as `CA_CERTIFICATE_EXPIRING_CHECK` in the CLI and API.

Severity: **Medium**

Details

This check applies to CA certificates that are ACTIVE or PENDING_TRANSFER.

The following reason codes are returned when this check finds a noncompliant CA certificate:

- CERTIFICATE_APPROACHING_EXPIRATION
- CERTIFICATE_PAST_EXPIRATION

Why it matters

An expired CA certificate should not be used to sign new device certificates.

How to fix it

Consult your security best practices for how to proceed. You might want to:

1. Register a new CA certificate with AWS IoT.
2. Verify that you are able to sign device certificates using the new CA certificate.
3. Use [UpdateCACertificate](#) to mark the old CA certificate as INACTIVE in AWS IoT. You can also use mitigation actions to do the following:
 - Apply the UPDATE_CA_CERTIFICATE mitigation action on your audit findings to make this change.
 - Apply the PUBLISH_FINDINGS_TO_SNS mitigation action if you want to implement a custom response in response to the Amazon SNS message.

For more information, see [Mitigation actions \(p. 929\)](#).

Conflicting MQTT client IDs

Multiple devices connect using the same client ID.

This check appears as CONFLICTING_CLIENT_IDS_CHECK in the CLI and API.

Severity: **High**

Details

Multiple connections were made using the same client ID, causing an already connected device to be disconnected. The MQTT specification allows only one active connection per client ID, so when another device connects using the same client ID, it knocks the previous one off the connection.

When performed as part of an on-demand audit, this check looks at how client IDs were used to connect during the 31 days prior to the start of the audit. For scheduled audits, this check looks at data from the last time the audit was run to the time this instance of the audit started. If you have taken steps to mitigate this condition during the time checked, note when the connections/disconnections were made to determine if the problem persists.

The following reason codes are returned when this check finds noncompliance:

- DUPLICATE_CLIENT_ID_ACROSS_CONNECTIONS

The findings returned by this check also include the client ID used to connect, principal IDs, and disconnect times. The most recent results are listed first.

Why it matters

Devices with conflicting IDs are forced to constantly reconnect, which might result in lost messages or leave a device unable to connect.

This might indicate that a device or a device's credentials have been compromised, and might be part of a DDoS attack. It is also possible that devices are not configured correctly in the account or a device has a bad connection and is forced to reconnect several times per minute.

How to fix it

Register each device as a unique thing in AWS IoT, and use the thing name as the client ID to connect. Or use a UUID as the client ID when connecting the device over MQTT. You can also use mitigation actions to:

- Apply the `PUBLISH_FINDINGS_TO_SNS` mitigation action if you want to implement a custom response in response to the Amazon SNS message.

For more information, see [Mitigation actions \(p. 929\)](#).

Device certificate expiring

A device certificate is expiring within 30 days or has expired.

This check appears as `DEVICE_CERTIFICATE_EXPIRING_CHECK` in the CLI and API.

Severity: **Medium**

Details

This check applies to device certificates that are `ACTIVE` or `PENDING_TRANSFER`.

The following reason codes are returned when this check finds a noncompliant device certificate:

- `CERTIFICATE_APPROACHING_EXPIRATION`
- `CERTIFICATE_PAST_EXPIRATION`

Why it matters

A device certificate should not be used after it expires.

How to fix it

Consult your security best practices for how to proceed. You might want to:

1. Provision a new certificate and attach it to the device.
2. Verify that the new certificate is valid and the device is able to use it to connect.
3. Use [UpdateCertificate](#) to mark the old certificate as `INACTIVE` in AWS IoT. You can also use mitigation actions to:
 - Apply the `UPDATE_DEVICE_CERTIFICATE` mitigation action on your audit findings to make this change.
 - Apply the `ADD_THINGS_TO_THING_GROUP` mitigation action to add the device to a group where you can take action on it.
 - Apply the `PUBLISH_FINDINGS_TO_SNS` mitigation action if you want to implement a custom response in response to the Amazon SNS message.

For more information, see [Mitigation actions \(p. 929\)](#).

4. Detach the old certificate from the device. (See [DetachThingPrincipal](#).)

Revoked device certificate still active

A revoked device certificate is still active.

This check appears as `REVOKE_DEVICE_CERTIFICATE_STILL_ACTIVE_CHECK` in the CLI and API.

Severity: **Medium**

Details

A device certificate is in its CA's [certificate revocation list](#), but it is still active in AWS IoT.

This check applies to device certificates that are `ACTIVE` or `PENDING_TRANSFER`.

The following reason codes are returned when this check finds noncompliance:

- `CERTIFICATE_REVOKED_BY_ISSUER`

Why it matters

A device certificate is usually revoked because it has been compromised. It is possible that it has not yet been revoked in AWS IoT due to an error or oversight.

How to fix it

Verify that the device certificate has not been compromised. If it has, follow your security best practices to mitigate the situation. You might want to:

1. Provision a new certificate for the device.
2. Verify that the new certificate is valid and the device is able to use it to connect.
3. Use [UpdateCertificate](#) to mark the old certificate as `REVOKED` in AWS IoT. You can also use mitigation actions to:
 - Apply the `UPDATE_DEVICE_CERTIFICATE` mitigation action on your audit findings to make this change.
 - Apply the `ADD_THINGS_TO_THING_GROUP` mitigation action to add the device to a group where you can take action on it.
 - Apply the `PUBLISH_FINDINGS_TO_SNS` mitigation action if you want to implement a custom response in response to the Amazon SNS message.

For more information, see [Mitigation actions \(p. 929\)](#).

4. Detach the old certificate from the device. (See [DetachThingPrincipal](#).)

Logging disabled

AWS IoT logs are not enabled in Amazon CloudWatch. Verifies both V1 and V2 logging.

This check appears as `LOGGING_DISABLED_CHECK` in the CLI and API.

Severity: **Low**

Details

The following reason codes are returned when this check finds noncompliance:

- LOGGING_DISABLED

Why it matters

AWS IoT logs in CloudWatch provide visibility into behaviors in AWS IoT, including authentication failures and unexpected connects and disconnects that might indicate that a device has been compromised.

How to fix it

Enable AWS IoT logs in CloudWatch. See [Monitoring Tools \(p. 390\)](#). You can also use mitigation actions to:

- Apply the ENABLE_IOT_LOGGING mitigation action on your audit findings to make this change.
- Apply the PUBLISH_FINDINGS_TO_SNS mitigation action if you want to implement a custom response in response to the Amazon SNS message.

For more information, see [Mitigation actions \(p. 929\)](#).

Audit commands

Manage audit settings

Use `UpdateAccountAuditConfiguration` to configure audit settings for your account. This command allows you to enable those checks you want to be available for audits, set up optional notifications, and configure permissions.

Check these settings with `DescribeAccountAuditConfiguration`.

Use `DeleteAccountAuditConfiguration` to delete your audit settings. This restores all default values, and effectively disables audits because all checks are disabled by default.

[UpdateAccountAuditConfiguration](#)

Configures or reconfigures the Device Defender audit settings for this account. Settings include how audit notifications are sent and which audit checks are enabled or disabled.

Synopsis

```
aws iot update-account-audit-configuration \
[--role-arn <value>] \
[--audit-notification-target-configurations <value>] \
[--audit-check-configurations <value>] \
[--cli-input-json <value>] \
[--generate-cli-skeleton]
```

cli-input-json format

```
{
  "roleArn": "string",
  "auditNotificationTargetConfigurations": {
    "string": {
      "targetArn": "string",
```

```

        "roleArn": "string",
        "enabled": "boolean"
    }
},
"auditCheckConfigurations": {
    "string": {
        "enabled": "boolean"
    }
}
}

```

cli-input-json Fields

Name	Type	Description
roleArn	string length- max:2048 min:20	The ARN of the role that grants permission to AWS IoT to access information about your devices, policies, certificates, and other items when performing an audit.
auditNotificationTargetConfiguration	map	Information about the targets to which audit notifications are sent.
targetArn	string	The ARN of the target (SNS topic) to which audit notifications are sent.
roleArn	string length- max:2048 min:20	The ARN of the role that grants permission to send notifications to the target.
enabled	boolean	True if notifications to the target are enabled.
auditCheckConfigurations	map	<p>Specifies which audit checks are enabled and disabled for this account. Use DescribeAccountAuditConfiguration to see the list of all checks, including those that are currently enabled.</p> <p>Some data collection might start immediately when certain checks are enabled. When a check is disabled, any data collected so far in relation to the check is deleted.</p> <p>You cannot disable a check if it is used by any scheduled audit. You must first delete the check from the scheduled audit or delete the scheduled audit itself.</p> <p>On the first call to UpdateAccountAuditConfiguration, this parameter is required</p>

Name	Type	Description
		and must specify at least one enabled check.
enabled	boolean	True if this audit check is enabled for this account.

Output

None

Errors

`InvalidRequestException`

The contents of the request were invalid.

`ThrottlingException`

The rate exceeds the limit.

`InternalFailureException`

An unexpected error has occurred.

DescribeAccountAuditConfiguration

Gets information about the Device Defender audit settings for this account. Settings include how audit notifications are sent and which audit checks are enabled or disabled.

Synopsis

```
aws iot describe-account-audit-configuration \
[--cli-input-json <value>] \
[--generate-cli-skeleton]
```

`cli-input-json` format

```
{  
}
```

Output

```
{
  "roleArn": "string",
  "auditNotificationTargetConfigurations": {
    "string": {
      "targetArn": "string",
      "roleArn": "string",
      "enabled": "boolean"
    }
  },
  "auditCheckConfigurations": {
    "string": {
      "enabled": "boolean"
    }
  }
}
```

```
}
```

CLI output fields

Name	Type	Description
roleArn	string length- max:2048 min:20	The ARN of the role that grants permission to AWS IoT to access information about your devices, policies, certificates, and other items when performing an audit. On the first call to <code>UpdateAccountAuditConfiguration</code> , this parameter is required.
auditNotificationTargetConfigurationMap	map	Information about the targets to which audit notifications are sent for this account.
targetArn	string	The ARN of the target (SNS topic) to which audit notifications are sent.
roleArn	string length- max:2048 min:20	The ARN of the role that grants permission to send notifications to the target.
enabled	boolean	True if notifications to the target are enabled.
auditCheckConfigurations	map	Which audit checks are enabled and disabled for this account.
enabled	boolean	True if this audit check is enabled for this account.

Errors

`ThrottlingException`

The rate exceeds the limit.

`InternalFailureException`

An unexpected error has occurred.

DeleteAccountAuditConfiguration

Restores the default settings for Device Defender audits for this account. Any configuration data you entered is deleted and all audit checks are reset to disabled.

Synopsis

```
aws iot delete-account-audit-configuration \
[--delete-scheduled-audits | --no-delete-scheduled-audits] \
[--cli-input-json <value>] \
```

```
[--generate-cli-skeleton]
```

cli-input-json format

```
{  
    "deleteScheduledAudits": "boolean"  
}
```

cli-input-json Fields

Name	Type	Description
deleteScheduledAudits	boolean	If true, all scheduled audits are deleted.

Output

None

Errors

InvalidRequestException

The contents of the request were invalid.

ResourceNotFoundException

The specified resource does not exist.

ThrottlingException

The rate exceeds the limit.

InternalFailureException

An unexpected error has occurred.

Schedule audits

Use `CreateScheduledAudit` to create one or more scheduled audits. This command allows you to specify the checks you want to perform during an audit and how often the audit should be run.

Keep track of your scheduled audits with `ListScheduledAudits` and `DescribeScheduledAudit`.

Change an existing scheduled audit with `UpdateScheduledAudit` or delete it with `DeleteScheduledAudit`.

CreateScheduledAudit

Creates a scheduled audit that is run at a specified time interval.

Synopsis

```
aws iot create-scheduled-audit \  
    --frequency <value> \  
    [--day-of-month <value>] \  
    [--day-of-week <value>] \  

```

```
--target-check-names <value> \
[--tags <value>] \
--scheduled-audit-name <value> \
[--cli-input-json <value>] \
[--generate-cli-skeleton]
```

cli-input-json format

```
{
  "frequency": "string",
  "dayOfMonth": "string",
  "dayOfWeek": "string",
  "targetCheckNames": [
    "string"
  ],
  "tags": [
    {
      "Key": "string",
      "Value": "string"
    }
  ],
  "scheduledAuditName": "string"
}
```

cli-input-json Fields

Name	Type	Description
frequency	string	How often the scheduled audit takes place. Can be one of DAILY, WEEKLY, BIWEEKLY, or MONTHLY. The actual start time of each audit is determined by the system. enum: DAILY WEEKLY BIWEEKLY MONTHLY
dayOfMonth	string pattern: ^([1-9] [12][0-9] 3[01])\$ ^LAST\$	The day of the month on which the scheduled audit takes place. Can be 1 through 31 or LAST. This field is required if the frequency parameter is set to MONTHLY. If days 29-31 are specified, and the month does not have that many days, the audit takes place on the LAST day of the month.
dayOfWeek	string	The day of the week on which the scheduled audit takes place. Can be one of SUN, MON, TUE, WED, THU, FRI, or SAT. This field is required if the frequency parameter is set to WEEKLY or BIWEEKLY. enum: SUN MON TUE WED THU FRI SAT

Name	Type	Description
targetCheckNames	list member: AuditCheckName	Which checks are performed during the scheduled audit. Checks must be enabled for your account. (Use DescribeAccountAuditConfiguration to see the list of all checks, including those that are enabled or UpdateAccountAuditConfiguration to select which checks are enabled.)
tags	list member: Tag java class: java.util.List	Metadata that can be used to manage the scheduled audit.
Key	string	The tag's key.
Value	string	The tag's value.
scheduledAuditName	string length- max:128 min:1 pattern: [a-zA-Z0-9_-]+	The name you want to give to the scheduled audit. (Maximum of 128 characters)

Output

```
{
  "scheduledAuditArn": "string"
}
```

CLI output fields

Name	Type	Description
scheduledAuditArn	string	The ARN of the scheduled audit.

Errors

`InvalidRequestException`

The contents of the request were invalid.

`ThrottlingException`

The rate exceeds the limit.

`InternalFailureException`

An unexpected error has occurred.

`LimitExceededException`

A limit has been exceeded.

ListScheduledAudits

Lists all of your scheduled audits.

Synopsis

```
aws iot list-scheduled-audits \
[--next-token <value>] \
[--max-results <value>] \
[--cli-input-json <value>] \
[--generate-cli-skeleton]
```

cli-input-json format

```
{
  "nextToken": "string",
  "maxResults": "integer"
}
```

cli-input-json Fields

Name	Type	Description
nextToken	string	The token for the next set of results.
maxResults	integer range- max:250 min:1	The maximum number of results to return at one time. The default is 25.

Output

```
{
  "scheduledAudits": [
    {
      "scheduledAuditName": "string",
      "scheduledAuditArn": "string",
      "frequency": "string",
      "dayOfMonth": "string",
      "dayOfWeek": "string"
    }
  ],
  "nextToken": "string"
}
```

CLI output fields

Name	Type	Description
scheduledAudits	list member: ScheduledAuditMetadata java class: java.util.List	The list of scheduled audits.
scheduledAuditName	string	The name of the scheduled audit.

Name	Type	Description
	length- max:128 min:1 pattern: [a-zA-Z0-9_-]+	
scheduledAuditArn	string	The ARN of the scheduled audit.
frequency	string	How often the scheduled audit takes place. enum: DAILY WEEKLY BIWEEKLY MONTHLY
dayOfMonth	string pattern: ^([1-9] [12][0-9] 3[01])\$ ^LAST\$	The day of the month on which the scheduled audit is run (if the frequency is MONTHLY). If days 29-31 are specified, and the month does not have that many days, the audit takes place on the LAST day of the month.
dayOfWeek	string	The day of the week on which the scheduled audit is run (if the frequency is WEEKLY or BIWEEKLY). enum: SUN MON TUE WED THU FRI SAT
nextToken	string	A token that can be used to retrieve the next set of results, or null if there are no more results.

Errors

InvalidRequestException

The contents of the request were invalid.

ThrottlingException

The rate exceeds the limit.

InternalFailureException

An unexpected error has occurred.

DescribeScheduledAudit

Gets information about a scheduled audit.

Synopsis

```
aws iot describe-scheduled-audit \
--scheduled-audit-name <value> \
[--cli-input-json <value>] \
[--generate-cli-skeleton]
```

cli-input-json format

```
{
  "scheduledAuditName": "string"
}
```

cli-input-json Fields

Name	Type	Description
scheduledAuditName	string length- max:128 min:1 pattern: [a-zA-Z0-9_-]+	The name of the scheduled audit whose information you want to get.

Output

```
{
  "frequency": "string",
  "dayOfMonth": "string",
  "dayOfWeek": "string",
  "targetCheckNames": [
    "string"
  ],
  "scheduledAuditName": "string",
  "scheduledAuditArn": "string"
}
```

CLI output fields

Name	Type	Description
frequency	string	How often the scheduled audit takes place. One of DAILY, WEEKLY, BIWEEKLY, or MONTHLY. The actual start time of each audit is determined by the system. enum: DAILY WEEKLY BIWEEKLY MONTHLY
dayOfMonth	string pattern: ^([1-9] [12][0-9] 3[01])\\$ ^LAST\\$	The day of the month on which the scheduled audit takes place. Can be 1 through 31 or LAST. If days 29-31 are specified, and the month does not have that many days, the audit takes place on the LAST day of the month.
dayOfWeek	string	The day of the week on which the scheduled audit takes place. One of SUN, MON, TUE, WED, THU, FRI, or SAT. enum: SUN MON TUE WED THU FRI SAT

Name	Type	Description
targetCheckNames	list member: AuditCheckName	Which checks are performed during the scheduled audit. Checks must be enabled for your account. (Use DescribeAccountAuditConfiguration to see the list of all checks, including those that are enabled or use UpdateAccountAuditConfiguration to select which checks are enabled.)
scheduledAuditName	string length- max:128 min:1 pattern: [a-zA-Z0-9_-]+	The name of the scheduled audit.
scheduledAuditArn	string	The ARN of the scheduled audit.

Errors

InvalidRequestException

The contents of the request were invalid.

ResourceNotFoundException

The specified resource does not exist.

ThrottlingException

The rate exceeds the limit.

InternalFailureException

An unexpected error has occurred.

UpdateScheduledAudit

Updates a scheduled audit, including which checks are performed and how often the audit takes place.

Synopsis

```
aws iot update-scheduled-audit \
[--frequency <value>] \
[--day-of-month <value>] \
[--day-of-week <value>] \
[--target-check-names <value>] \
--scheduled-audit-name <value> \
[--cli-input-json <value>] \
[--generate-cli-skeleton]
```

cli-input-json format

```
{
  "frequency": "string",
  "dayOfMonth": "string",
  "dayOfWeek": "string",
```

```

    "targetCheckNames": [
        "string"
    ],
    "scheduledAuditName": "string"
}

```

cli-input-json Fields

Name	Type	Description
frequency	string	How often the scheduled audit takes place. Can be one of DAILY, WEEKLY, BIWEEKLY, or MONTHLY. The actual start time of each audit is determined by the system. enum: DAILY WEEKLY BIWEEKLY MONTHLY
dayOfMonth	string pattern: ^([1-9] [12][0-9] 3[01])\$ ^LAST\$	The day of the month on which the scheduled audit takes place. Can be 1 through 31 or LAST. This field is required if the frequency parameter is set to MONTHLY. If days 29-31 are specified, and the month does not have that many days, the audit takes place on the LAST day of the month.
dayOfWeek	string	The day of the week on which the scheduled audit takes place. Can be one of SUN, MON, TUE, WED, THU, FRI, or SAT. This field is required if the frequency parameter is set to WEEKLY or BIWEEKLY. enum: SUN MON TUE WED THU FRI SAT
targetCheckNames	list member: AuditCheckName	Which checks are performed during the scheduled audit. Checks must be enabled for your account. (Use DescribeAccountAuditConfiguration to see the list of all checks, including those that are enabled or use UpdateAccountAuditConfiguration to select which checks are enabled.)
scheduledAuditName	string length- max:128 min:1 pattern: [a-zA-Z0-9_-]+	The name of the scheduled audit. (Maximum of 128 characters)

Output

```
{
  "scheduledAuditArn": "string"
}
```

CLI output fields

Name	Type	Description
scheduledAuditArn	string	The ARN of the scheduled audit.

Errors

`InvalidRequestException`

The contents of the request were invalid.

`ResourceNotFoundException`

The specified resource does not exist.

`ThrottlingException`

The rate exceeds the limit.

`InternalFailureException`

An unexpected error has occurred.

DeleteScheduledAudit

Deletes a scheduled audit.

Synopsis

```
aws iot delete-scheduled-audit \
  --scheduled-audit-name <value> \
  [--cli-input-json <value>] \
  [--generate-cli-skeleton]
```

`cli-input-json` format

```
{
  "scheduledAuditName": "string"
}
```

cli-input-json Fields

Name	Type	Description
scheduledAuditName	string length- max:128 min:1 pattern: [a-zA-Z0-9_-]+	The name of the scheduled audit you want to delete.

Output

None

Errors

`InvalidRequestException`

The contents of the request were invalid.

`ResourceNotFoundException`

The specified resource does not exist.

`ThrottlingException`

The rate exceeds the limit.

`InternalFailureException`

An unexpected error has occurred.

Run an On-Demand audit

Use `StartOnDemandAuditTask` to specify the checks you want to perform and start an audit running right away.

StartOnDemandAuditTask

Starts an on-demand Device Defender audit.

Synopsis

```
aws iot start-on-demand-audit-task \
  --target-check-names <value> \
  [--cli-input-json <value>] \
  [--generate-cli-skeleton]
```

`cli-input-json` format

```
{
  "targetCheckNames": [
    "string"
  ]
}
```

cli-input-json Fields

Name	Type	Description
<code>targetCheckNames</code>	list member: <code>AuditCheckName</code>	Which checks are performed during the audit. The checks you specify must be enabled for your account or an exception occurs. Use <code>DescribeAccountAuditConfiguration</code> to see the list of all checks, including those that are enabled or use <code>UpdateAccountAuditConfiguration</code> to select which checks are enabled.

Output

```
{  
    "taskId": "string"  
}
```

CLI output fields

Name	Type	Description
taskId	string length- max:40 min:1 pattern: [a-zA-Z0-9-]+	The ID of the on-demand audit you started.

Errors

`InvalidRequestException`

The contents of the request were invalid.

`ThrottlingException`

The rate exceeds the limit.

`InternalFailureException`

An unexpected error has occurred.

`LimitExceededException`

A limit has been exceeded.

Manage audit instances

Use `DescribeAuditTask` to get information about a specific audit instance. If it has already run, the results include which checks failed and which passed, those that the system was unable to complete, and if the audit is still in progress, those it is still working on.

Use `ListAuditTasks` to find the audits that were run during a specified time interval.

Use `CancelAuditTask` to halt an audit in progress.

DescribeAuditTask

Gets information about a Device Defender audit.

Synopsis

```
aws iot describe-audit-task \  
  --task-id <value> \  
  [--cli-input-json <value>] \  
  [--generate-cli-skeleton]
```

`cli-input-json` format

```
{  
    "taskId": "string"
```

}

cli-input-json Fields

Name	Type	Description
taskId	string length- max:40 min:1 pattern: [a-zA-Z0-9-]+	The ID of the audit whose information you want to get.

Output

```
{
  "taskStatus": "string",
  "taskType": "string",
  "taskStartTime": "timestamp",
  "taskStatistics": {
    "totalChecks": "integer",
    "inProgressChecks": "integer",
    "waitingForDataCollectionChecks": "integer",
    "compliantChecks": "integer",
    "nonCompliantChecks": "integer",
    "failedChecks": "integer",
    "canceledChecks": "integer"
  },
  "scheduledAuditName": "string",
  "auditDetails": {
    "string": {
      "checkRunStatus": "string",
      "checkCompliant": "boolean",
      "totalResourcesCount": "long",
      "nonCompliantResourcesCount": "long",
      "errorCode": "string",
      "message": "string"
    }
  }
}
```

CLI output fields

Name	Type	Description
taskStatus	string	The status of the audit: one of IN_PROGRESS, COMPLETED, FAILED, or CANCELED. enum: IN_PROGRESS COMPLETED FAILED CANCELED
taskType	string	The type of audit: ON_DEMAND_AUDIT_TASK or SCHEDULED_AUDIT_TASK. enum: ON_DEMAND_AUDIT_TASK SCHEDULED_AUDIT_TASK

Name	Type	Description
taskStartTime	timestamp	The time the audit started.
taskStatistics	TaskStatistics	Statistical information about the audit.
totalChecks	integer	The number of checks in this audit.
inProgressChecks	integer	The number of checks in progress.
waitingForDataCollectionChecks	integer	The number of checks waiting for data collection.
compliantChecks	integer	The number of checks that found compliant resources.
nonCompliantChecks	integer	The number of checks that found noncompliant resources.
failedChecks	integer	The number of checks.
canceledChecks	integer	The number of checks that did not run because the audit was canceled.
scheduledAuditName	string length- max:128 min:1 pattern: [a-zA-Z0-9_-]+	The name of the scheduled audit (only if the audit was a scheduled audit).
auditDetails	map	Detailed information about each check performed during this audit.
checkRunStatus	string enum: IN_PROGRESS WAITING_FOR_DATA_COLLECTION CANCELED COMPLETED_COMPLIANT COMPLETED_NON_COMPLIANT FAILED	The completion status of this check, one of IN_PROGRESS, WAITING_FOR_DATA_COLLECTION, CANCELED, COMPLETED_COMPLIANT, COMPLETED_NON_COMPLIANT, or FAILED. enum: IN_PROGRESS WAITING_FOR_DATA_COLLECTION CANCELED COMPLETED_COMPLIANT COMPLETED_NON_COMPLIANT FAILED
checkCompliant	boolean	True if the check completed and found all resources compliant.
totalResourcesCount	long	The number of resources on which the check was performed.

Name	Type	Description
nonCompliantResourcesCount	long	The number of resources that the check found noncompliant.
errorCode	string	The code of any error encountered when performing this check during this audit. One of INSUFFICIENT_PERMISSIONS or AUDIT_CHECK_DISABLED.
message	string length- max:2048	The message associated with any error encountered when performing this check during this audit.

Errors

`InvalidRequestException`

The contents of the request were invalid.

`ResourceNotFoundException`

The specified resource does not exist.

`ThrottlingException`

The rate exceeds the limit.

`InternalFailureException`

An unexpected error has occurred.

ListAuditTasks

Lists the Device Defender audits that have been performed during a given time period.

Synopsis

```
aws iot list-audit-tasks \
  --start-time <value> \
  --end-time <value> \
  [--task-type <value>] \
  [--task-status <value>] \
  [--next-token <value>] \
  [--max-results <value>] \
  [--cli-input-json <value>] \
  [--generate-cli-skeleton]
```

`cli-input-json` format

```
{
  "startTime": "timestamp",
  "endTime": "timestamp",
  "taskType": "string",
  "taskStatus": "string",
  "nextToken": "string",
  "maxResults": "integer"
```

}

cli-input-json Fields

Name	Type	Description
startTime	timestamp	The beginning of the time period. Audit information is retained for a limited time (180 days). Requesting a start time prior to what is retained results in an <code>InvalidRequestException</code> .
endTime	timestamp	The end of the time period.
taskType	string	A filter to limit the output to the specified type of audit: can be one of <code>ON_DEMAND_AUDIT_TASK</code> or <code>SCHEDULED_AUDIT_TASK</code> . enum: <code>ON_DEMAND_AUDIT_TASK</code> <code>SCHEDULED_AUDIT_TASK</code>
taskStatus	string	A filter to limit the output to audits with the specified completion status: can be one of <code>IN_PROGRESS</code> , <code>COMPLETED</code> , <code>FAILED</code> , or <code>CANCELED</code> . enum: <code>IN_PROGRESS</code> <code>COMPLETED</code> <code>FAILED</code> <code>CANCELED</code>
nextToken	string	The token for the next set of results.
maxResults	integer range- max:250 min:1	The maximum number of results to return at one time. The default is 25.

Output

```
{
  "tasks": [
    {
      "taskId": "string",
      "taskStatus": "string",
      "taskType": "string"
    }
  ],
  "nextToken": "string"
}
```

CLI output fields

Name	Type	Description
tasks	list member: AuditTaskMetadata java class: java.util.List	The audits that were performed during the specified time period.
taskId	string length- max:40 min:1 pattern: [a-zA-Z0-9-]+	The ID of this audit.
taskStatus	string	The status of this audit: one of IN_PROGRESS, COMPLETED, FAILED, or CANCELED. enum: IN_PROGRESS COMPLETED FAILED CANCELED
taskType	string	The type of this audit: one of ON_DEMAND_AUDIT_TASK or SCHEDULED_AUDIT_TASK. enum: ON_DEMAND_AUDIT_TASK SCHEDULED_AUDIT_TASK
nextToken	string	A token that can be used to retrieve the next set of results, or null if there are no additional results.

Errors

InvalidRequestException

The contents of the request were invalid.

ThrottlingException

The rate exceeds the limit.

InternalFailureException

An unexpected error has occurred.

CancelAuditTask

Cancels an audit that is in progress. The audit can be either scheduled or on-demand. If the audit is not in progress, an InvalidRequestException occurs.

Synopsis

```
aws iot cancel-audit-task \
--task-id <value> \
```

```
[--cli-input-json <value>] \  
[--generate-cli-skeleton]
```

cli-input-json format

```
{  
    "taskId": "string"  
}
```

cli-input-json Fields

Name	Type	Description
taskId	string length- max:40 min:1 pattern: [a-zA-Z0-9-]+	The ID of the audit you want to cancel. You can only cancel an audit that is IN_PROGRESS.

Output

None

Errors

ResourceNotFoundException

The specified resource does not exist.

InvalidRequestException

The contents of the request were invalid.

ThrottlingException

The rate exceeds the limit.

InternalFailureException

An unexpected error has occurred.

Check audit results

Use `ListAuditFindings` to see the results of an audit. You can filter the results by the type of check, a specific resource, or the time of the audit. You can use this information to mitigate any problems that were found.

You can define mitigation actions and apply them to the findings from your audit. For more information, see [Mitigation actions \(p. 929\)](#).

ListAuditFindings

Lists the findings (results) of a Device Defender audit or of the audits performed during a specified time period. (Findings are retained for 180 days.)

Synopsis

```
aws iot list-audit-findings \  
[--task-id <value>] \  

```

```
[--check-name <value>] \
[--resource-identifier <value>] \
[--max-results <value>] \
[--next-token <value>] \
[--start-time <value>] \
[--end-time <value>] \
[--cli-input-json <value>] \
[--generate-cli-skeleton]
```

cli-input-json format

```
{
    "taskId": "string",
    "checkName": "string",
    "resourceIdentifier": {
        "deviceCertificateId": "string",
        "caCertificateId": "string",
        "cognitoIdentityPoolId": "string",
        "clientId": "string",
        "policyVersionIdentifier": {
            "policyName": "string",
            "policyVersionId": "string"
        },
        "roleAliasArn": "string",
        "account": "string"
    },
    "maxResults": "integer",
    "nextToken": "string",
    "startTime": "timestamp",
    "endTime": "timestamp"
}
```

cli-input-json Fields

Name	Type	Description
taskId	string length- max:40 min:1 pattern: [a-zA-Z0-9-]+	A filter to limit results to the audit with the specified ID. You must specify either the taskId or the startTime and endTime, but not both.
checkName	string	A filter to limit results to the findings for the specified audit check.
resourceIdentifier	ResourceIdentifier	Information that identifies the noncompliant resource.
deviceCertificateId	string length- max:64 min:64 pattern: (0x)?[a-fA-F0-9]+	The ID of the certificate attached to the resource.
caCertificateId	string length- max:64 min:64 pattern: (0x)?[a-fA-F0-9]+	The ID of the CA certificate used to authorize the certificate.

Name	Type	Description
cognitoidentityPoolId	string	The ID of the Amazon Cognito identity pool.
clientId	string	The client ID.
policyVersionIdentifier	PolicyVersionIdentifier	The version of the policy associated with the resource.
policyName	string length- max:128 min:1 pattern: [w+=,.@-]+	The name of the policy.
policyVersionId	string pattern: [0-9]+	The ID of the version of the policy associated with the resource.
roleAliasArn	string	The ARN of the role alias that has overly permissive actions. length- max:2048 min:1
account	string length- max:12 min:12 pattern: [0-9]+	The account with which the resource is associated.
maxResults	integer range- max:250 min:1	The maximum number of results to return at one time. The default is 25.
nextToken	string	The token for the next set of results.
startTime	timestamp	A filter to limit results to those found after the specified time. You must specify either the startTime and endTime or the taskId, but not both.
endTime	timestamp	A filter to limit results to those found before the specified time. You must specify either the startTime and endTime or the taskId, but not both.

Output

```
{
  "findings": [
    {
      "taskId": "string",
      "checkName": "string",
      "taskStartTime": "timestamp",
      "findingTime": "timestamp",
      "severity": "string",
      "status": "string"
    }
  ]
}
```

```

"nonCompliantResource": {
    "resourceType": "string",
    "resourceIdentifier": {
        "deviceCertificateId": "string",
        "caCertificateId": "string",
        "cognitoIdentityPoolId": "string",
        "clientId": "string",
        "policyVersionIdentifier": {
            "policyName": "string",
            "policyVersionId": "string"
        },
        "account": "string"
    },
    "additionalInfo": {
        "string": "string"
    }
},
"relatedResources": [
    {
        "resourceType": "string",
        "resourceIdentifier": {
            "deviceCertificateId": "string",
            "caCertificateId": "string",
            "cognitoIdentityPoolId": "string",
            "clientId": "string",

            "iamRoleArn": "string",

            "policyVersionIdentifier": {
                "policyName": "string",
                "policyVersionId": "string"
            },
            "account": "string"
        },
        "roleAliasArn": "string",

        "additionalInfo": {
            "string": "string"
        }
    }
],
"reasonForNonCompliance": "string",
"reasonForNonComplianceCode": "string"
}
],
"nextToken": "string"
}
}

```

CLI output fields

Name	Type	Description
findings	list member: AuditFinding	The findings (results) of the audit.
taskId	string length- max:40 min:1 pattern: [a-zA-Z0-9-]+	The ID of the audit that generated this result (finding).

Name	Type	Description
checkName	string	The audit check that generated this result.
taskStartTime	timestamp	The time the audit started.
findingTime	timestamp	The time the result (finding) was discovered.
severity	string	The severity of the result (finding). enum: CRITICAL HIGH MEDIUM LOW
nonCompliantResource	NonCompliantResource	The resource that was found to be noncompliant with the audit check.
resourceType	string	The type of the noncompliant resource. enum: DEVICE_CERTIFICATE CA_CERTIFICATE IOT_POLICY COGNITO_IDENTITY_POOL CLIENT_ID ACCOUNT_SETTINGS
resourceIdentifier	ResourceIdentifier	Information that identifies the noncompliant resource.
deviceCertificateId	string length- max:64 min:64 pattern: (0x)?[a-fA-F0-9]+	The ID of the certificate attached to the resource.
caCertificateId	string length- max:64 min:64 pattern: (0x)?[a-fA-F0-9]+	The ID of the CA certificate used to authorize the certificate.
cognitoIdentityPoolId	string	The ID of the Amazon Cognito identity pool.
clientId	string	The client ID.
policyVersionIdentifier	PolicyVersionIdentifier	The version of the policy associated with the resource.
policyName	string length- max:128 min:1 pattern: [w+=,.@-]+	The name of the policy.
policyVersionId	string pattern: [0-9]+	The ID of the version of the policy associated with the resource.

Name	Type	Description
account	string length- max:12 min:12 pattern: [0-9]+	The account with which the resource is associated.
additionalInfo	map	Other information about the noncompliant resource.
relatedResources	list member: RelatedResource	The list of related resources.
resourceType	string enum: DEVICE_CERTIFICATE CA_CERTIFICATE IOT_POLICY COGNITO_IDENTITY_POOL CLIENT_ID ACCOUNT_SETTINGS	The type of resource.
resourceIdentifier	ResourceIdentifier	Information that identifies the resource.
deviceCertificateId	string length- max:64 min:64 pattern: (0x)?[a-fA-F0-9]+	The ID of the certificate attached to the resource.
caCertificateId	string length- max:64 min:64 pattern: (0x)?[a-fA-F0-9]+	The ID of the CA certificate used to authorize the certificate.
cognitoIdentityPoolId	string	The ID of the Amazon Cognito identity pool.
clientId	string	The client ID.
policyVersionIdentifier	PolicyVersionIdentifier	The version of the policy associated with the resource.
iamRoleArn	string length- max:2048 min:20	The ARN of the IAM role that has overly permissive actions.
policyName	string length- max:128 min:1 pattern: [w+=.,@-]+	The name of the policy.
policyVersionId	string pattern: [0-9]+	The ID of the version of the policy associated with the resource.

Name	Type	Description
roleAliasArn	string length- max:2048 min:1	The ARN of the role alias that has overly permissive actions.
account	string length- max:12 min:12 pattern: [0-9]+	The account with which the resource is associated.
additionalInfo	map	Other information about the resource.
reasonForNonCompliance	string	The reason the resource was noncompliant.
reasonForNonComplianceCode	string	A code that indicates the reason that the resource was noncompliant.
nextToken	string	A token that can be used to retrieve the next set of results, or <code>null</code> if there are no additional results.

Errors

InvalidRequestException

The contents of the request were invalid.

ThrottlingException

The rate exceeds the limit.

InternalFailureException

An unexpected error has occurred.

Audit finding suppressions

When you run an audit, it reports findings for all non-compliant resources. This means your audit reports include findings for resources where you're working toward mitigating issues and also for resources that are known to be non-compliant, such as test or broken devices. The audit continues to report findings for resources that remain non-compliant in successive audit runs, which may add unwanted information to your reports. Audit finding suppressions enable you to suppress or filter out findings for a defined period of time until the resource is fixed, or indefinitely for a resource associated with a test or broken device.

Note

Mitigation actions won't be available for suppressed audit findings. For more information about mitigation actions, see [Mitigation actions \(p. 929\)](#).

For information about audit finding suppression quotas, see [AWS IoT Device Defender endpoints and quotas](#).

How audit finding suppressions work

When you create an audit finding suppression for a non-compliant resource, your audit reports and notifications behave differently.

Your audit reports will include a new section that lists all the suppressed findings associated with the report. Suppressed findings won't be considered when we evaluate whether an audit check is compliant or not. A suppressed resource count is also returned for each audit check when you use the [describe-audit-task](#) command in the command line interface (CLI).

For audit notifications, suppressed findings aren't considered when we evaluate whether an audit check is compliant or not. A suppressed resource count is also included in each audit check notification AWS IoT Device Defender publishes to Amazon CloudWatch and Amazon Simple Notification Service (Amazon SNS).

How to use audit finding suppressions in the console

To suppress a finding from an audit report

The following procedure shows you how to create an audit finding suppression in the AWS IoT console.

1. In the [AWS IoT console](#), in the navigation pane, expand **Defend**, and then choose **Audit**, **Results**.
2. Select an audit report you'd like to review.

The screenshot shows the AWS IoT Device Defender Audit Results page. The left sidebar has sections for Monitor, Activity, Onboard, Manage, Greengrass, Secure, Defend (with Intro), Audit (with Results, Schedules, Action executions, Finding suppressions), Detect, Mitigation actions (new), and Settings. The main content area shows a table titled "Audit results (10+)" with columns Name, Date, Status, and Summary. The table lists 10 audit findings, mostly marked as "Not compliant". One entry is marked as "Compliant". The status column includes icons for Not compliant (red triangle), Compliant (green circle), and Pending (blue square). The summary column indicates 1 of 14 non-compliant findings.

Name	Date	Status	Summary
On-demand	July 28, 2020, 14:14:18 (UTC-0700)	⚠️ Not compliant	1 of 14 non-compliant
On-demand	July 28, 2020, 11:55:43 (UTC-0700)	🟢 Compliant	14 of 14 completed
AWSIoTDeviceDefenderDailyAudit	July 28, 2020, 05:12:39 (UTC-0700)	⚠️ Not compliant	1 of 14 non-compliant
AWSIoTDeviceDefenderDailyAudit	July 27, 2020, 05:12:39 (UTC-0700)	⚠️ Not compliant	1 of 14 non-compliant
AWSIoTDeviceDefenderDailyAudit	July 26, 2020, 05:12:39 (UTC-0700)	⚠️ Not compliant	1 of 14 non-compliant
AWSIoTDeviceDefenderDailyAudit	July 25, 2020, 05:12:39 (UTC-0700)	⚠️ Not compliant	1 of 14 non-compliant
AWSIoTDeviceDefenderDailyAudit	July 24, 2020, 05:12:39 (UTC-0700)	⚠️ Not compliant	1 of 14 non-compliant
AWSIoTDeviceDefenderDailyAudit	July 23, 2020, 05:12:39 (UTC-0700)	⚠️ Not compliant	1 of 14 non-compliant
AWSIoTDeviceDefenderDailyAudit	July 22, 2020, 05:12:39 (UTC-0700)	⚠️ Not compliant	1 of 14 non-compliant
AWSIoTDeviceDefenderDailyAudit	July 21, 2020, 05:12:39 (UTC-0700)	⚠️ Not compliant	1 of 14 non-compliant

3. In the **Non-compliant checks** section, under **Check name**, choose the audit check that you're interested in.

The screenshot shows the AWS IoT Device Defender Audit Report interface. At the top, a breadcrumb navigation path is visible: AWS IoT > Device Defender > Audit > Audit Results > Audit Report. The main title is "Audit Report" with the subtitle "On-demand - July 28, 2020, 14:14:18 (UTC-0700)". Below this, there are two tables: one for "Non-compliant checks (1 of 14)" and one for "Compliant checks (13 of 14)". The "Non-compliant checks" table has one row: "Logging disabled" (Severity: Low, Non-compliant resources: 1, % Resources: 100%, Mitigation: Logging disabled). The "Compliant checks" table has 13 rows, all of which are marked as "Scanned".

Non-compliant checks (1 of 14)				
Check name	Severity	Non-compliant resources	% Resources	Mitigation
Logging disabled	Low	1	100%	Logging disabled ⓘ

Compliant checks (13 of 14)		
Check name	Severity	Scanned ⓘ
Authenticated Cognito role overly permissive	Critical	0
CA certificate key quality	Critical	0
CA certificate revoked but device certificates still active	Critical	0
Device certificate key quality	Critical	0
Device certificate shared	Critical	0
IoT policies overly permissive	Critical	0
Role alias overly permissive	Critical	0
Unauthenticated Cognito role overly permissive	Critical	0
Conflicting MQTT client IDs	High	0
CA certificate expiring	Medium	0
Device certificate expiring	Medium	0
Revoked device certificate still active	Medium	0
Role alias allows access to unused services	Medium	0

4. On the audit check details screen, if there are findings you don't want to see, select the option button next to the finding. Next, choose **Actions**, and then choose the amount of time you'd like your audit finding suppression to persist.

Note

In the console, you can select *1 week*, *1 month*, *3 months*, *6 months*, or *Indefinitely* as expiration dates for your audit finding suppression. If you want to set a specific expiration date, you can do so only in the CLI or API. Audit finding suppressions can also be canceled anytime regardless of expiration date.

The screenshot shows the AWS IoT Device Defender Audit Findings page. At the top, there is a breadcrumb navigation: AWS IoT > Device Defender > Audit > Audit Results > Audit Report > Audit Findings. Below the breadcrumb, the title "Audit Findings" is displayed, followed by a note "Logging disabled". A section titled "1 account non-compliant" contains a "Mitigation" link: "Enable CloudWatch Logs.". Below this, a table lists a single "Non-compliant account (1)". The table has columns: Finding, Reason, and Account settings. The first row shows a blue circular icon, the ID "417b2f816eac7a2e40fdb0bc709b01a2", the reason "Logging disabled on account.", and the account ID "765219403047". To the right of the table is a "Actions" dropdown menu with options: Start mitigation actions, Suppress Finding, 1 week, 1 month, 3 months, 6 months, and Indefinitely.

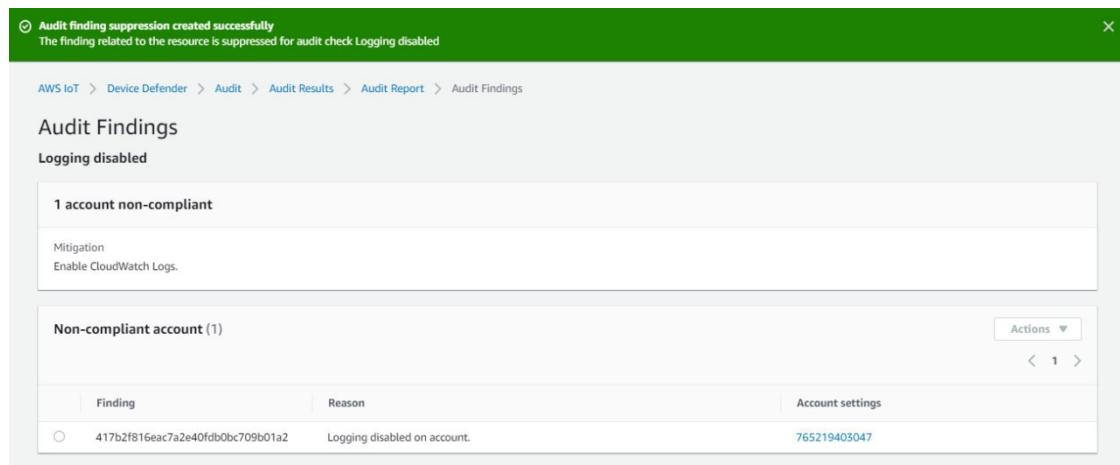
5. Confirm the suppression details, and then choose **Enable suppression**.

The screenshot shows a "Confirm suppression" dialog box. The title is "Confirm suppression" with a close button "X" in the top right. The main content area contains the following details:

- Please verify the details of the audit finding suppression
- Check name: Logging disabled
- Account settings: 765219403047
- Expiration period: 3 months
- Expiration date: 2020-10-28T21:25:41.100Z

At the bottom of the dialog are two buttons: "Cancel" and a large orange "Enable suppression" button.

6. After you've created the audit finding suppression, a banner appears confirming your audit finding suppression was created.



To view your suppressed findings in an audit report

1. In the [AWS IoT console](#), in the navigation pane, expand **Defend**, and then choose **Audit**, **Results**.
2. Select an audit report you'd like to review.
3. In the **Suppressed findings** section, view which audit findings have been suppressed for your chosen audit report.

The screenshot shows the AWS IoT Device Defender Audit Report interface. The left sidebar navigation includes sections like Monitor, Activity, Onboard, Manage, Greengrass, Secure, Defend (with Intro selected), Audit (with Results, Schedules, Action executions, Finding suppressions selected), Detect (with Mitigation actions), Settings, Act (with Test), Software, Settings, Learn, and Documentation. The main content area displays an Audit Report for an on-demand audit on July 28, 2020, at 11:55:43 (UTC-0700). It lists 'Audit findings' and 'Compliant checks (14 of 14)' with details like check name, severity, and scan status. It also shows 'Suppressed findings (1)' with a single entry for 'Logging disabled'.

Check name	Severity	Scanned
Authenticated Cognito role overly permissive	Critical	0
CA certificate key quality	Critical	0
CA certificate revoked but device certificates still active	Critical	0
Device certificate key quality	Critical	0
Device certificate shared	Critical	0
IoT policies overly permissive	Critical	0
Role alias overly permissive	Critical	0
Unauthenticated Cognito role overly permissive	Critical	0
Conflicting MQTT client IDs	High	0
CA certificate expiring	Medium	0
Device certificate expiring	Medium	0
Revoked device certificate still active	Medium	0
Role alias allows access to unused services	Medium	0
Logging disabled	Low	1

Check name	Finding	Reason	Resource identifier
Logging disabled	755a27914fb2ca24a8b3d47ef3563726	Logging disabled on account.	765219403047

To list your audit finding suppressions

- In the [AWS IoT console](#), in the navigation pane, expand **Defend**, and then choose **Audit**, **Finding suppressions**.

The screenshot shows the AWS IoT Device Defender interface. In the navigation pane on the left, under the 'Defend' section, the 'Audit' option is expanded, and 'Finding suppressions' is selected. The main content area displays a table titled 'Audit finding suppressions (1)'. The table has columns for 'Resource identifier', 'Check name', 'Expiration date', and 'Description'. A single row is shown with the resource identifier '765219403047', check name 'Logging disabled', expiration date 'October 28, 2020, 14:26:53 (UTC-0700)', and an empty description field.

To edit your audit finding suppression

1. In the [AWS IoT console](#), in the navigation pane, expand **Defend**, and then choose **Audit**, **Finding suppressions**.
2. Select the option button next to the audit finding suppression you'd like to edit. Next, choose **Actions**, **Edit**.
3. On the **Edit audit finding suppression** window, you can change the **Suppression duration** or **Description (optional)**.

Edit audit finding suppression X

Suppressing an audit finding on a specified resource means that the finding related to the resource for the specified audit check will no longer be flagged as non-compliant.

Audit check

Logging disabled

Resource identifier

Account ID

765219403047

Suppression duration

The expiration date is October 28, 2020, 14:26:53 (UTC-0700). Select a different duration to change this.

6 months

Description (optional)

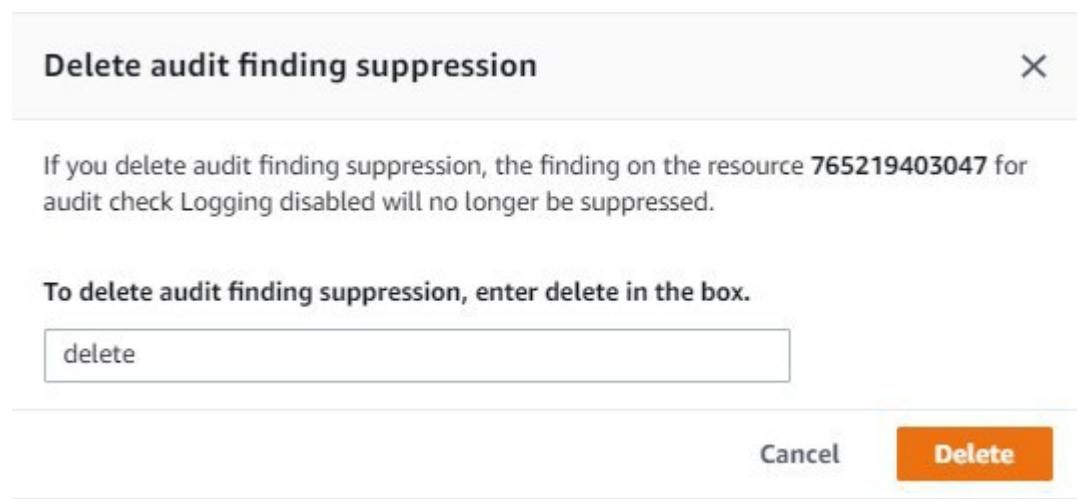
Suppresses "Logging disabled" check because I don't want to enable logging for now.

Cancel **Save**

4. After you've made your changes, choose **Save**. The **Finding suppressions** window opens.

To delete an audit finding suppression

1. In the [AWS IoT console](#), in the navigation pane, expand **Defend**, and then choose **Audit**, **Finding suppressions**.
2. Select the option button next to the audit finding suppression you'd like to delete, and then choose **Actions**, **Delete**.
3. On the **Delete audit finding suppression** window, enter `delete` in the text box to confirm your deletion, and then choose **Delete**. The **Finding suppressions** window opens.



How to use audit finding suppressions in the CLI

You can use the following CLI commands to create and manage audit finding suppressions.

- [create-audit-suppression](#)
- [describe-audit-suppression](#)
- [update-audit-suppression](#)
- [delete-audit-suppression](#)
- [list-audit-suppressions](#)

The `resource-identifier` you input depends on the `check-name` you're suppressing findings for. The following table details which checks require which `resource-identifier` for creating and editing suppressions.

Note

The suppression commands do not indicate turning off an audit. Audits will still run on your AWS IoT devices. Suppressions are only applicable to the audit findings.

check-name	resource-identifier
AUTHENTICATE_COGNITO_ROLE_OVERLY_PERMISSIVE_CHECK	deviceIdentityPoolId
CA_CERT_APPROACHING_EXPIRATION_CHECK	caCertificateId
CA_CERTIFICATE_KEY_QUALITY_CHECK	caCertificateId
CONFLICTING_CLIENT_IDS_CHECK	clientId
DEVICE_CERT_APPROACHING_EXPIRATION_CHECK	deviceCertificateId
DEVICE_CERTIFICATE_KEY_QUALITY_CHECK	deviceCertificateId
DEVICE_CERTIFICATE_SHARED_CHECK	deviceCertificateId
IOT_POLICY_OVERLY_PERMISSIVE_CHECK	policyVersionIdentifier
IOT_ROLE_ALIAS_ALLows_ACCESS_TO_UNUSED_SERVICES_CHECK	serviceSaCheck

check-name	resource-identifier
IOT_ROLE_ALIAS_OVERLY_PERMISSIVE_CHECK	roleAliasArn
LOGGING_DISABLED_CHECK	account
REVOKE_CA_CERT_CHECK	caCertificateId
REVOKE_DEVICE_CERT_CHECK	deviceCertificateId
UNAUTHENTICATED_COGNITO_ROLE_OVERLY_PERMISSIVE_CHECK	identityPoolId

To create and apply an audit finding suppression

The following procedure shows you how to create an audit finding suppression in the AWS CLI.

- Use the `create-audit-suppression` command to create an audit finding suppression. The following example creates an audit finding suppression for AWS account `123456789012` on the basis of the check **Logging disabled**.

```
aws iot create-audit-suppression \
  --check-name LOGGING_DISABLED_CHECK \
  --resource-identifier account=123456789012 \
  --client-request-token 28ac32c3-384c-487a-a368-c7bbd481f554 \
  --suppress-indefinitely \
  --description "Suppresses logging disabled check because I don't want to enable
logging for now."
```

There is no output for this command.

Audit finding suppressions APIs

The following APIs can be used to create and manage audit finding suppressions.

- [CreateAuditSuppression](#)
- [DescribeAuditSuppression](#)
- [UpdateAuditSuppression](#)
- [DeleteAuditSuppression](#)
- [ListAuditSuppressions](#)

To filter *for* specific audit findings, you can use the [ListAuditFindings](#) API.

Detect

AWS IoT Device Defender Detect lets you identify unusual behavior that might indicate a compromised device by monitoring the behavior of your devices. Using a combination of cloud-side metrics (from AWS IoT) and device-side metrics (from agents that you install on your devices) you can detect:

- Changes in connection patterns.
- Devices that communicate to unauthorized or unrecognized endpoints.
- Changes in inbound and outbound device traffic patterns.

You create security profiles, which contain definitions of expected device behaviors, and assign them to a group of devices or to all the devices in your fleet. AWS IoT Device Defender Detect uses these security profiles to detect anomalies and send alarms through Amazon CloudWatch metrics and Amazon Simple Notification Service notifications.

AWS IoT Device Defender Detect can detect security issues frequently found in connected devices:

- Traffic from a device to a known malicious IP address or to an unauthorized endpoint that indicates a potential malicious command and control channel.
- Anomalous traffic, such as a spike in outbound traffic, that indicates a device is participating in a DDoS.
- Devices with remote management interfaces and ports that are remotely accessible.
- A spike in the rate of messages sent to your account (for example, from a rogue device that can result in excessive per-message charges).

Use cases:

Measure attack surface

You can use AWS IoT Device Defender Detect to measure the attack surface of your devices. For example, you can identify devices with service ports that are often the target of attack campaigns (telnet service running on ports 23/2323, SSH service running on port 22, HTTP/S services running on ports 80/443/8080/8081). While these service ports might have legitimate reasons to be used on the devices, they are also usually part of the attack surface for adversaries and carry associated risks. After AWS IoT Device Defender Detect alarms you to the attack surface, you can minimize it (by eliminating unused network services) or run additional assessments to identify security weaknesses (for example, telnet configured with common, default, or weak passwords).

Detect device behavioral anomalies with possible security root causes

You can use AWS IoT Device Defender Detect to alarm you to unexpected device behavioral metrics (the number of open ports, number of connections, an unexpected open port, connections to unexpected IP addresses) that might indicate a security breach. For example, a higher than expected number of TCP connections might indicate a device is being used for a DDoS attack. A process listening on a port other than the one you expect might indicate a backdoor installed on a device for remote control. You can use AWS IoT Device Defender Detect to probe the health of your device fleets and verify your security assumptions (for example, no device is listening on port 23 or 2323).

You can enable machine learning (ML)-based threat detection to automatically identify potential threats.

Detect an incorrectly configured device

A spike in the number or size of messages sent from a device to your account might indicate an incorrectly configured device. Such a device might increase your per-message charges. Similarly, a device with many authorization failures might require a reconfigured policy.

Monitoring the behavior of unregistered devices

AWS IoT Device Defender Detect makes it possible to identify unusual behaviors for devices that are not registered in the AWS IoT registry. You can define security profiles that are specific to one of the following target types:

- All devices
- All registered devices (things in the AWS IoT registry)
- All unregistered devices
- Devices in a thing group

A security profile defines a set of expected behaviors for devices in your account and specifies the actions to take when an anomaly is detected. Security profiles should be attached to the most specific targets to give you granular control over which devices are being evaluated against that profile.

Unregistered devices must provide a consistent MQTT client identifier or thing name (for devices that report device metrics) over the device lifetime so all violations and metrics are attributed to the same device.

Important

Messages reported by devices are rejected if the thing name contains control characters or if the thing name is longer than 128 bytes of UTF-8 encoded characters.

Security use cases

This section describes the different types of attacks that threaten your device fleet and the recommended metrics you can use to monitor for these attacks. We recommend using metric anomalies as a starting point to investigate security issues, but you should not base your determination of any security threats solely on a metric anomaly.

To investigate an anomaly alarm, correlate the alarm details with other contextual information such as device attributes, device metric historical trends, Security Profile metric historical trends, custom metrics, and logs to determine if a security threat is present.

Cloud-side use cases

Device Defender can monitor the following use cases on the AWS IoT cloud side.

Intellectual property theft:

Intellectual property theft involves stealing a person's or companies' intellectual properties, including trade secrets, hardware, or software. It often occurs during the manufacturing stage of devices. Intellectual property theft can come in the form of piracy, device theft, or device certificate theft. Cloud-based intellectual property theft can occur due to the presence of policies that permit unintended access to IoT resources. You should review your [IoT policies](#) and turn on [Audit overly permissive checks](#) to identify overly permissive policies.

Related metrics:

Metric	Rationale
Source IP	If device is stolen, then its source IP address would fall outside of the normally expected IP address range for devices circulated in a normal supply chain.
Number of messages received	Because an attacker may use a device in cloud-based IP theft, metrics related to message counts or message sizes sent to the device from AWS IoT cloud can spike up, indicating a possible security issue.
Message size	

MQTT-based data exfiltration:

Data exfiltration occurs when a malicious actor carries out an unauthorized data transfer from an IoT deployment or from a device. The attacker launches this type of attacks through MQTT against cloud-side data sources.

Related metrics:

Metric	Rationale
Source IP	If a device is stolen, then its source IP address would fall outside of the normally expected IP address range for devices circulated in a standard supply chain.
Number of messages received	Because an attacker may use a device in a MQTT-based data exfiltration, metrics related to message counts or message sizes sent to the device from AWS IoT cloud can spike up, indicating a possible security issue.
Message size	

Impersonation:

An impersonation attack is where attackers pose as known or trusted entities in an effort to access AWS IoT cloud-side services, applications, data, or engage in command and control of IoT devices.

Related metrics:

Metric	Rationale
Authorization failures	
Connection attempts	
Disconnects	When attackers pose as trusted entities by using stolen identities, connectivity related metrics often spike, as the credentials may no longer be valid or may be used by a trusted device already. Anomalous behaviors in authorization failures, connection attempts, or disconnects point to a potential impersonation scenario.

Cloud Infrastructure abuse:

Abuse to AWS IoT cloud services occurs when publishing or subscribing to topics with a high message volume or with messages in large sizes. Overly permissive policies or device vulnerability exploit for command and control can also cause cloud infrastructure abuse. One of the main objectives of this attack is to increase your AWS bill. You should review your [IoT policies](#) and turn on [Audit overly permissive checks](#) to identify overly permissive policies.

Related metrics:

Metric	Rationale
Number of messages received	
Number of messages sent	The objective of this attack is to increase your AWS bill, metrics that monitor activities like message count, messages received and message size will spike up.
Message size	
Source IP	Suspicious source IP lists may appear, from which attackers generate their messaging volume.

Device-side use cases

Device Defender can monitor the following use cases on your device side.

Denial-of-service attack:

A denial-of-service (DoS) attack is aimed at shutting down a device or network, making the device or network inaccessible to their intended users. DoS attacks block access by flooding the target with traffic, or sending it requests that start a system slow-down or cause the system to fail. Your IoT devices can be used in DoS attacks.

Related metrics:

Metric	Rationale
Packets out	DoS attacks typically involve higher rates of outbound communication from a given device, and depending on the type of DoS attack, there could be an increase in either or both of the numbers of packets out and bytes out.
Bytes out	
Destination IP	If you define the IP addresses/CIDR ranges your devices should communicate with, then an anomaly in destination IP can indicate unauthorized IP communication from your devices.
Listening TCP ports	
Listening TCP port count	
Listening UDP ports	
Listening UDP port count	A DoS attack usually requires a larger command and control infrastructure where malware installed on your devices receives commands and information about who to attack and when to attack. Therefore, in order to receive such information, the malware would typically listen on ports that aren't normally used by your devices.

Lateral threat escalation:

Lateral threat escalation usually begins with an attacker gaining access to one point of a network, for example a connected device. The attacker then tries to increase their level of privileges, or their access to other devices through methods such as stolen credentials or vulnerability exploits.

Related metrics:

Metric	Rationale
Packets out	In typical situations, the attacker would have to run a scan on the local area network in order to perform reconnaissance and identify the available devices in order to narrow down their attack target selection. This kind of scan could result in a spike of bytes and packets out counts.
Bytes out	
Destination IP	If a device is supposed to communicate with a known set of IP addresses or CIDRs, you can identify if it attempts to communicate with an abnormal IP address, which would often be a private IP address on the local network in a lateral threat escalation use case.
Authorization failures	As the attacker tries to increase their level of privileges across an IoT network, they may

Metric	Rationale
	use stolen credentials that have been revoked or have expired, which would cause increased authorization failures.

Data exfiltration or surveillance:

Data exfiltration occurs when malware or a malicious actor carries out an unauthorized data transfer from a device or a network endpoint. Data exfiltration normally serves two purposes for the attacker, obtaining data or intellectual property, or conducting reconnaissance of a network. Surveillance means that malicious code is used to monitor user activities for the purpose of stealing credentials and gathering information. The metrics below can provide a starting point of investigating either type of attacks.

Related metrics:

Metric	Rationale
Packets out	
Bytes out	When data exfiltration or surveillance attacks occur, the attacker would often mirror the data being sent from the device rather than simply redirecting the data, which would be identified by the defender when they don't see the intended data coming. Such mirrored data would increase the total amount of data sent from the device significantly, resulting in a spike of packets and bytes out counts.
Destination IP	When an attacker is using a device in data exfiltration or surveillance attacks, the data would have to be sent to an abnormal IP address controlled by the attacker. Monitoring the destination IP can help identify such an attack.

Cryptocurrency mining

Attackers leverage processing power from devices to mine cryptocurrency. Crypto-mining is a computationally intensive process, typically requiring network communication with other mining peers and pools.

Related metrics:

Metric	Rationale
Destination IP	Network communication is typically a requirement during cryptomining. Having a tightly controlled list of IP addresses the device should communicate with can help identify unintended communication on a device, like cryptocurrency mining.
CPU usage custom metric	Cryptocurrency mining requires intensive computation resulting in high utilization of the device CPU. If you choose to collect and monitor this metric, a higher-than-normal CPU

Metric	Rationale
	usage could be an indicator of crypto-mining activities.

Command and control, malware and ransomware

Malware or ransomware restricts your control over your devices, and limits your device functionality. In the case of a ransomware attack, data access would be lost due to encryption the ransomware uses.

Related metrics:

Metric	Rationale
Destination IP	Network or remote attacks represent a large portion of attacks on IoT devices. A tightly controlled list of IP addresses the device should communicate with can help identify abnormal destination IPs resulted from a malware or ransomware attack.
Listening TCP ports	Several malware attacks involve starting a command-and-control server that sends commands to execute on a device. This type of server is critical to a malware or ransomware operation and can be identified by tightly monitoring the open TCP/UDP ports and port counts.
Listening TCP port count	
Listening UDP ports	
Listening UDP port count	

Concepts

metric

AWS IoT Device Defender Detect uses metrics to detect anomalous behavior of devices. AWS IoT Device Defender Detect compares the reported value of a metric with the expected value you provide. These metrics can be taken from two sources: cloud-side metrics and device-side metrics. There are 17 total metrics, 6 of which are supported by ML Detect. For a list of supported metrics for ML Detect, see [Supported metrics \(p. 890\)](#).

Abnormal behavior on the AWS IoT network is detected by using cloud-side metrics such as the number of authorization failures, or the number or size of messages a device sends or receives through AWS IoT.

AWS IoT Device Defender Detect can also collect, aggregate, and monitor metrics data generated by AWS IoT devices (for example, the ports a device is listening on, the number of bytes or packets sent, or the device's TCP connections).

You can use AWS IoT Device Defender Detect with cloud-side metrics alone. To use device-side metrics, you must first deploy the AWS IoT SDK on your AWS IoT connected devices or device gateways to collect the metrics and send them to AWS IoT. See [Sending metrics from devices \(p. 912\)](#).

Security Profile

A Security Profile defines anomalous behaviors for a group of devices (a [thing group \(p. 258\)](#)) or for all devices in your account, and specifies which actions to take when an anomaly is detected. You can use the AWS IoT console or API commands to create a Security Profile and associate it with a

group of devices. AWS IoT Device Defender Detect starts recording security-related data and uses the behaviors defined in the Security Profile to detect anomalies in the behavior of the devices.

behavior

A behavior tells AWS IoT Device Defender Detect how to recognize when a device is doing something anomalous. Any device action that doesn't match a behavior triggers an alert. A Rules Detect behavior consists of a metric and an absolute-value or statistical threshold with an operator (for example, less than or equal to, greater than or equal to), which describe the expected device behavior. An ML Detect behavior consists of a metric and an ML Detect configuration, which set an ML model to learn the normal behavior of devices.

ML model

An ML model is a machine learning model created to monitor each behavior a customer configures. The model trains on metric data patterns from targeted device groups and generates three anomaly confidence thresholds (high, medium, and low) for the metric-based behavior. It infers anomalies based on ingested metric data at the device level. In the context of ML Detect, one ML model is created to evaluate one metric-based behavior. For more information, see [ML Detect \(p. 888\)](#).

confidence level

ML Detect supports three confidence levels: High, Medium, and Low. High confidence means low sensitivity in anomalous behavior evaluation and frequently a lower number of alarms. Medium confidence means medium sensitivity and Low confidence means high sensitivity and frequently a higher number of alarms.

dimension

You can define a dimension to adjust the scope of a behavior. For example, you can define a topic filter dimension that applies a behavior to MQTT topics that match a pattern. For information about defining a dimension for use in a Security Profile, see [CreateDimension](#).

alarm

When an anomaly is detected, an alarm notification can be sent through a CloudWatch metric (see [Using AWS IoT metrics \(p. 407\)](#)) or an SNS notification. An alarm notification is also displayed in the AWS IoT console along with information about the alarm, and a history of alarms for the device. An alarm is also sent when a monitored device stops exhibiting anomalous behavior or when it had been causing an alarm but stops reporting for an extended period.

alarm verification state

After an alarm has been created, you can verify the alarm as True positive, Benign positive, False positive, or Unknown. You can also add a description to your alarm verification state. You can view, organize, and filter AWS IoT Device Defender alarms by using one of the four verification states. You can use alarm verification states and related descriptions to inform members of your team. This helps your team to take follow-up actions, for example, performing mitigation actions on True positive alarms, skipping Benign positive alarms, or continuing investigation on Unknown alarms. The default verification state for all alarms is Unknown.

alarm suppression

Manage Detect alarm SNS notifications by setting behavior notification to on or suppressed. Suppressing alarms doesn't stop Detect from performing device behavior evaluations; Detect continues to flag anomalous behaviors as violation alarms. However, suppressed alarms wouldn't be forwarded for SNS notification. They can only be accessed through the AWS IoT console or API.

Behaviors

A Security Profile contains a set of behaviors. Each behavior contains a metric that specifies the normal behavior for a group of devices or for all devices in your account. Behaviors fall into two categories: Rules

Detect behaviors and ML Detect behaviors. With Rules Detect behaviors, you define how your devices should behave whereas ML Detect uses ML models built on historical device data to evaluate how your devices should behave.

A Security Profile can be one of two threshold types: **ML** or **Rule-based**. ML Security Profiles automatically detect device-level operational and security anomalies across your fleet by learning from past data. Rule-based Security Profiles require that you manually set static rules to monitor your device behaviors.

The following describes some of the fields that are used in the definition of a behavior:

Common to Rules Detect and ML Detect

name

The name for the behavior.

metric

The name of the metric used (that is, what is measured by the behavior).

consecutiveDatapointsToAlarm

If a device is in violation of the behavior for the specified number of consecutive data points, an alarm occurs. If not specified, the default is 1.

consecutiveDatapointsToClear

If an alarm has occurred and the offending device is no longer in violation of the behavior for the specified number of consecutive data points, the alarm is cleared. If not specified, the default is 1.

threshold type

A Security Profile can be one of two threshold types: ML or Rules based. ML Security Profiles automatically detect device-level operational and security anomalies across your fleet by learning from past data. Rule-based Security Profiles require that you manually set static rules to monitor your device behaviors.

alarm suppressions

Manage Detect alarm SNS notifications by setting behavior notification to on or suppressed. Suppressing alarms doesn't stop Detect from performing device behavior evaluations; Detect continues to flag anomalous behaviors as violation alarms. However, suppressed alarms aren't forwarded for SNS notification. They can be accessed only through the AWS IoT console or API.

Rules Detect

dimension

You can define a dimension to adjust the scope of a behavior. For example, you can define a topic filter dimension that applies a behavior to MQTT topics that match a pattern. To define a dimension for use in a Security Profile, see [CreateDimension](#). Applies to Rules Detect only.

criteria

The criteria that determine if a device is behaving normally in regard to the **metric**.

comparisonOperator

The operator that relates the thing measured (**metric**) to the criteria (**value** or **statisticalThreshold**).

Possible values are: "less-than", "less-than-equals", "greater-than", "greater-than-equals", "in-cidr-set", "not-in-cidr-set", "in-port-set", and "not-in-port-set". Not all operators are valid for

every metric. Operators for CIDR sets and ports are only for use with metrics involving such entities.

value

The value to be compared with the metric. Depending on the type of metric, this should contain a count (a value), cidrs (a list of CIDRs), or ports (a list of ports).

statisticalThreshold

The statistical threshold by which a behavior violation is determined. This field contains a statistic field that has the following possible values: "p0", "p0.1", "p0.01", "p1", "p10", "p50", "p90", "p99", "p99.9", "p99.99", or "p100".

This statistic indicates a percentile. It resolves to a value by which compliance with the behavior is determined. Metrics are collected one or more times over the specified duration (durationSeconds) from all reporting devices associated with this Security Profile, and percentiles are calculated based on that data. After that, measurements are collected for a device and accumulated over the same duration. If the resulting value for the device falls above or below (comparisonOperator) the value associated with the percentile specified, then the device is considered to be in compliance with the behavior. Otherwise, the device is in violation of the behavior.

A percentile indicates the percentage of all the measurements considered that fall below the associated value. For example, if the value associated with "p90" (the 90th percentile) is 123, then 90% of all measurements were below 123.

durationSeconds

Use this to specify the period of time over which the behavior is evaluated, for those criteria that have a time dimension (for example, NUM_MESSAGES_SENT). For a statisticalThreshold metric comparison, this is the time period during which measurements are collected for all devices to determine the statisticalThreshold values, and then for each device to determine how its behavior ranks in comparison.

ML Detect

ML Detect confidence

ML Detect supports three confidence levels: High, Medium, and Low. High confidence means low sensitivity in anomalous behavior evaluation and frequently a lower number of alarms, Medium confidence means medium sensitivity, and Low confidence means high sensitivity and frequently a higher number of alarms.

ML Detect

With machine learning Detect (ML Detect), you create Security Profiles that use machine learning to learn expected device behaviors by automatically creating models based on historical device data, and assign these profiles to a group of devices or all the devices in your fleet. AWS IoT Device Defender then identifies anomalies and triggers alarms using the ML models.

For information about how to get started with ML Detect, see [ML Detect guide \(p. 786\)](#).

This chapter contains the following sections:

- [Use cases of ML Detect \(p. 889\)](#)
- [How ML Detect works \(p. 889\)](#)
- [Minimum requirements \(p. 889\)](#)
- [Limitations \(p. 890\)](#)

- [Marking false positives and other verification states in alarms \(p. 890\)](#)
- [Supported metrics \(p. 890\)](#)
- [Service quotas \(p. 891\)](#)
- [ML Detect CLI commands \(p. 891\)](#)
- [ML Detect APIs \(p. 891\)](#)
- [Pause or delete an ML Detect Security Profile \(p. 891\)](#)

Use cases of ML Detect

You can use ML Detect to monitor your fleet devices when it's difficult to set the expected behaviors of devices. For example, to monitor the number of disconnects metric, it might not be clear what is considered an acceptable threshold. In this case, you can enable ML Detect to identify anomalous disconnect metric datapoints based off historical data reported from devices.

Another use case of ML Detect is to monitor device behaviors that change dynamically over time. ML Detect periodically learns the dynamic expected device behaviors based on changing data patterns from devices. For example, device message sent volume could vary between weekdays and weekends, and ML detect will learn this dynamic behavior.

How ML Detect works

Using ML Detect, you can create behaviors to identify operational and security anomalies across [6 cloud-side metrics \(p. 890\)](#) and [7 device-side metrics \(p. 890\)](#). After the initial model training period, ML Detect refreshes the models daily based on the trailing 14 days of data. It monitors datapoints for these metrics with the ML models and triggers an alarm if an anomaly is detected.

ML Detect works best if you attach a Security Profile to a collection of devices with similar expected behaviors. For example, if some of your devices are used at customers' homes and other devices at business offices, the device behavior patterns might differ significantly between the two groups. You can organize the devices into a *home-device* thing group and an *office-device* thing group. For the best anomaly detection efficacy, attach each thing group to a separate ML Detect Security Profile.

While ML Detect is building the initial model, it requires 14 days and a minimum of 25,000 datapoints per metric over the trailing 14-day period to generate a model. Afterwards, it updates the model every day there is a minimum number of metric datapoints. If the minimum requirement isn't met, ML Detect attempts to build the model the next day, and will retry daily for the next 30 days before discontinuing the model for evaluations.

Minimum requirements

For training and creating the initial ML model, ML Detect has the following minimum requirements.

Minimum training period

It takes 14 days for the initial models to be built. After that, the model refreshes every day with metric data from a 14-day trailing period.

Minimum total datapoints

The minimum required datapoints to build an ML model is 25,000 datapoints per metric for the last 14 days. For ongoing training and refreshing of the model, ML Detect requires the minimum datapoints be met from monitored devices. It's roughly the equivalent of the following setups:

- 60 devices connecting and having activity on AWS IoT at 45-minute intervals.
- 40 devices at 30-minute intervals.
- 15 devices at 10-minute intervals.

- 7 devices at 5-minute intervals.

Device group targets

In order for data collection to progress, you must have things in the target thing groups for the Security Profile.

After the initial model is created, ML models refresh every day and require at least 25,000 datapoints for 14-day trailing period.

Limitations

You can't currently use ML Detect with dimensions or with custom metrics. The following metrics are not supported with ML Detect.

Cloud-side metrics not supported with ML Detect:

- [Source IP \(aws:source-ip-address\) \(p. 917\)](#)

Device-side metrics not supported with ML Detect:

- [Destination IPs \(aws:destination-ip-addresses\) \(p. 904\)](#)
- [Listening TCP ports \(aws:listening-tcp-ports\) \(p. 904\)](#)
- [Listening UDP ports \(aws:listening-udp-ports\) \(p. 905\)](#)

Marking false positives and other verification states in alarms

If you verify that an ML Detect alarm is a false positive through your investigation, you can set the verification state of the alarm to False positive. This can help you and your team identify alarms you don't have to respond to. You can also mark alarms as True positive, Benign positive, or Unknown.

You can mark alarms through the [AWS IoT Device Defender console](#) or by using the `PutVerificationStateOnViolation` API action.

Supported metrics

You can use the following cloud-side metrics with ML Detect:

- [Authorization failures \(aws:num-authorization-failures\) \(p. 916\)](#)
- [Connection attempts \(aws:num-connection-attempts\) \(p. 917\)](#)
- [Disconnects \(aws:num-disconnects\) \(p. 918\)](#)
- [Message size \(aws:message-byte-size\) \(p. 912\)](#)
- [Messages sent \(aws:num-messages-sent\) \(p. 913\)](#)
- [Messages received \(aws:num-messages-received\) \(p. 915\)](#)

You can use the following device-side metrics with ML Detect:

- [Bytes out \(aws:all-bytes-out\) \(p. 897\)](#)
- [Bytes in \(aws:all-bytes-in\) \(p. 898\)](#)
- [Listening TCP port count \(aws:num-listening-tcp-ports\) \(p. 899\)](#)
- [Listening UDP port count \(aws:num-listening-udp-ports\) \(p. 901\)](#)
- [Packets out \(aws:all-packets-out\) \(p. 902\)](#)
- [Packets in \(aws:all-packets-in\) \(p. 903\)](#)
- [Established TCP connections count \(aws:num-established-tcp-connections\) \(p. 905\)](#)

Service quotas

For information about ML Detect service quotas and limits, see [AWS IoT Device Defender endpoints and quotas](#).

ML Detect CLI commands

You can use the following CLI commands to create and manage ML Detect.

- [create-security-profile](#)
- [attach-security-profile](#)
- [list-security-profiles](#)
- [describe-security-profile](#)
- [update-security-profile](#)
- [delete-security-profile](#)
- [get-behavior-model-training-summaries](#)
- [list-active-violations](#)
- [list-violation-events](#)

ML Detect APIs

The following APIs can be used to create and manage ML Detect Security Profiles.

- [CreateSecurityProfile](#)
- [AttachSecurityProfile](#)
- [ListSecurityProfiles](#)
- [DescribeSecurityProfile](#)
- [UpdateSecurityProfile](#)
- [DeleteSecurityProfile](#)
- [GetBehaviorModelTrainingSummaries](#)
- [ListActiveViolations](#)
- [ListViolationEvents](#)
- [PutVerificationStateOnViolation](#)

Pause or delete an ML Detect Security Profile

You can pause your ML Detect Security Profile to stop monitoring device behaviors temporarily, or delete your ML Detect Security Profile to stop monitoring device behaviors for an extended period of time.

Pause ML Detect Security Profile by using the console

To pause an ML Detect Security Profile using the console, you must first have an empty thing group. To create an empty thing group, see [Static thing groups \(p. 258\)](#). If you have created an empty thing group, then set the empty thing group as the target of the ML Detect Security Profile.

Note

You need to set the target of your Security Profile back to a device group with devices within 30 days, or you won't be able to reactivate the Security Profile.

Delete ML Detect Security Profile by using the console

To delete a Security Profile, follow these steps:

1. In the AWS IoT console navigate to the sidebar and choose the **Defend** section.
2. Under **Defend**, choose **Detect** and then **Security Profiles**.
3. Choose the ML Detect Security Profile you want to delete.
4. Choose **Actions**, and then from the options, choose **Delete**.

Note

After an ML Detect Security Profile is deleted, you won't be able to reactivate the Security Profile.

Pause an ML Detect Security Profile by using the CLI

To pause a ML Detect Security Profile by using the CLI, use the `detach-security-profile` command:

```
$aws iot detach-security-profile --security-profile-name SecurityProfileName --  
security-profile-target-arn arn:aws:iot:us-east-1:123456789012:all/registered-things
```

Note

This option is only available in AWS CLI. Similar to the console workflow, you need to set the target of your Security Profile back to a device group with devices within 30 days, or you won't be able to reactivate the Security Profile. To attach a Security Profile to a device group, use the `attach-security-profile` command.

Delete a ML Detect Security Profile by using the CLI

You can delete a Security Profile by using the `delete-security-profile` command below:

```
delete-security-profile --security-profile-name SecurityProfileName
```

Note

After an ML Detect Security Profile is deleted, you won't be able to reactivate the Security Profile.

Custom metrics

With AWS IoT Device Defender custom metrics, you can define and monitor metrics that are unique to your fleet or use case, such as number of devices connected to Wi-Fi gateways, charge levels for batteries, or number of power cycles for smart plugs. Custom metric behaviors are defined in Security Profiles, which specify expected behaviors for a group of devices (a thing group) or for all devices. You can monitor behaviors by setting up alarms, which you can use to detect and respond to issues that are specific to the devices.

This chapter contains the following sections:

- [How to use custom metrics in the console \(p. 892\)](#)
- [How to use custom metrics from the CLI \(p. 894\)](#)
- [Custom metrics CLI commands \(p. 897\)](#)
- [Custom metrics APIs \(p. 897\)](#)

How to use custom metrics in the console

Tutorials

- [AWS IoT Device Defender Agent SDK \(Python\) \(p. 893\)](#)
- [Create a custom metric and add it to a Security Profile \(p. 893\)](#)
- [View custom metric details \(p. 893\)](#)
- [Update a custom metric \(p. 894\)](#)
- [Delete a custom metric \(p. 894\)](#)

AWS IoT Device Defender Agent SDK (Python)

To get started, download the AWS IoT Device Defender Agent SDK (Python) sample agent. The agent gathers the metrics and publishes reports. Once your device-side metrics are publishing, you can view the metrics being collected and determine thresholds for setting up alarms. Instructions for setting up the device agent are available on the [AWS IoT Device Defender Agent SDK \(Python\) Readme](#). For more information, see [AWS IoT Device Defender Agent SDK \(Python\)](#).

Create a custom metric and add it to a Security Profile

The following procedure shows you how to create a custom metric in the console.

1. In the [AWS IoT console](#), in the navigation pane, expand **Defend**, and then choose **Detect, Metrics**.
2. On the **Custom metrics** page, choose **Create**.
3. On the **Create custom metric** page, do the following.
 1. Under **Name**, enter a name for your custom metric. You can't modify this name after you create the custom metric.
 2. Under **Display name (optional)**, you can enter a friendly name for your custom metric. It doesn't have to be unique and it can be modified after creation.
 3. Under **Type**, choose the type of metric you'd like to monitor. Metric types include **string-list**, **ip-address-list**, **number-list**, and **number**. The type can't be modified after creation.
 4. Under **Tags**, you can select tags to be associated with the resource.

When you're done, choose **Confirm**.

4. After you've created your custom metric, the **Custom metrics** page appears, where you can see your newly created custom metric.
5. Next, you need to add your custom metric to a Security Profile. In the [AWS IoT console](#), in the navigation pane, expand **Defend**, and then choose **Detect, Security profiles**.
6. Choose the Security Profile you'd like to add your custom metric to.
7. Choose **Actions, Edit**.
8. Choose **Additional Metrics to retain**, and then choose your custom metric. Choose **Next** on the following screens until you reach the **Confirm** page. Choose **Save** and **Continue**. After your custom metric has been successfully added, the Security Profile details page appears.

Note

Percentile statistics are not available for metrics when any of the metric values are negative numbers.

View custom metric details

The following procedure shows you how to view a custom metric's details in the console.

1. In the [AWS IoT console](#), in the navigation pane, expand **Defend**, and then choose **Detect, Metrics**.
2. Choose the **Metric name** of the custom metric you'd like to view the details of.

Update a custom metric

The following procedure shows you how to update a custom metric in the console.

1. In the [AWS IoT console](#), in the navigation pane, expand **Defend**, and then choose **Detect, Metrics**.
2. Choose the option button next to the custom metric you'd like to update. Then, for **Actions**, choose **Edit**.
3. On the **Update custom metric** page, you can edit the display name and remove or add tags.
4. After you're done, choose **Update**. The **Custom metrics** page.

Delete a custom metric

The following procedure shows you how to delete a custom metric in the console.

1. First, remove your custom metric from any Security Profile it's referenced in. You can view which Security Profiles contain your custom metric on your custom metric details page. In the [AWS IoT console](#), in the navigation pane, expand **Defend**, and then choose **Detect, Metrics**.
2. Choose the custom metric you'd like to remove. Remove the custom metric from any Security Profile listed under **Security Profiles** on the custom metric details page.
3. In the [AWS IoT console](#), in the navigation pane, expand **Defend**, and then choose **Detect, Metrics**.
4. Choose the option button next to the custom metric you'd like to delete. Then, for **Actions**, choose **Delete**.
5. On the **Are you sure you want to delete custom metric?** message, choose **Delete custom metric**.

Warning

After you've deleted a custom metric, you lose all data associated with the metric. This action can't be undone.

How to use custom metrics from the CLI

Tutorials

- [AWS IoT Device Defender Agent SDK \(Python\) \(p. 894\)](#)
- [Create a custom metric and add it to a Security Profile \(p. 894\)](#)
- [View custom metric details \(p. 895\)](#)
- [Update a custom metric \(p. 896\)](#)
- [Delete a custom metric \(p. 896\)](#)

AWS IoT Device Defender Agent SDK (Python)

To get started, download the AWS IoT Device Defender Agent SDK (Python) sample agent. The agent gathers the metrics and publishes reports. After your device-side metrics are publishing, you can view the metrics being collected and determine thresholds for setting up alarms. Instructions for setting up the device agent are available on the [AWS IoT Device Defender Agent SDK \(Python\) Readme](#). For more information, see [AWS IoT Device Defender Agent SDK \(Python\)](#).

Create a custom metric and add it to a Security Profile

The following procedure shows you how to create a custom metric and add it to a Security Profile from the CLI.

1. Use the `create-custom-metric` command to create your custom metric. The following example creates a custom metric that measures battery percentage.

```
aws iot create-custom-metric \
--metric-name "batteryPercentage" \
--metric-type "number" \
--display-name "Remaining battery percentage." \
--region us-east-1 \
--client-request-token "02ccb92b-33e8-4dfa-a0c1-35b181ed26b0" \
```

Output:

```
{
  "metricName": "batteryPercentage",
  "metricArn": "arn:aws:iot:us-east-1:1234564789012:custommetric/batteryPercentage"
}
```

- After you've created your custom metric, you can either add the custom metric to an existing profile using `update-security-profile` or create a new security profile to add the custom metric to using `create-security-profile`. Here, we create a new security profile called `batteryUsage` to add our new `batteryPercentage` custom metric to. We also add a Rules Detect metric called `cellularBandwidth`.

```
aws iot create-security-profile \
--security-profile-name batteryUsage \
--security-profile-description "Shows how much battery is left in percentile." \
--behaviors "[{\\"name\\":\\"great-than-75\\",\\"metric\\":\\"batteryPercentage\\", \
\\"criteria\\":{\\"comparisonOperator\\":\\"greater-than\\",\\"value\\":{\\"number \
\\":75},\\"consecutiveDatapointsToAlarm\\":5,\\"consecutiveDatapointsToClear \
\\":1},{\"name\\":\\"cellularBandwidth\\",\\"metric\\":\\"aws:message-byte-size\\", \
\\"criteria\\":{\\"comparisonOperator\\":\\"less-than\\",\\"value\\":{\\"count\\":128}, \
\\"consecutiveDatapointsToAlarm\\":1,\\"consecutiveDatapointsToClear\\":1}}}]" \
--region us-east-1
```

Output:

```
{
  "securityProfileArn": "arn:aws:iot:us-east-1:1234564789012:securityprofile/
batteryUsage",
  "securityProfileName": "batteryUsage"
}
```

Note

Percentile statistics are not available for metrics when any of the metric values are negative numbers.

View custom metric details

The following procedure shows you how to view the details for a custom metric from the CLI.

- Use the `list-custom-metrics` command to view all of your custom metrics.

```
aws iot list-custom-metrics \
--region us-east-1
```

The output of this command looks like the following.

```
{
  "metricNames": [
```

```

        "batteryPercentage"
    ]
}
```

Update a custom metric

The following procedure shows you how to update a custom metric from the CLI.

- Use the [update-custom-metric](#) command to update a custom metric. The following example updates the display-name.

```
aws iot update-custom-metric \
--metric-name batteryPercentage \
--display-name 'remaining battery percentage on device' \
--region us-east-1
```

The output of this command looks like the following.

```
{
    "metricName": "batteryPercentage",
    "metricArn": "arn:aws:iot:us-east-1:1234564789012:custommetric/batteryPercentage",
    "metricType": "number",
    "displayName": "remaining battery percentage on device",
    "creationDate": "2020-11-17T23:01:35.110000-08:00",
    "lastModifiedDate": "2020-11-17T23:02:12.879000-08:00"
}
```

Delete a custom metric

The following procedure shows you how to delete a custom metric from the CLI.

1. To delete a custom metric, first remove it from any Security Profiles that it's attached to. Use the [list-security-profiles](#) command to view Security Profiles with a certain custom metric.
2. To remove a custom metric from a Security Profile, use the [update-security-profiles](#) command. Enter all information that you want to keep, but exclude the custom metric.

```
aws iot update-security-profile \
--security-profile-name batteryUsage \
--behaviors "[{\\"name\\":\\"cellularBandwidth\",\\"metric\\":\\"aws:message-byte-size\\",\\"criteria\\":{\\\"comparisonOperator\\\":\\\"less-than\\\",\\\"value\\\":{\\\"count\\\":128},\\\"consecutiveDatapointsToAlarm\\\":1,\\\"consecutiveDatapointsToClear\\\":1}}}]
```

The output of this command looks like the following.

```
{
    "behaviors": [{}],
    "securityProfileName": "batteryUsage",
    "lastModifiedDate": 2020-11-17T23:02:12.879000-09:00,
    "securityProfileDescription": "Shows how much battery is left in percentile.",
    "version": 2,
    "securityProfileArn": "arn:aws:iot:us-east-1:1234564789012:securityprofile/batteryUsage",
    "creationDate": 2020-11-17T23:02:12.879000-09:00
}
```

3. After the custom metric is detached, use the [delete-custom-metric](#) command to delete the custom metric.

```
aws iot delete-custom-metric \
--metric-name batteryPercentage \
--region us-east-1
```

The output of this command looks like the following

```
HTTP 200
```

Custom metrics CLI commands

You can use the following CLI commands to create and manage custom metrics.

- [create-custom-metric](#)
- [describe-custom-metric](#)
- [list-custom-metrics](#)
- [update-custom-metric](#)
- [delete-custom-metric](#)
- [list-security-profiles](#)

Custom metrics APIs

The following APIs can be used to create and manage custom metrics.

- [CreateCustomMetric](#)
- [DescribeCustomMetric](#)
- [ListCustomMetrics](#)
- [UpdateCustomMetric](#)
- [DeleteCustomMetric](#)
- [ListSecurityProfiles](#)

Device-side metrics

When creating a Security Profile, you can specify your IoT device's expected behavior by configuring behaviors and thresholds for metrics generated by IoT devices. The following are device-side metrics, which are metrics from agents that you install on your devices.

Bytes out (aws:all-bytes-out)

The number of outbound bytes from a device during a given time period.

Use this metric to specify the maximum or minimum amount of outbound traffic that a device should send, measured in bytes, in a given period of time.

Compatible with: Rules Detect | ML Detect

Operators: less-than | less-than-equals | greater-than | greater-than-equals

Value: a non-negative integer

Units: bytes

Duration: a non-negative integer. Valid values are 300, 600, 900, 1800, or 3600 seconds.

Example

```
{
  "name": "TCP outbound traffic",
  "metric": "aws:all-bytes-out",
  "criteria": {
    "comparisonOperator": "less-than-equals",
    "value": {
      "count": 4096
    },
    "durationSeconds": 300,
    "consecutiveDatapointsToAlarm": 1,
    "consecutiveDatapointsToClear": 1
  },
  "suppressAlerts": true
}
```

Example Example using a statisticalThreshold

```
{
  "name": "TCP outbound traffic",
  "metric": "aws:all-bytes-out",
  "criteria": {
    "comparisonOperator": "less-than-equals",
    "statisticalThreshold": {
      "statistic": "p50"
    },
    "durationSeconds": 900,
    "consecutiveDatapointsToAlarm": 1,
    "consecutiveDatapointsToClear": 1
  },
  "suppressAlerts": true
}
```

Example Example using ML Detect

```
{
  "name": "Outbound traffic ML behavior",
  "metric": "aws:all-bytes-out",
  "criteria": {
    "consecutiveDatapointsToAlarm": 1,
    "consecutiveDatapointsToClear": 1,
    "mlDetectionConfig": {
      "confidenceLevel": "HIGH"
    }
  },
  "suppressAlerts": true
}
```

Bytes in (aws:all-bytes-in)

The number of inbound bytes to a device during a given time period.

Use this metric to specify the maximum or minimum amount of inbound traffic that a device should receive, measured in bytes, in a given period of time.

Compatible with: Rules Detect | ML Detect

Operators: less-than | less-than-equals | greater-than | greater-than-equals

Value: a non-negative integer

Units: bytes

Duration: a non-negative integer. Valid values are 300, 600, 900, 1800, or 3600 seconds.

Example

```
{  
    "name": "TCP inbound traffic",  
    "metric": "aws:all-bytes-in",  
    "criteria": {  
        "comparisonOperator": "less-than-equals",  
        "value": {  
            "count": 4096  
        },  
        "durationSeconds": 300,  
        "consecutiveDatapointsToAlarm": 1,  
        "consecutiveDatapointsToClear": 1  
    },  
    "suppressAlerts": true  
}
```

Example Example using a statisticalThreshold

```
{  
    "name": "TCP inbound traffic",  
    "metric": "aws:all-bytes-in",  
    "criteria": {  
        "comparisonOperator": "less-than-equals",  
        "statisticalThreshold": {  
            "statistic": "p90"  
        },  
        "durationSeconds": 300,  
        "consecutiveDatapointsToAlarm": 1,  
        "consecutiveDatapointsToClear": 1  
    },  
    "suppressAlerts": true  
}
```

Example Example using ML Detect

```
{  
    "name": "Inbound traffic ML behavior",  
    "metric": "aws:all-bytes-in",  
    "criteria": {  
        "consecutiveDatapointsToAlarm": 1,  
        "consecutiveDatapointsToClear": 1,  
        "mlDetectionConfig": {  
            "confidenceLevel": "HIGH"  
        }  
    },  
    "suppressAlerts": true  
}
```

Listening TCP port count (aws:num-listening-tcp-ports)

The number of TCP ports the device is listening on.

Use this metric to specify the maximum number of TCP ports that each device should monitor.

Compatible with: Rules Detect | ML Detect

Unit: failures

Operators: less-than | less-than-equals | greater-than | greater-than-equals

Value: a non-negative integer

Units: failures

Duration: a non-negative integer. Valid values are 300, 600, 900, 1800, or 3600 seconds.

Example

```
{  
  "name": "Max TCP Ports",  
  "metric": "aws:num-listening-tcp-ports",  
  "criteria": {  
    "comparisonOperator": "less-than-equals",  
    "value": {  
      "count": 5  
    },  
    "durationSeconds": 300,  
    "consecutiveDatapointsToAlarm": 1,  
    "consecutiveDatapointsToClear": 1  
  },  
  "suppressAlerts": true  
}
```

Example Example using a statisticalThreshold

```
{  
  "name": "Max TCP Ports",  
  "metric": "aws:num-listening-tcp-ports",  
  "criteria": {  
    "comparisonOperator": "less-than-equals",  
    "statisticalThreshold": {  
      "statistic": "p50"  
    },  
    "durationSeconds": 300,  
    "consecutiveDatapointsToAlarm": 1,  
    "consecutiveDatapointsToClear": 1  
  },  
  "suppressAlerts": true  
}
```

Example Example using ML detect

```
{  
  "name": "Max TCP Port ML behavior",  
  "metric": "aws:num-listening-tcp-ports",  
  "criteria": {  
    "consecutiveDatapointsToAlarm": 1,  
    "consecutiveDatapointsToClear": 1,  
    "mlDetectionConfig": {  
      "confidenceLevel": "HIGH"  
    }  
  },  
  "suppressAlerts": true  
}
```

}

Listening UDP port count (aws: num-listening-udp-ports)

The number of UDP ports the device is listening on.

Use this metric to specify the maximum number of UDP ports that each device should monitor.

Compatible with: Rules Detect | ML Detect

Unit: failures

Operators: less-than | less-than-equals | greater-than | greater-than-equals

Value: a non-negative integer

Units: failures

Duration: a non-negative integer. Valid values are 300, 600, 900, 1800, or 3600 seconds.

Example

```
{  
  "name": "Max UDP Ports",  
  "metric": "aws:num-listening-udp-ports",  
  "criteria": {  
    "comparisonOperator": "less-than-equals",  
    "value": {  
      "count": 5  
    },  
    "durationSeconds": 300,  
    "consecutiveDatapointsToAlarm": 1,  
    "consecutiveDatapointsToClear": 1  
  },  
  "suppressAlerts": true  
}
```

Example Example using a statisticalThreshold

```
{  
  "name": "Max UDP Ports",  
  "metric": "aws:num-listening-udp-ports",  
  "criteria": {  
    "comparisonOperator": "less-than-equals",  
    "statisticalThreshold": {  
      "statistic": "p50"  
    },  
    "durationSeconds": 300,  
    "consecutiveDatapointsToAlarm": 1,  
    "consecutiveDatapointsToClear": 1  
  },  
  "suppressAlerts": true  
}
```

Example Example using ML Detect

```
{  
  "name": "Max UDP Port ML behavior",  
  "metric": "aws:num-listening-tcp-ports",
```

```

    "criteria": {
      "consecutiveDatapointsToAlarm": 1,
      "consecutiveDatapointsToClear": 1,
      "mlDetectionConfig": {
        "confidenceLevel": "HIGH"
      }
    },
    "suppressAlerts": true
}

```

Packets out (aws:all-packets-out)

The number of outbound packets from a device during a given time period.

Use this metric to specify the maximum or minimum amount of total outbound traffic that a device should send in a given period of time.

Compatible with: Rules Detect | ML Detect

Operators: less-than | less-than-equals | greater-than | greater-than-equals

Value: a non-negative integer

Units: packets

Duration: a non-negative integer. Valid values are 300, 600, 900, 1800, or 3600 seconds.

Example

```
{
  "name": "TCP outbound traffic",
  "metric": "aws:all-packets-out",
  "criteria": {
    "comparisonOperator": "less-than-equals",
    "value": {
      "count": 100
    },
    "durationSeconds": 300,
    "consecutiveDatapointsToAlarm": 1,
    "consecutiveDatapointsToClear": 1
  },
  "suppressAlerts": true
}
```

Example Example using a statisticalThreshold

```
{
  "name": "TCP outbound traffic",
  "metric": "aws:all-packets-out",
  "criteria": {
    "comparisonOperator": "less-than-equals",
    "statisticalThreshold": {
      "statistic": "p90"
    },
    "durationSeconds": 300,
    "consecutiveDatapointsToAlarm": 1,
    "consecutiveDatapointsToClear": 1
  },
  "suppressAlerts": true
}
```

Example Example using ML Detect

```
{
  "name": "Outbound sent ML behavior",
  "metric": "aws:all-packets-out",
  "criteria": {
    "consecutiveDatapointsToAlarm": 1,
    "consecutiveDatapointsToClear": 1,
    "mlDetectionConfig": {
      "confidenceLevel": "HIGH"
    }
  },
  "suppressAlerts": true
}
```

Packets in (aws:all-packets-in)

The number of inbound packets to a device during a given time period.

Use this metric to specify the maximum or minimum amount of total inbound traffic that a device should receive in a given period of time.

Compatible with: Rule Detect | ML Detect

Operators: less-than | less-than-equals | greater-than | greater-than-equals

Value: a non-negative integer

Units: packets

Duration: a non-negative integer. Valid values are 300, 600, 900, 1800 or 3600 seconds.

Example

```
{
  "name": "TCP inbound traffic",
  "metric": "aws:all-packets-in",
  "criteria": {
    "comparisonOperator": "less-than-equals",
    "value": {
      "count": 100
    },
    "durationSeconds": 300,
    "consecutiveDatapointsToAlarm": 1,
    "consecutiveDatapointsToClear": 1
  },
  "suppressAlerts": true
}
```

Example

Example using a statisticalThreshold

```
{
  "name": "TCP inbound traffic",
  "metric": "aws:all-packets-in",
  "criteria": {
    "comparisonOperator": "less-than-equals",
    "statisticalThreshold": {
      "statistic": "p90"
    }
  }
}
```

```

    "durationSeconds": 300,
    "consecutiveDatapointsToAlarm": 1,
    "consecutiveDatapointsToClear": 1
},
"suppressAlerts": true
}

```

Example Example using ML Detect

```
{
  "name": "Inbound sent ML behavior",
  "metric": "aws:all-packets-in",
  "criteria": {
    "consecutiveDatapointsToAlarm": 1,
    "consecutiveDatapointsToClear": 1,
    "mlDetectionConfig": {
      "confidenceLevel": "HIGH"
    }
  },
  "suppressAlerts": true
}
```

Destination IPs (aws:destination-ip-addresses)

A set of IP destinations.

Use this metric to specify a set of allowed (formerly referred to as whitelisted) or denied (formerly referred to as blacklisted) Classless Inter-Domain Routings (CIDR) from which each device must or must not connect to AWS IoT.

Compatible with: Rules Detect

Operators: in-cidr-set | not-in-cidr-set

Values: a list of CIDRs

Units: n/a

Example

```
{
  "name": "Denied source IPs",
  "metric": "aws:source-ip-address",
  "criteria": {
    "comparisonOperator": "not-in-cidr-set",
    "value": {
      "cidrs": [ "12.8.0.0/16", "15.102.16.0/24" ]
    }
  },
  "suppressAlerts": true
}
```

Listening TCP ports (aws:listening-tcp-ports)

The TCP ports that the device is listening on.

Use this metric to specify a set of allowed (formerly referred to as whitelisted) or denied (formerly referred to as blacklisted) TCP ports on which each device must or must not listen.

Compatible with: Rules Detect

Operators: in-port-set | not-in-port-set

Values: a list of ports

Units: n/a

Example

```
{  
  "name": "Listening TCP Ports",  
  "metric": "aws:listening-tcp-ports",  
  "criteria": {  
    "comparisonOperator": "in-port-set",  
    "value": {  
      "ports": [ 443, 80 ]  
    }  
  },  
  "suppressAlerts": true  
}
```

Listening UDP ports (aws:listening-udp-ports)

The UDP ports that the device is listening on.

Use this metric to specify a set of allowed (formerly referred to as whitelisted) or denied (formerly referred to as blacklisted) UDP ports on which each device must or must not listen.

Compatible with: Rules Detect

Operators: in-port-set | not-in-port-set

Values: a list of ports

Units: n/a

Example

```
{  
  "name": "Listening UDP Ports",  
  "metric": "aws:listening-udp-ports",  
  "criteria": {  
    "comparisonOperator": "in-port-set",  
    "value": {  
      "ports": [ 1025, 2000 ]  
    }  
  }  
}
```

Established TCP connections count (aws:num-established-tcp-connections)

The number of TCP connections for a device.

Use this metric to specify the maximum or minimum number of active TCP connections that each device should have (All TCP states).

Compatible with: Rules Detect | ML Detect

Operators: less-than | less-than-equals | greater-than | greater-than-equals

Value: a non-negative integer

Units: connections

Example

```
{
  "name": "TCP Connection Count",
  "metric": "aws:num-established-tcp-connections",
  "criteria": {
    "comparisonOperator": "less-than-equals",
    "value": {
      "count": 3
    },
    "consecutiveDatapointsToAlarm": 1,
    "consecutiveDatapointsToClear": 1
  },
  "suppressAlerts": true
}
```

Example Example using a statisticalThreshold

```
{
  "name": "TCP Connection Count",
  "metric": "aws:num-established-tcp-connections",
  "criteria": {
    "comparisonOperator": "less-than-equals",
    "statisticalThreshold": {
      "statistic": "p90"
    },
    "durationSeconds": 900,
    "consecutiveDatapointsToAlarm": 1,
    "consecutiveDatapointsToClear": 1
  },
  "suppressAlerts": true
}
```

Example Example using ML Detect

```
{
  "name": "Connection count ML behavior",
  "metric": "aws:num-established-tcp-connections",
  "criteria": {
    "consecutiveDatapointsToAlarm": 1,
    "consecutiveDatapointsToClear": 1,
    "mlDetectionConfig": {
      "confidenceLevel": "HIGH"
    }
  },
  "suppressAlerts": true
}
```

Device metrics document specification

Overall structure

Long name	Short name	Required	Type	Constraints	Notes
header	hed	Y	Object		Complete block required

Long name	Short name	Required	Type	Constraints	Notes
					for well-formed report.
metrics	met	Y	Object		A report can have both or at least one metrics or custom_metrics block.
custom_metrics	cmet	Y	Object		A report can have both or at least one metrics or custom_metrics block.

Header block

Long name	Short name	Required	Type	Constraints	Notes
report_id	rid	Y	Integer		Monotonically increasing value. Epoch timestamp recommended.
version	v	Y	String	Major.Minor	Minor increments with addition of field. Major increments if metrics removed.

Metrics block:

TCP connections

Long name	Short name	Parent element	Required	Type	Constraints	Notes
tcp_connections	tc	metrics	N	Object		
established_connections	est	tcp_connections	N	Object		Established TCP state
connections	cs	established_connections	N	List<Object>		
remote_addr	rad	connections	Y	Number	ip:port	IP can be IPv6 or IPv4
local_port	lp	connections	N	Number	>= 0	
local_interface	li	connections	N	String		Interface name

Long name	Short name	Parent element	Required	Type	Constraints	Notes
total	t	established_connections		Number	>= 0	Number of established connections

Listening TCP ports

Long name	Short name	Parent element	Required	Type	Constraints	Notes
listening_tcp_ports		metrics	N	Object		
ports	pts	listening_tcp_ports		List<Object>	> 0	
port	pt	ports	N	Number	> 0	ports should be numbers greater than 0
interface	if	ports	N	String		Interface name
total	t	listening_tcp_ports		Number	>= 0	

Listening UDP ports

Long name	Short name	Parent element	Required	Type	Constraints	Notes
listening_udp_ports		metrics	N	Object		
ports	pts	listening_udp_ports		List<Port>	> 0	
port	pt	ports	N	Number	> 0	Ports should be numbers greater than 0
interface	if	ports	N	String		Interface name
total	t	listening_udp_ports		Number	>= 0	

Network statistics

Long name	Short name	Parent element	Required	Type	Constraints	Notes
network_stats	ns	metrics	N	Object		
bytes_in	bi	network_stats	N	Number	Delta Metric, >= 0	
bytes_out	bo	network_stats	N	Number	Delta Metric, >= 0	

Long name	Short name	Parent element	Required	Type	Constraints	Notes
packets_in	pi	network_stats	N	Number	Delta Metric, >= 0	
packets_out	po	network_stats	N	Number	Delta Metric, >= 0	

Example

The following JSON structure uses long names.

```
{
  "header": {
    "report_id": 1530304554,
    "version": "1.0"
  },
  "metrics": {
    "listening_tcp_ports": {
      "ports": [
        {
          "interface": "eth0",
          "port": 24800
        },
        {
          "interface": "eth0",
          "port": 22
        },
        {
          "interface": "eth0",
          "port": 53
        }
      ],
      "total": 3
    },
    "listening_udp_ports": {
      "ports": [
        {
          "interface": "eth0",
          "port": 5353
        },
        {
          "interface": "eth0",
          "port": 67
        }
      ],
      "total": 2
    },
    "network_stats": {
      "bytes_in": 29358693495,
      "bytes_out": 26485035,
      "packets_in": 10013573555,
      "packets_out": 11382615
    },
    "tcp_connections": {
      "established_connections": {
        "connections": [
          {
            "local_interface": "eth0",
            "local_port": 80,
            "remote_addr": "192.168.0.1:8000"
          },
          {
            "local_interface": "eth0",
            "local_port": 443,
            "remote_addr": "192.168.0.1:443"
          }
        ]
      }
    }
  }
}
```

```
{
    "local_interface": "eth0",
    "local_port": 80,
    "remote_addr": "192.168.0.1:8000"
}
],
"total": 2
}
},
"custom_metrics": {
    "MyMetricOfType_Number": [
        {
            "number": 1
        }
    ],
    "MyMetricOfType_NumberList": [
        {
            "number_list": [
                1,
                2,
                3
            ]
        }
    ],
    "MyMetricOfType_StringList": [
        {
            "string_list": [
                "value_1",
                "value_2"
            ]
        }
    ],
    "MyMetricOfType_IpList": [
        {
            "ip_list": [
                "172.0.0.0",
                "172.0.0.10"
            ]
        }
    ]
}
}
```

Example Example JSON structure using short names

```
{
    "hed": {
        "rid": 1530305228,
        "v": "1.0"
    },
    "met": {
        "tp": {
            "pts": [
                {
                    "if": "eth0",
                    "pt": 24800
                },
                {
                    "if": "eth0",
                    "pt": 22
                },
                {
                    "if": "eth0",
                    "pt": 2
                }
            ]
        }
    }
}
```

```

        "pt": 53
    }
],
"t": 3
},
"up": {
    "pts": [
        {
            "if": "eth0",
            "pt": 5353
        },
        {
            "if": "eth0",
            "pt": 67
        }
    ],
    "t": 2
},
"ns": {
    "bi": 29359307173,
    "bo": 26490711,
    "pi": 10014614051,
    "po": 11387620
},
"tc": {
    "ec": {
        "cs": [
            {
                "li": "eth0",
                "lp": 80,
                "rad": "192.168.0.1:8000"
            },
            {
                "li": "eth0",
                "lp": 80,
                "rad": "192.168.0.1:8000"
            }
        ],
        "t": 2
    }
},
"cmet": {
    "MyMetricOfType_Number": [
        {
            "number": 1
        }
    ],
    "MyMetricOfType_NumberList": [
        {
            "number_list": [
                1,
                2,
                3
            ]
        }
    ],
    "MyMetricOfType_StringList": [
        {
            "string_list": [
                "value_1",
                "value_2"
            ]
        }
    ],
    "MyMetricOfType_IpList": [

```

```
{  
    "ip_list": [  
        "172.0.0.0",  
        "172.0.0.10"  
    ]  
}  
}  
}
```

Sending metrics from devices

AWS IoT Device Defender Detect can collect, aggregate, and monitor metrics data generated by AWS IoT devices to identify devices that exhibit abnormal behavior. This section shows you how to send metrics from a device to AWS IoT Device Defender.

You must securely deploy the AWS IoT SDK version two on your AWS IoT connected devices or device gateways to collect device-side metrics. See the full list of SDKs [here](#).

You should use AWS IoT Device Client to publish metrics because it provides a single agent that covers the features present in both AWS IoT Device Defender and AWS IoT Device Management. These features include jobs, secure tunneling, AWS IoT Device Defender metrics publishing, and more.

Using the AWS IoT Device Client to publish metrics

To install AWS IoT Device Client, you can download it from [Github](#). After you've installed the AWS IoT Device Client on the device for which you want to collect device-side data, you must configure it to send device-side metrics to AWS IoT Device Defender. Verify that the AWS IoT Device Client [configuration file](#) has the following parameters set in the device-defender section:

```
"device-defender": {  
    "enabled": true,  
    "interval-in-seconds": 300  
}
```

Warning

You should set the time interval to a minimum of 300 seconds. If you set the time interval to anything less than 300 seconds, your metric data may be throttled.

After you've updated your configuration, you can create security profiles and behaviors in the AWS IoT Device Defender console to monitor the metrics that your devices publish to the cloud. You can find published metrics in the AWS IoT Core console by choosing Defend, Detect, and then Metrics.

Cloud-side metrics

When creating a Security Profile, you can specify your IoT device's expected behavior by configuring behaviors and thresholds for metrics generated by IoT devices. The following are cloud-side metrics, which are metrics from AWS IoT.

Message size (aws:message-byte-size)

The number of bytes in a message. Use this metric to specify the maximum or minimum size (in bytes) of each message transmitted from a device to AWS IoT.

Compatible with: Rules Detect | ML Detect

Operators: less-than | less-than-equals | greater-than | greater-than-equals

Value: a non-negative integer

Units: bytes

Example

```
{
  "name": "Max Message Size",
  "metric": "aws:message-byte-size",
  "criteria": {
    "comparisonOperator": "less-than-equals",
    "value": {
      "count": 1024
    },
    "consecutiveDatapointsToAlarm": 1,
    "consecutiveDatapointsToClear": 1
  },
  "suppressAlerts": true
}
```

Example Example using a statisticalThreshold

```
{
  "name": "Large Message Size",
  "metric": "aws:message-byte-size",
  "criteria": {
    "comparisonOperator": "less-than-equals",
    "statisticalThreshold": {
      "statistic": "p90"
    },
    "durationSeconds": 300,
    "consecutiveDatapointsToAlarm": 1,
    "consecutiveDatapointsToClear": 1
  },
  "suppressAlerts": true
}
```

Example Example using ML Detect

```
{
  "name": "Message size ML behavior",
  "metric": "aws:message-byte-size",
  "criteria": {
    "consecutiveDatapointsToAlarm": 1,
    "consecutiveDatapointsToClear": 1,
    "mlDetectionConfig": {
      "confidenceLevel": "HIGH"
    }
  },
  "suppressAlerts": true
}
```

An alarm occurs for a device if during three consecutive five-minute periods, it transmits messages where the cumulative size is more than that measured for 90 percent of all other devices reporting for this Security Profile behavior.

Messages sent (aws:num-messages-sent)

The number of messages sent by a device during a given time period.

Use this metric to specify the maximum or minimum number of messages that can be sent between AWS IoT and each device in a given period of time.

Compatible with: Rules Detect | ML Detect

Operators: less-than | less-than-equals | greater-than | greater-than-equals

Value: a non-negative integer

Units: messages

Duration: a non-negative integer. Valid values are 300, 600, 900, 1800, or 3600 seconds.

Example

```
{  
  
  "name": "Out bound message count",  
  "metric": "aws:num-messages-sent",  
  "criteria": {  
    "comparisonOperator": "less-than-equals",  
    "value": {  
      "count": 50  
    },  
    "durationSeconds": 300,  
    "consecutiveDatapointsToAlarm": 1,  
    "consecutiveDatapointsToClear": 1  
  },  
  "suppressAlerts": true  
}
```

Example Example using a statisticalThreshold

```
{  
  
  "name": "Out bound message rate",  
  "metric": "aws:num-messages-sent",  
  "criteria": {  
    "comparisonOperator": "less-than-equals",  
    "statisticalThreshold": {  
      "statistic": "p99"  
    },  
    "durationSeconds": 300,  
    "consecutiveDatapointsToAlarm": 1,  
    "consecutiveDatapointsToClear": 1  
  },  
  "suppressAlerts": true  
}
```

Example Example using ML Detect

```
{  
  
  "name": "Messages sent ML behavior",  
  "metric": "aws:num-messages-sent",  
  "criteria": {  
    "consecutiveDatapointsToAlarm": 1,  
    "consecutiveDatapointsToClear": 1,  
    "mlDetectionConfig": {  
      "confidenceLevel": "HIGH"  
    },  
  },  
  "suppressAlerts": true  
}
```

}

Messages received (aws:num-messages-received)

The number of messages received by a device during a given time period.

Use this metric to specify the maximum or minimum number of messages that can be received between AWS IoT and each device in a given period of time.

Compatible with: Rules Detect | ML Detect

Operators: less-than | less-than-equals | greater-than | greater-than-equals

Value: a non-negative integer

Units: messages

Duration: a non-negative integer. Valid values are 300, 600, 900, 1800, or 3600 seconds.

Example

```
{
  "name": "In bound message count",
  "metric": "aws:num-messages-received",
  "criteria": {
    "comparisonOperator": "less-than-equals",
    "value": {
      "count": 50
    },
    "durationSeconds": 300,
    "consecutiveDatapointsToAlarm": 1,
    "consecutiveDatapointsToClear": 1
  },
  "suppressAlerts": true
}
```

Example Example using a statisticalThreshold

```
{
  "name": "In bound message rate",
  "metric": "aws:num-messages-received",
  "criteria": {
    "comparisonOperator": "less-than-equals",
    "statisticalThreshold": {
      "statistic": "p99"
    },
    "durationSeconds": 300,
    "consecutiveDatapointsToAlarm": 1,
    "consecutiveDatapointsToClear": 1
  },
  "suppressAlerts": true
}
```

Example Example using ML Detect

```
{
  "name": "Messages received ML behavior",
  "metric": "aws:num-messages-received",
  "criteria": {
    "consecutiveDatapointsToAlarm": 1,
    "consecutiveDatapointsToClear": 1,
```

```

        "mlDetectionConfig": {
            "confidenceLevel": "HIGH"
        },
        "suppressAlerts": true
    }
}

```

Authorization failures (aws:num-authorization-failures)

Use this metric to specify the maximum number of authorization failures allowed for each device in a given period of time. An authorization failure occurs when a request from a device to AWS IoT is denied (for example, if a device attempts to publish to a topic for which it does not have sufficient permissions).

Compatible with: Rules Detect | ML Detect

Unit: failures

Operators: less-than | less-than-equals | greater-than | greater-than-equals

Value: a non-negative integer

Duration: a non-negative integer. Valid values are 300, 600, 900, 1800, or 3600 seconds.

Example

```
{
    "name": "Authorization Failures",
    "metric": "aws:num-authorization-failures",
    "criteria": {
        "comparisonOperator": "less-than",
        "value": {
            "count": 5
        },
        "durationSeconds": 300,
        "consecutiveDatapointsToAlarm": 1,
        "consecutiveDatapointsToClear": 1
    },
    "suppressAlerts": true
}
```

Example Example using a statisticalThreshold

```
{
    "name": "Authorization Failures",
    "metric": "aws:num-authorization-failures",
    "criteria": {
        "comparisonOperator": "less-than-equals",
        "statisticalThreshold": {
            "statistic": "p50"
        },
        "durationSeconds": 300,
        "consecutiveDatapointsToAlarm": 1,
        "consecutiveDatapointsToClear": 1
    },
    "suppressAlerts": true
}
```

Example Example using ML Detect

```
{
    "name": "Authorization failures ML behavior",
```

```

    "metric": "aws:num-authorization-failures",
    "criteria": {
        "consecutiveDatapointsToAlarm": 1,
        "consecutiveDatapointsToClear": 1,
        "mlDetectionConfig": {
            "confidenceLevel": "HIGH"
        }
    },
    "suppressAlerts": true
}

```

Source IP (aws:source-ip-address)

The IP address from which a device has connected to AWS IoT.

Use this metric to specify a set of allowed (formerly referred to as whitelisted) or denied (formerly referred to as blacklisted) Classless Inter-Domain Routings (CIDR) from which each device must or must not connect to AWS IoT.

Compatible with: Rules Detect

Operators: in-cidr-set | not-in-cidr-set

Values: a list of CIDRs

Units: n/a

Example

```
{
    "name": "Denied source IPs",
    "metric": "aws:source-ip-address",
    "criteria": {
        "comparisonOperator": "not-in-cidr-set",
        "value": {
            "cidrs": [ "12.8.0.0/16", "15.102.16.0/24" ]
        }
    },
    "suppressAlerts": true
}
```

Connection attempts (aws:num-connection-attempts)

The number of times a device attempts to make a connection in a given time period.

Use this metric to specify the maximum or minimum number of connection attempts for each device. Successful and unsuccessful attempts are counted.

Compatible with: Rules Detect | ML Detect

Operators: less-than | less-than-equals | greater-than | greater-than-equals

Value: a non-negative integer

Units: connection attempts

Duration: a non-negative integer. Valid values are 300, 600, 900, 1800, or 3600 seconds.

Example

```
{
```

```

    "name": "Connection Attempts",
    "metric": "aws:num-connection-attempts",
    "criteria": {
        "comparisonOperator": "less-than-equals",
        "value": {
            "count": 5
        },
        "durationSeconds": 600,
        "consecutiveDatapointsToAlarm": 1,
        "consecutiveDatapointsToClear": 1
    },
    "suppressAlerts": true
}

```

Example Example using a statisticalThreshold

```

{
    "name": "Connection Attempts",
    "metric": "aws:num-connection-attempts",
    "criteria": {
        "comparisonOperator": "less-than-equals",
        "statisticalThreshold": {
            "statistic": "p10"
        },
        "durationSeconds": 300,
        "consecutiveDatapointsToAlarm": 1,
        "consecutiveDatapointsToClear": 1
    },
    "suppressAlerts": true
}

```

Example Example using ML Detect

```

{
    "name": "Connection attempts ML behavior",
    "metric": "aws:num-connection-attempts",
    "criteria": {
        "consecutiveDatapointsToAlarm": 1,
        "consecutiveDatapointsToClear": 1,
        "mlDetectionConfig": {
            "confidenceLevel": "HIGH"
        }
    },
    "suppressAlerts": false
}

```

Disconnects (aws:num-disconnects)

The number of times a device disconnects from AWS IoT during a given time period.

Use this metric to specify the maximum or minimum number of times a device disconnected from AWS IoT during a given time period.

Compatible with: Rules Detect | ML Detect

Operators: less-than | less-than-equals | greater-than | greater-than-equals

Value: a non-negative integer

Units: disconnects

Duration: a non-negative integer. Valid values are 300, 600, 900, 1800, or 3600 seconds.

Example

```
{
  "name": "Disconnections",
  "metric": "aws:num-disconnects",
  "criteria": {
    "comparisonOperator": "less-than-equals",
    "value": {
      "count": 5
    },
    "durationSeconds": 600,
    "consecutiveDatapointsToAlarm": 1,
    "consecutiveDatapointsToClear": 1
  },
  "suppressAlerts": true
}
```

Example Example using a statisticalThreshold

```
{
  "name": "Disconnections",
  "metric": "aws:num-disconnects",
  "criteria": {
    "comparisonOperator": "less-than-equals",
    "statisticalThreshold": {
      "statistic": "p10"
    },
    "durationSeconds": 300,
    "consecutiveDatapointsToAlarm": 1,
    "consecutiveDatapointsToClear": 1
  },
  "suppressAlerts": true
}
```

Example Example using ML Detect

```
{
  "name": "Disconnects ML behavior",
  "metric": "aws:num-disconnects",
  "criteria": {
    "consecutiveDatapointsToAlarm": 1,
    "consecutiveDatapointsToClear": 1,
    "mlDetectionConfig": {
      "confidenceLevel": "HIGH"
    }
  },
  "suppressAlerts": true
}
```

Scoping metrics in security profiles using dimensions

Dimensions are attributes that you can define to get more precise data about metrics and behaviors in your security profile. You define the scope by providing a value or pattern that is used as a filter. For example, you can define a topic filter dimension that applies a metric only to MQTT topics that match a particular value, such as "data/bulb/+/activity". For information about defining a dimension that you can use in your security profile, see [CreateDimension](#).

Dimension values support MQTT wildcards. MQTT wildcards help you subscribe to multiple topics simultaneously. There are two different kinds of wildcards: single-level (+) and multi-level (#). For

example, the dimension value `Data/bulb/+/activity` creates a subscription that matches all topics that exist on the same level as the `+`. Dimension values also support the MQTT client ID substitution variable `#{iot:ClientId}`.

Dimensions of type `TOPIC_FILTER` are compatible with the following set of cloud-side metrics:

- Number of messages sent
- Number of messages received
- Message byte size
- Source IP address
- Number of authorization failures

How to use dimensions in the console

To create and apply a dimension to a security profile behavior

1. In the [AWS IoT console](#), in the navigation pane, expand **Defend**, expand **Detect**, and then choose **Security profiles**.
2. On the **Security profiles** page, choose **Create** to add a new security profile, or **Edit** to apply a dimension to an existing security profile.
3. On the **Expected behaviors** page, select one of the five cloud-side metrics dimensions supports under **Metric**. The **Dimension** and **Dimension operator** boxes display. For information about supported cloud-side metrics, see [Scoping metrics in security profiles using dimensions \(p. 919\)](#).
4. For **Dimension**, choose **Add dimension**.
5. On the **Create a new dimension** page, enter details for your new dimension. **Dimensions values** supports MQTT wildcards `#` and `+` and the MQTT client ID substitution variable `#{iot:ClientId}`.

The screenshot shows the 'Create a new dimension' dialog box. At the top, a teal header bar contains the title 'Create a new dimension'. Below the header, a descriptive text states: 'Dimensions control the scope of behaviors that you define in your security profiles. For example, define a dimension that monitors specific MQTT topics.' The form fields are as follows:

- Dimension name**: A text input field containing 'Room_Temperature'.
- Dimension type**: A dropdown menu currently set to 'Topic filter'.
- Dimension values**: A text input field containing '/temperature/room/+'. Below this field is a button labeled 'Add value'.

At the bottom of the dialog, there is a section titled 'Tags' with a small arrow icon. At the very bottom, there are two buttons: 'Cancel' on the left and 'Save' on the right, both in white text on a dark blue background.

6. Choose **Save**.

7. You can optionally add dimensions to metrics under **Additional Metrics to retain**.
8. To finish creating the behavior, type the information in the other required fields, and then choose **Next**.
9. Complete the remaining steps to finish creating a security profile.

To view your violations

1. In the [AWS IoT console](#), in the navigation pane, expand **Defend**, expand **Detect**, and then choose **Violations**.

Event time	Thing name	Security profile	Behavior	Last emitted
Mar 26, 2020 10:55:00 PM -0700	iotconsole-1585288160280-0	test_SP	TamperDetected	4 message(s)
Mar 26, 2020 10:55:00 PM -0700	iotconsole-1585288160280-0		Messages sent less than 1 in 5 minutes with dimension Tamper, with datapoints to Alarm: 1, and datapoints to Clear: 2	

2. In the **Behavior** column, pause over the behavior you want to see the violation information for.

To view and update your dimensions

1. In the [AWS IoT console](#), in the navigation pane, expand **Defend**, expand **Detect**, and then choose **Dimensions**.
2. Select the dimension that you want to edit.
3. Choose **Actions**, and then choose **Edit**.

Created date	Dimension name	Type	Value
Mar 27, 2020 3:22:51 PM -0700	Sensor_Temperature	Topic filter	/sensor/temperature/+

To delete a dimension

1. In the [AWS IoT console](#), in the navigation pane, expand **Defend**, expand **Detect**, and then choose **Dimensions**.
2. Select the dimension that you want to delete.
3. Confirm that the dimension isn't attached to a security profile by checking the **Used in** column. If the dimension is attached to a security profile, open the **Security profiles** page on the left, and edit the security profiles that the dimension is attached to. When you delete the dimension, you also delete the behavior. If you want to keep the behavior, choose the ellipsis, then choose **Copy**. Then you can proceed with deleting the behavior. If you want to delete another dimension, follow the steps in this section.

EDIT SECURITY PROFILE

Expected behaviors STEP 1/4

Name	Description (optional)
Temperature_Profile	An optional short description

Behaviors

Specify how your device **should behave**. You can use cloud-side metrics without a device agent deployed [learn more](#) ⓘ

Note: once created, behavior names cannot be edited. ⓘ

Name ⓘ	Metric ⓘ	Dimension (optional) ⓘ	Dimension operator ⓘ	...
Sensor_failures	Authorization failures	Sensor_Temperature	In	

Check type ⓘ	Operator ⓘ	Value	Duration ⓘ
Absolute value	Greater than	5	5 minutes

Datapoints to alarm ⓘ	Datapoints to clear ⓘ
1	1

Name ⓘ	Metric ⓘ	Dimension (optional) ⓘ	Dimension operator ⓘ	...
Behavior name	Authorization failures	Select	Select	

Check type ⓘ	Operator ⓘ	Value	Duration ⓘ
Absolute value	Greater than	5	5 minutes

Datapoints to alarm ⓘ	Datapoints to clear ⓘ
1	1

Add behavior

4. Choose **Actions**, and then choose **Delete**.

How to use dimensions on the AWS CLI

To create and apply a dimension to a security profile behavior

1. First create the dimension before attaching it to a security profile. Use the [CreateDimension](#) command to create a dimension:

```
aws iot create-dimension \
--name TopicFilterForAuthMessages \
--type TOPIC_FILTER \
--string-values device/+/*/auth
```

The output of this command looks like the following:

```
{
  "arn": "arn:aws:iot:us-west-2:123456789012:dimension/TopicFilterForAuthMessages",
  "name": "TopicFilterForAuthMessages"
}
```

2. Either add the dimension to an existing security profile by using [UpdateSecurityProfile](#), or add the dimension to a new security profile by using [CreateSecurityProfile](#). In the following example, we create a new security profile that checks if messages to TopicFilterForAuthMessages are under 128 bytes, and retains the number of messages sent to non-auth topics.

```
aws iot create-security-profile \
--security-profile-name ProfileForConnectedDevice \
--security-profile-description "Check to see if messages to TopicFilterForAuthMessages are under 128 bytes and retains the number of messages sent to non-auth topics." \
--behaviors "[{\\"name\\":\\"CellularBandwidth\\",\\"metric\\":\\"aws:message-byte-size\\",\\"criteria\\":{\\"comparisonOperator\\":\\"less-than\\",\\"value\\":{\\"count\\":128},\\"consecutiveDatapointsToAlarm\\":1,\\"consecutiveDatapointsToClear\\":1},{\\"name\\":\\"Authorization\\",\\"metric\\":\\"aws:num-authorization-failures\\",\\"criteria\\":{\\"comparisonOperator\\":\\"less-than\\",\\"value\\":{\\"count\\":10},\\"durationSeconds\\":300,\\"consecutiveDatapointsToAlarm\\":1,\\"consecutiveDatapointsToClear\\":1}}}]" \
--additional-metrics-to-retain-v2 "[{\\"metric\\": \\"aws:num-authorization-failures\\",\\"metricDimension\\": {\"dimensionName\": \"TopicFilterForAuthMessages\", \"operator\": \"NOT_IN\"}}]"
```

The output of this command looks like the following:

```
{
  "securityProfileArn": "arn:aws:iot:us-west-2:123456789012:securityprofile/ProfileForConnectedDevice",
  "securityProfileName": "ProfileForConnectedDevice"
}
```

To save time, you can also load a parameter from a file instead of typing it as a command line parameter value. For more information, see [Loading AWS CLI Parameters from a File](#). The following shows the `behavior` parameter in expanded JSON format:

```
[
  {
    "criteria": {
      "comparisonOperator": "less-than",
      "consecutiveDatapointsToAlarm": 1,
      "consecutiveDatapointsToClear": 1,
      "value": {
        "count": 128
      }
    },
    "metric": "aws:message-byte-size",
    "metricDimension": {
      "dimensionName": "TopicFilterForAuthMessages"
    },
    "name": "CellularBandwidth"
```

```
    }  
]
```

To view security profiles with a dimension

- Use the [ListSecurityProfiles](#) command to view security profiles with a certain dimension:

```
aws iot list-security-profiles \  
  --dimension-name TopicFilterForAuthMessages
```

The output of this command looks like the following:

```
{  
  "securityProfileIdentifiers": [  
    {  
      "name": "ProfileForConnectedDevice",  
      "arn": "arn:aws:iot:us-west-2:1234564789012:securityprofile/  
ProfileForConnectedDevice"  
    }  
  ]  
}
```

To update your dimension

- Use the [UpdateDimension](#) command to update a dimension:

```
aws iot update-dimension \  
  --name TopicFilterForAuthMessages \  
  --string-values device/${iot:ClientId}/auth
```

The output of this command looks like the following:

```
{  
  "name": "TopicFilterForAuthMessages",  
  "lastModifiedDate": 1585866222.317,  
  "stringValues": [  
    "device/${iot:ClientId}/auth"  
  ],  
  "creationDate": 1585854500.474,  
  "type": "TOPIC_FILTER",  
  "arn": "arn:aws:iot:us-west-2:1234564789012:dimension/TopicFilterForAuthMessages"  
}
```

To delete a dimension

1. To delete a dimension, first detach it from any security profiles that it's attached to. Use the [ListSecurityProfiles](#) command to view security profiles with a certain dimension.
2. To remove a dimension from a security profile, use the [UpdateSecurityProfile](#) command. Enter all information that you want to keep, but exclude the dimension:

```
aws iot update-security-profile \  
  --security-profile-name ProfileForConnectedDevice \  
  --security-profile-description "Check to see if authorization fails 10 times in 5  
minutes or if cellular bandwidth exceeds 128" \
```

```
--behaviors "[{\\"name\\":\\"metric\\":\\"aws:message-byte-size\\",\\"criteria\\":{\\"comparisonOperator\\":\\"less-than\\",\\"value\\":{\\"count\\":128},\\"consecutiveDatapointsToAlarm\\":1,\\"consecutiveDatapointsToClear\\":1}}, {\\"name\\":\\"Authorization\\",\\"metric\\":\\"aws:num-authorization-failures\\",\\"criteria\\":{\\"comparisonOperator\\":\\"less-than\\",\\"value\\":{\\"count\\":10},\\"durationSeconds\\":300,\\"consecutiveDatapointsToAlarm\\":1,\\"consecutiveDatapointsToClear\\":1}}}]"
```

The output of this command looks like the following:

```
{  
  "behaviors": [  
    {  
      "metric": "aws:message-byte-size",  
      "name": "CellularBandwidth",  
      "criteria": {  
        "consecutiveDatapointsToClear": 1,  
        "comparisonOperator": "less-than",  
        "consecutiveDatapointsToAlarm": 1,  
        "value": {  
          "count": 128  
        }  
      }  
    },  
    {  
      "metric": "aws:num-authorization-failures",  
      "name": "Authorization",  
      "criteria": {  
        "durationSeconds": 300,  
        "comparisonOperator": "less-than",  
        "consecutiveDatapointsToClear": 1,  
        "consecutiveDatapointsToAlarm": 1,  
        "value": {  
          "count": 10  
        }  
      }  
    }  
  ],  
  "securityProfileName": "ProfileForConnectedDevice",  
  "lastModifiedDate": 1585936349.12,  
  "securityProfileDescription": "Check to see if authorization fails 10 times in 5 minutes or if cellular bandwidth exceeds 128",  
  "version": 2,  
  "securityProfileArn": "arn:aws:iot:us-west-2:123456789012:securityprofile/Preo/ProfileForConnectedDevice",  
  "creationDate": 1585846909.127  
}
```

- After the dimension is detached, use the [DeleteDimension](#) command to delete the dimension:

```
aws iot delete-dimension \  
  --name TopicFilterForAuthMessages
```

Permissions

This section contains information about how to set up the IAM roles and policies required to manage AWS IoT Device Defender Detect. For more information, see the [IAM User Guide](#).

Give AWS IoT Device Defender detect permission to publish alarms to an SNS topic

If you use the `alertTargets` parameter in [CreateSecurityProfile](#), you must specify an IAM role with two policies: a permissions policy and a trust policy. The permissions policy grants permission to AWS IoT Device Defender to publish notifications to your SNS topic. The trust policy grants AWS IoT Device Defender permission to assume the required role.

Permission policy

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "sns:Publish"  
            ],  
            "Resource": [  
                "arn:aws:sns:region:account-id:your-topic-name"  
            ]  
        }  
    ]  
}
```

Trust policy

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "",  
            "Effect": "Allow",  
            "Principal": {  
                "Service": "iot.amazonaws.com"  
            },  
            "Action": "sts:AssumeRole"  
        }  
    ]  
}
```

Pass role policy

You also need an IAM permissions policy attached to the IAM user that allows the user to pass roles. See [Granting a User Permissions to Pass a Role to an AWS Service](#).

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "",  
            "Effect": "Allow",  
            "Action": [  
                "iam:GetRole",  
                "iam:PassRole"  
            ],  
            "Resource": "arn:aws:iam::account-id:role/Role_To_Pass"  
        }  
    ]  
}
```

}

Detect commands

You can use the Detect commands in this section to configure ML Detect or Rules Detect Security Profiles, to identify and monitor unusual behaviors that may indicate a compromised device.

DetectMitigation action commands

Start and manage Detect execution
CancelDetectMitigationActionsTask
DescribeDetectMitigationActionsTask
ListDetectMitigationActionsTasks
StartDetectMitigationActionsTask
ListDetectMitigationActionsExecutions

Dimension action commands

Start and manage Dimension execution
CreateDimension
DescribeDimension
ListDimensions
DeleteDimension
UpdateDimension

CustomMetric action commands

Start and manage CustomMetric execution
CreateCustomMetric
UpdateCustomMetric
DescribeCustomMetric
ListCustomMetrics
DeleteCustomMetric

Security Profile action commands

Start and manage Security Profile execution
CreateSecurityProfile
AttachSecurityProfile
DeleteSecurityProfile

Start and manage Security Profile execution

[DescribeSecurityProfile](#)

[ListTargetsForSecurityProfile](#)

[UpdateSecurityProfile](#)

[ValidateSecurityProfileBehaviors](#)

[ListSecurityProfilesForTarget](#)

Alarm action commands

Manage alarms and targets

[ListActiveViolations](#)

[ListViolationEvents](#)

[DetachSecurityProfile](#)

ML Detect action commands

List ML model training data

[GetBehaviorModelTrainingSummaries](#)

How to use AWS IoT Device Defender detect

1. You can use AWS IoT Device Defender Detect with just cloud-side metrics, but if you plan to use device-reported metrics, you must first deploy the AWS IoT SDK on your AWS IoT connected devices or device gateways. For more information, see [Sending metrics from devices \(p. 912\)](#).
2. Consider viewing the metrics that your devices generate before you define behaviors and create alarms. AWS IoT can collect metrics from your devices so you can first identify usual or unusual behavior for a group of devices, or for all devices in your account. Use [CreateSecurityProfile](#), but specify only those `additionalMetricsToRetain` that you're interested in. Don't specify behaviors at this point.

Use the AWS IoT console to look at your device metrics to see what constitutes typical behavior for your devices.
3. Create a set of behaviors for your security profile. Behaviors contain metrics that specify normal behavior for a group of devices or for all devices in your account. For more information and examples, see [Cloud-side metrics \(p. 912\)](#) and [Device-side metrics \(p. 897\)](#). After you create a set of behaviors, you can validate them with [ValidateSecurityProfileBehaviors](#).
4. Use the [CreateSecurityProfile](#) action to create a security profile that includes your behaviors. You can use the `alertTargets` parameter to have alarms sent to a target (an SNS topic) when a device violates a behavior. (If you send alarms using SNS, be aware that these count against your AWS account's SNS topic quota. It's possible that a large burst of violations can exceed your SNS topic quota. You can also use CloudWatch metrics to check for violations. For more information, see [Using AWS IoT metrics \(p. 407\)](#).)
5. Use the [AttachSecurityProfile](#) action to attach the security profile to a group of devices (a thing group), all registered things in your account, all unregistered things, or all devices. AWS IoT Device Defender Detect starts checking for abnormal behavior and, if any behavior violations are detected,

sends alarms. You might want to attach a security profile to all unregistered things if, for example, you expect to interact with mobile devices that are not in your account's thing registry. You can define different sets of behaviors for different groups of devices to meet your needs.

To attach a security profile to a group of devices, you must specify the ARN of the thing group that contains them. A thing group ARN has the following format.

```
arn:aws:iot:<region>:<account-id>:thinggroup/<thing-group-name>
```

To attach a security profile to all of the registered things in an AWS account (ignoring unregistered things), you must specify an ARN with the following format.

```
arn:aws:iot:<region>:<account-id>:all/registered-things
```

To attach a security profile to all unregistered things, you must specify an ARN with the following format.

```
arn:aws:iot:<region>:<account-id>:all/unregistered-things
```

To attach a security profile to all devices, you must specify an ARN with the following format.

```
arn:aws:iot:<region>:<account-id>:all/things
```

6. You can also keep track of violations with the [ListActiveViolations](#) action, which lets you to see which violations were detected for a given security profile or target device.

Use the [ListViolationEvents](#) action to see which violations were detected during a specified time period. You can filter these results by security profile, device, or alarm verification state.

7. You can verify, organize, and manage your alarms, by marking their verification state and providing a description of that verification state, by using the [PutVerificationStateOnViolation](#) action.
8. If your devices violate the defined behaviors too often, or not often enough, you should fine-tune the behavior definitions.
9. To review the security profiles that you set up and the devices that are being monitored, use the [ListSecurityProfiles](#), [ListSecurityProfilesForTarget](#), and [ListTargetsForSecurityProfile](#) actions.

Use the [DescribeSecurityProfile](#) action to get more details about a security profile.

10. To update a security profile, use the [UpdateSecurityProfile](#) action. Use the [DetachSecurityProfile](#) action to detach a security profile from an account or target thing group. Use the [DeleteSecurityProfile](#) action to delete a security profile entirely.

Mitigation actions

You can use AWS IoT Device Defender to take actions to mitigate issues that were found in an Audit finding or Detect alarm.

Note

Mitigation actions won't be performed on suppressed audit findings. For more information about audit finding suppressions, see [Audit finding suppressions \(p. 870\)](#).

Audit mitigation actions

AWS IoT Device Defender provides predefined actions for the different audit checks. You configure those actions for your AWS account and then apply them to a set of findings. Those findings can be:

- All findings from an audit. This option is available in both the AWS IoT console and by using the AWS CLI.
- A list of individual findings. This option is only available by using the AWS CLI.
- A filtered set of findings from an audit.

The following table lists the types of audit checks and the supported mitigation actions for each:

Audit check to mitigation action mapping

Audit check	Supported mitigation actions
REVOKED_CA_CERT_CHECK	PUBLISH_FINDING_TO_SNS, UPDATE_CA_CERTIFICATE
DEVICE_CERTIFICATE_SHARED_CHECK	PUBLISH_FINDING_TO_SNS, UPDATE_DEVICE_CERTIFICATE, ADD_THINGS_TO_THING_GROUP
UNAUTHENTICATED_COGNITO_ROLE_OVERLY_PERMISSIVE_CHECK	PUBLISH_FINDING_TO_SNS
AUTHENTICATED_COGNITO_ROLE_OVERLY_PERMISSIVE_CHECK	PUBLISH_FINDING_TO_SNS
IOT_POLICY_OVERLY_PERMISSIVE_CHECK	PUBLISH_FINDING_TO_SNS, REPLACE_DEFAULT_POLICY_VERSION
CA_CERT_APPROACHING_EXPIRATION_CHECK	PUBLISH_FINDING_TO_SNS, UPDATE_CA_CERTIFICATE
CONFLICTING_CLIENT_IDS_CHECK	PUBLISH_FINDING_TO_SNS
DEVICE_CERT_APPROACHING_EXPIRATION_CHECK	PUBLISH_FINDING_TO_SNS, UPDATE_DEVICE_CERTIFICATE, ADD_THINGS_TO_THING_GROUP
REVOKED_DEVICE_CERT_CHECK	PUBLISH_FINDING_TO_SNS, UPDATE_DEVICE_CERTIFICATE, ADD_THINGS_TO_THING_GROUP
LOGGING_DISABLED_CHECK	PUBLISH_FINDING_TO_SNS, ENABLE_IOT_LOGGING
DEVICE_CERTIFICATE_KEY_QUALITY_CHECK	PUBLISH_FINDING_TO_SNS, UPDATE_DEVICE_CERTIFICATE, ADD_THINGS_TO_THING_GROUP
CA_CERTIFICATE_KEY_QUALITY_CHECK	PUBLISH_FINDING_TO_SNS, UPDATE_CA_CERTIFICATE
IOT_ROLE_ALIAS_OVERLY_PERMISSIVE_CHECK	PUBLISH_FINDING_TO_SNS
IOT_ROLE_ALIAS_ALLOWS_ACCESS_TO_UNUSED_SOURCES_CHECK	PUBLISH_FINDING_TO_SNS

All audit checks support publishing the audit findings to Amazon SNS so you can take custom actions in response to the notification. Each type of audit check can support additional mitigation actions:

REVOKED_CA_CERT_CHECK

- Change the state of the certificate to mark it as inactive in AWS IoT.

DEVICE_CERTIFICATE_SHARED_CHECK

- Change the state of the device certificate to mark it as inactive in AWS IoT.
- Add the devices that use that certificate to a thing group.

UNAUTHENTICATED_COGNITO_ROLE_OVERLY_PERMISSIVE_CHECK

- No additional supported actions.

AUTHENTICATED_COGNITO_ROLE_OVERLY_PERMISSIVE_CHECK

- No additional supported actions.

IOT_POLICY_OVERLY_PERMISSIVE_CHECK

- Add a blank AWS IoT policy version to restrict permissions.

CA_CERT_APPROACHING_EXPIRATION_CHECK

- Change the state of the certificate to mark it as inactive in AWS IoT.

CONFLICTING_CLIENT_IDS_CHECK

- No additional supported actions.

DEVICE_CERT_APPROACHING_EXPIRATION_CHECK

- Change the state of the device certificate to mark it as inactive in AWS IoT.
- Add the devices that use that certificate to a thing group.

DEVICE_CERTIFICATE_KEY_QUALITY_CHECK

- Change the state of the device certificate to mark it as inactive in AWS IoT.
- Add the devices that use that certificate to a thing group.

CA_CERTIFICATE_KEY_QUALITY_CHECK

- Change the state of the certificate to mark it as inactive in AWS IoT.

REVOKED_DEVICE_CERT_CHECK

- Change the state of the device certificate to mark it as inactive in AWS IoT.
- Add the devices that use that certificate to a thing group.

LOGGING_DISABLED_CHECK

- Enable logging.

AWS IoT Device Defender supports the following types of mitigation actions on Audit findings:

Action type	Notes
ADD_THINGS_TO_THING_GROUP	You specify the group to which you want to add the devices. You also specify whether membership in one or more dynamic groups should be overridden if that would exceed the maximum number of groups to which the thing can belong.
ENABLE_IOT_LOGGING	You specify the logging level and the role with permissions for logging. You cannot specify a logging level of DISABLED.
PUBLISH_FINDING_TO_SNS	You specify the topic to which the finding should be published.
REPLACE_DEFAULT_POLICY_VERSION	You specify the template name. Replaces the policy version with a default or blank policy. Only a value of BLANK_POLICY is currently supported.

Action type	Notes
UPDATE_CA_CERTIFICATE	You specify the new state for the CA certificate. Only a value of <code>DEACTIVATE</code> is currently supported.
UPDATE_DEVICE_CERTIFICATE	You specify the new state for the device certificate. Only a value of <code>DEACTIVATE</code> is currently supported.

By configuring standard actions when issues are found during an audit, you can respond to those issues consistently. Using these defined mitigation actions also helps you resolve the issues more quickly and with less chance of human error.

Important

Applying mitigation actions that change certificates, add things to a new thing group, or replace the policy can have an impact on your devices and applications. For example, devices might be unable to connect. Consider the implications of the mitigation actions before you apply them. You might need to take other actions to correct the problems before your devices and applications can function normally. For example, you might need to provide updated device certificates. Mitigation actions can help you quickly limit your risk, but you must still take corrective actions to address the underlying issues.

Some actions, such as reactivating a device certificate, can only be performed manually. AWS IoT Device Defender does not provide a mechanism to automatically roll back mitigation actions that have been applied.

Detect mitigation actions

AWS IoT Device Defender supports the following types of mitigation actions on Detect alarms:

Action type	Notes
ADD_THINGS_TO_THING_GROUP	You specify the group to which you want to add the devices. You also specify whether membership in one or more dynamic groups should be overridden if that would exceed the maximum number of groups to which the thing can belong.

How to define and manage mitigation actions

You can use the AWS IoT console or the AWS CLI to define and manage mitigation actions for your AWS account.

Create mitigation actions

Each mitigation action that you define is a combination of a predefined action type and parameters specific to your account.

To use the AWS IoT console to create mitigation actions

1. Open the [AWS IoT console](#).
2. In the left navigation pane, choose **Defend**, and then choose **Mitigation Actions**.
3. On the **Mitigation Actions** page, choose **Create**.

Create a new mitigation action

You can use AWS IoT Device Defender to take actions to mitigate issues that were found during an audit. AWS IoT Device Defender provides predefined actions for the different audit checks. You can configure those actions for your AWS account and then apply them to a set of findings. [Learn more](#)

Action name [?](#)
EnableErrorLoggingAction

Action type [?](#)
Enable IoT logging

Permissions

Please create or select a role with the following mitigation action type specific permission(s) and trust relationship.

Required permissions: [Manage your service permissions](#)

- ▶ Permissions
- ▶ Trust relationships

You can also attach an action specific managed policy to an existing role, or create a new role with the required managed policy attached.

Action execution role [?](#)
IoTMitigationActionErrorLoggingRole Managed policy attached ✓ [Create Role](#) [Select](#)

Parameters

Role for logging [?](#)
AWSIoTLoggingRole [Clear](#) [Select](#)

Log level [?](#)
Error

Tags

Tag name
Provide a tag name, e.g. Manufacturer [Clear](#)

Value
Provide a tag value, e.g. Acme-Corporation [Clear](#)

[Add another](#)

[Cancel](#) [Save](#)

4. On the **Create a Mitigation Action** page, in **Action name**, enter a unique name for your mitigation action.
5. In **Action type**, specify the type of action that you want to define.
6. Each action type requests a different set of parameters. Enter the parameters for the action. For example, if you choose the **Add things to thing group** action type, choose the destination group and select or clear **Override dynamic groups**.
7. In **Action execution role**, choose the role under whose permissions the action is applied.
8. Choose **Save** to save your mitigation action to your AWS account.

To use the AWS CLI to create mitigation actions

- Use the [CreateMitigationAction](#) command to create your mitigation action. The unique name that you give the action is used when you apply that action to audit findings. Choose a meaningful name.

To use the AWS IoT console to view and modify mitigation actions

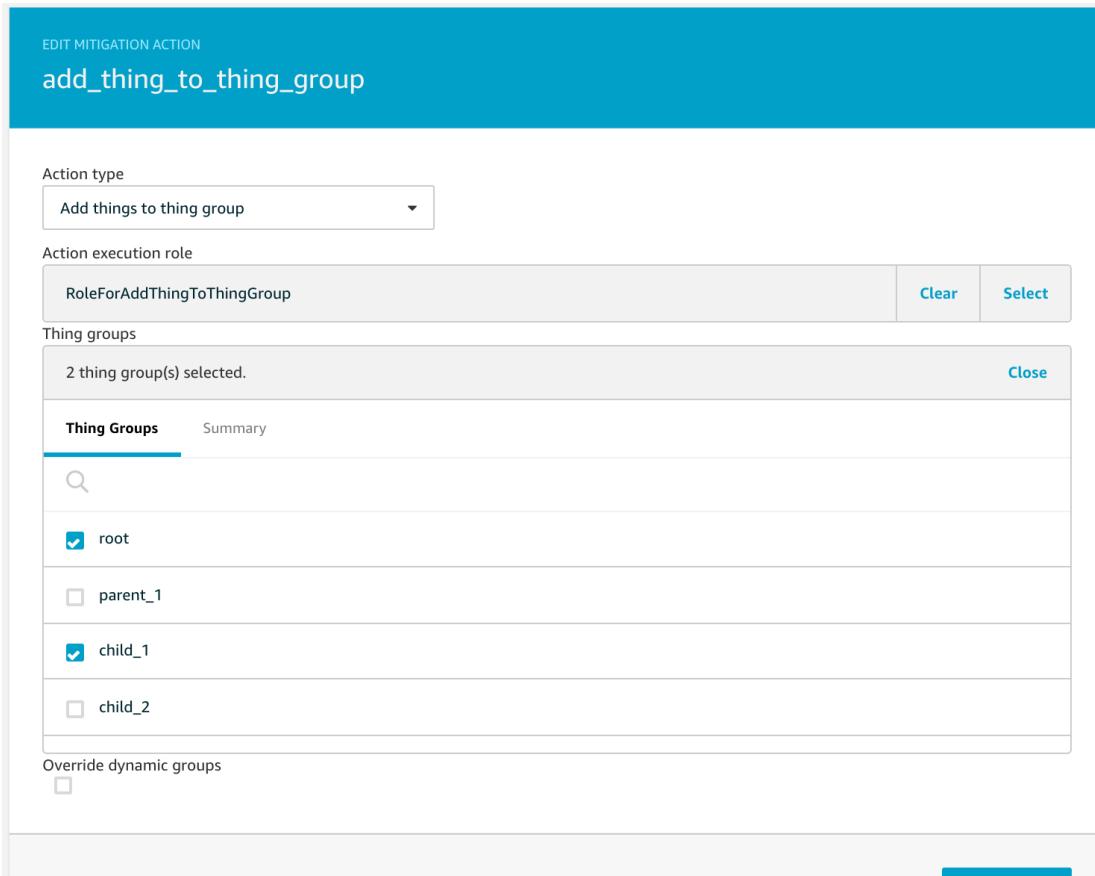
1. Open the [AWS IoT console](#).
2. In the left navigation pane, choose **Defend**, and then choose **Mitigation Actions**.

The **Mitigation Actions** page displays a list of all of the mitigation actions that are defined for your AWS account.

The screenshot shows the AWS IoT Device Defender interface with the 'Mitigation actions' section. At the top right are 'Create', 'Bell', and 'More' buttons. Below is a table with four rows of mitigation actions:

Created date	Action name	ARN	...
Jun 10, 2019 10:09:53 AM -0700	enable_logging	arn:aws:iot:us-east-1:██:mitigationa...
Jun 6, 2019 6:08:47 PM -0700	sns_publish	arn:aws:iot:us-east-1:██:mitigationa...
Jun 6, 2019 6:08:26 PM -0700	replace_default_policy_version	arn:aws:iot:us-east-1:██:mitigationa...
Jun 3, 2019 10:51:16 PM -0700	add_thing_to_thing_group	arn:aws:iot:us-east-1:██:mitigationa...

3. Choose the action name link for the mitigation action that you want to change.
4. Make your changes to the mitigation action. Because the name of the mitigation action is used to identify it, you cannot change the name.



- Choose **Save** to save the changes to the mitigation action to your AWS account.

To use the AWS CLI to list a mitigation action

- Use the [ListMitigationAction](#) command to list your mitigation actions. If you want to change or delete a mitigation action, make a note of the name.

To use the AWS CLI to update a mitigation action

- Use the [UpdateMitigationAction](#) command to change your mitigation action.

To use the AWS IoT console to delete a mitigation action

- Open the [AWS IoT console](#).
- In the left navigation pane, choose **Defend**, and then choose **Mitigation Actions**.

The **Mitigation Actions** page displays all of the mitigation actions that are defined for your AWS account.

- Choose the ellipsis (...) for the mitigation action that you want to delete, and then choose **Delete**.

To use the AWS CLI to delete mitigation actions

- Use the [UpdateMitigationAction](#) command to change your mitigation action.

To use the AWS IoT console to view mitigation action details

1. Open the [AWS IoT console](#).
2. In the left navigation pane, choose **Defend**, and then choose **Mitigation Actions**.

The screenshot shows the 'Mitigation actions' page in the AWS IoT Device Defender console. At the top, there's a breadcrumb trail: 'Device Defender > Mitigation actions'. On the right, there are 'Create' and 'Bell' icons. Below the header, it says 'Mitigation actions (4)'. A table lists the actions:

Created date	Action name	ARN	...
Jun 10, 2019 10:09:53 AM -0700	enable_logging	arn:aws:iot:us-east-1:...:mitigationa...	...
Jun 6, 2019 6:08:47 PM -0700	sns_publish	arn:aws:iot:us-east-1:...:mitigationa...	...
Jun 6, 2019 6:08:26 PM -0700	replace_default_policy_version	arn:aws:iot:us-east-1:...:mitigationa...	...
Jun 3, 2019 10:51:16 PM -0700	add_thing_to_thing_group	arn:aws:iot:us-east-1:...:mitigationa...	...

The **Mitigation Actions** page displays all of the mitigation actions that are defined for your AWS account.

3. Choose the action name link for the mitigation action that you want to change.
4. In the **Are you sure you want to delete the mitigation action** window, choose **Confirm**.

The screenshot shows a confirmation dialog box. The title bar says 'Are you sure you want to delete the mitigation action?'. The main body text reads: 'You will no longer be able to use this action to mitigate non-compliant resources identified by AWS IoT Device Defender.' At the bottom, there are 'Cancel' and 'Confirm' buttons. A status bar at the bottom indicates 'Role: admin'.

To use the AWS CLI to view mitigation action details

- Use the [DescribeMitigationAction](#) command to view details for your mitigation action.

Apply mitigation actions

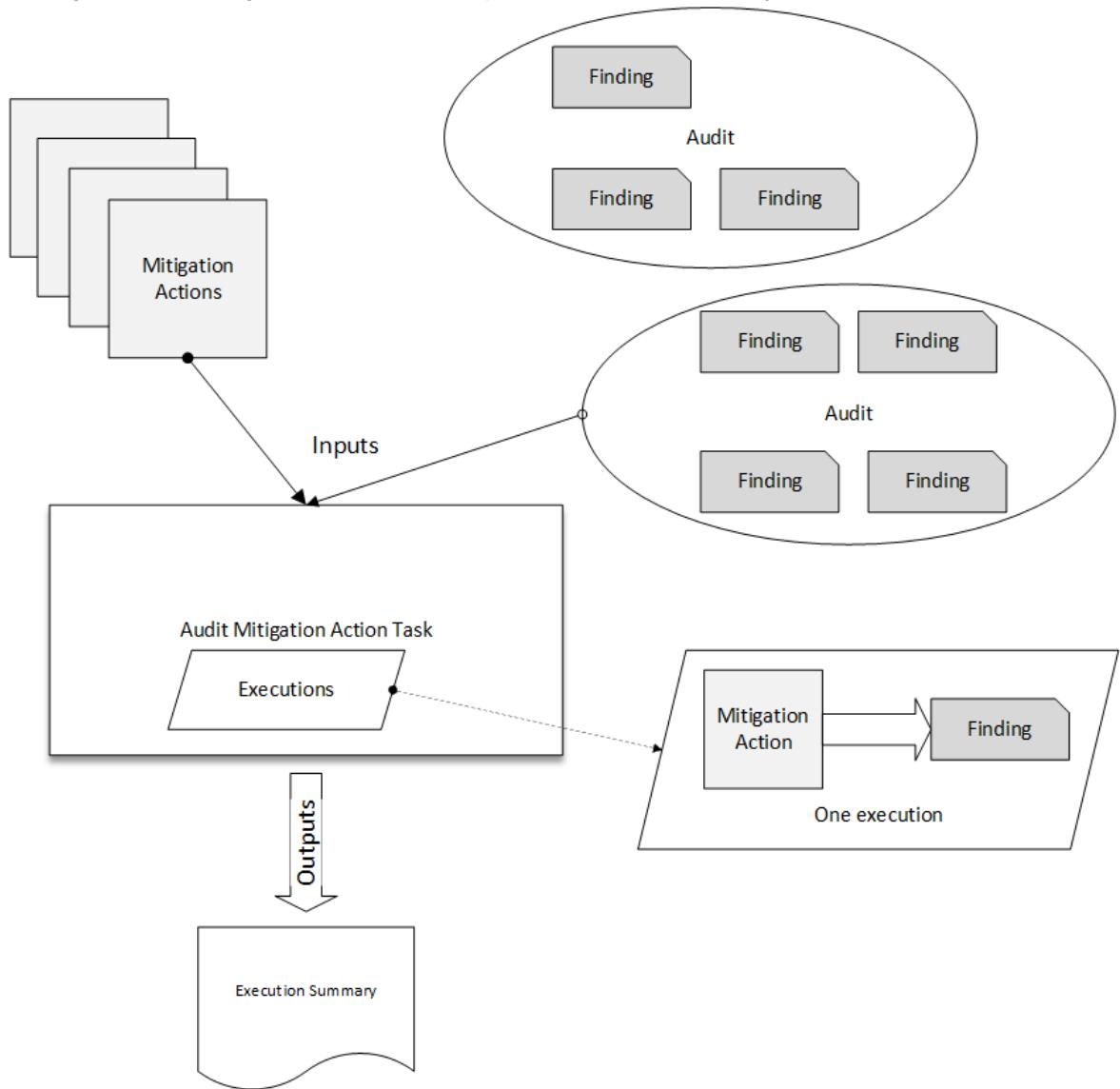
After you have defined a set of mitigation actions, you can apply those actions to the findings from an audit. When you apply actions, you start an audit mitigation actions task. This task might take some time to complete, depending on the set of findings and the actions that you apply to them. For example, if you have a large pool of devices whose certificates have expired, it might take some time to deactivate all of those certificates or to move those devices to a quarantine group. Other actions, such as enabling logging, can be completed quickly.

You can view the list of action executions and cancel an execution that has not yet been completed. Actions already performed as part of the canceled action execution are not rolled back. If you are

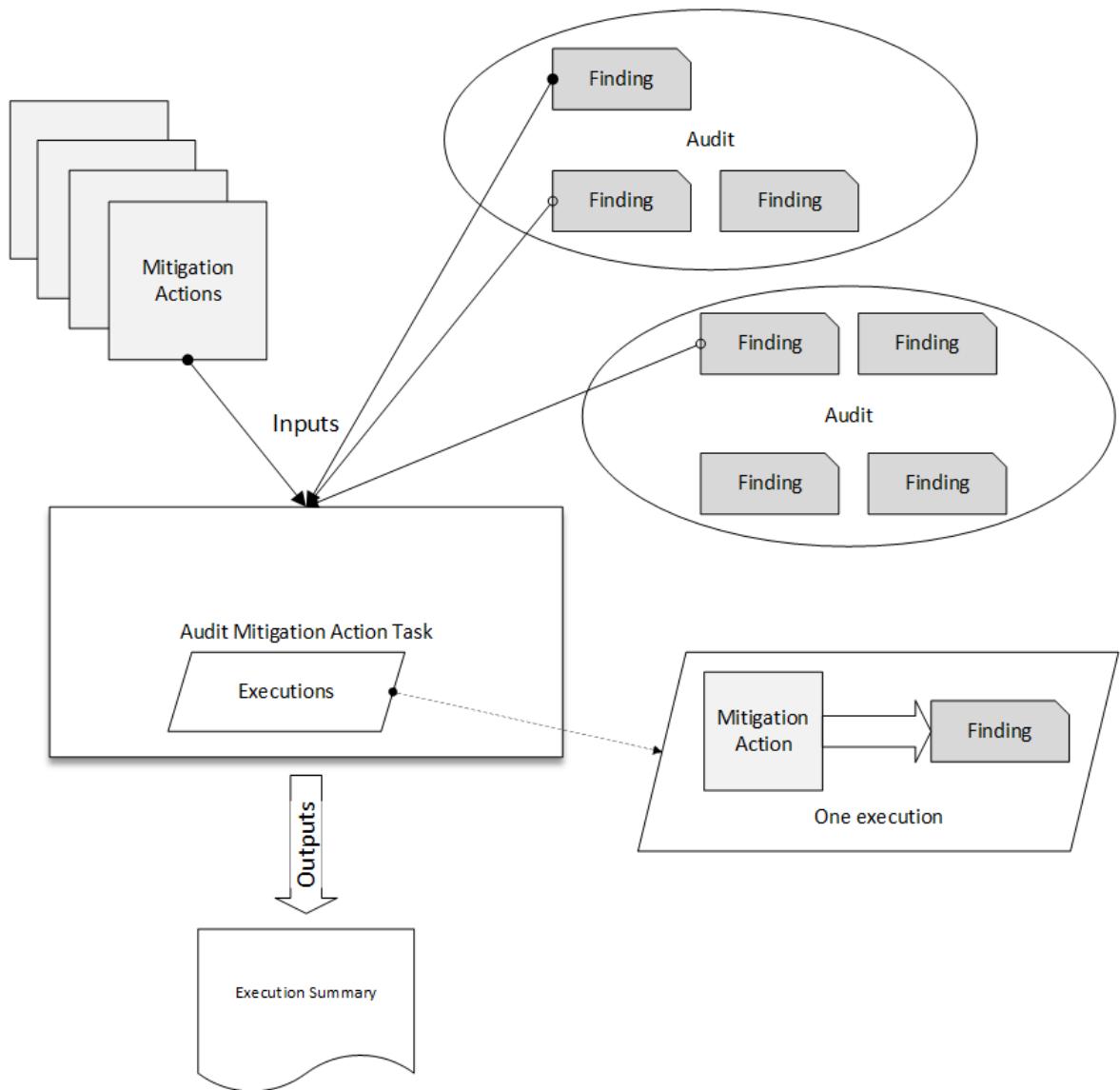
applying multiple actions to a set of findings and one of those actions failed, the subsequent actions are skipped for that finding (but are still applied to other findings). The task status for the finding is FAILED. The taskStatus is set to failed if one or more of the actions failed when applied to the findings. Actions are applied in the order in which they are specified.

Each action execution applies a set of actions to a target. That target can be a list of findings or it can be all findings from an audit.

The following diagram shows how you can define an audit mitigation task that takes all findings from one audit and applies a set of actions to those findings. A single execution applies one action to one finding. The audit mitigation actions task outputs an execution summary.



The following diagram shows how you can define an audit mitigation task that takes a list of individual findings from one or more audits and applies a set of actions to those findings. A single execution applies one action to one finding. The audit mitigation actions task outputs an execution summary.



You can use the AWS IoT console or the AWS CLI to apply mitigation actions.

To use the AWS IoT console to apply mitigation actions by starting an action execution

1. Open the [AWS IoT console](#).
2. In the left navigation pane, choose **Defend**, choose **Audit**, and then choose **Results**.

The screenshot shows the AWS IoT Device Defender Audit Results page. At the top, it displays the path: Device Defender > Audit > Results > On-demand. Below that, it shows the date and time: On-demand - Jun 9, 2019 10:26:36 PM -0700. A blue button labeled "Start mitigation actions" is visible on the right.

Audit findings

Audit task ID ee58d7b18ec45be15f5596958dc2a2ff Started at Jun 9, 2019 10:26:36 PM -0700

Non-compliant checks (5 of 10)

Check name	Severity	Non-compliant	% Resources	Mitigation
CA certificate revoked but device certificates still a...	Critical	79	83.2%	Review & deactivate (i)
CA certificate expiring	Medium	79	83.2%	Reprovision & deactivate (i)
Device certificate expiring	Medium	50	92.6%	Reprovision & deactivate (i)
Revoked device certificate still active	Medium	50	92.6%	Reprovision & revoke (i)
Logging disabled	Low	1	100%	Enable logging (i)

3. Choose the name for the audit to which you want to apply actions.

The screenshot shows the "Start a new mitigation action" dialog box. It includes fields for "Task name" (set to 8115481b332374e3309c13050320323c), "Select options for Device certificate expiring", and "Select reason codes". A note on the right explains that each mitigation action task is identified by a unique user-provided name. A "Help" link is also present.

4. Choose Start Mitigation Actions. This button is not available if all of your checks are compliant.

5. In **Are you sure that you want to start mitigation action task**, the task name defaults to the audit ID, but you can change it to something more meaningful.
6. For each type of check that had one or more noncompliant findings in the audit, you can choose one or more actions to apply. Only actions that are valid for the check type are displayed.

Note

If you have not configured actions for your AWS account, the list of actions is empty. You can choose the [click here](#) link to create one or more mitigation actions.

7. When you have specified all of the actions that you want to apply, choose **Confirm**.

To use the AWS CLI to apply mitigation actions by starting an audit mitigation actions execution

1. If you want to apply actions to all findings for the audit, use the [ListAuditTasks](#) command to find the task ID.

2. If you want to apply actions to selected findings only, use the [ListAuditFindings](#) command to get the finding IDs.
3. Use the [ListMitigationActions](#) command and make note of the names of the mitigation actions that you want to apply.
4. Use the [StartAuditMitigationActionsTask](#) command to apply actions to the target. Make note of the task ID. You can use the ID to check the state of the action execution, review the details, or cancel it.

To use the AWS IoT console to view your action executions

1. Open the [AWS IoT console](#).
2. In the left navigation pane, choose **Defend**, and then choose **Action Executions**.

Action tasks (1)		
Date	Name	Status
Jun 6, 2019 6:09:07 PM -0700	ff82164a6439e6024e83b4fc104817d7	Completed

A list of action tasks shows when each was started and the current status.

3. Choose the **Name** link to see details for the task. The details include all of the actions that are applied by the task, their target, and their status.

MITIGATION ACTION EXECUTION TASK ff82164a6439e6024e83b4fc104817d7						
Details						
Status COMPLETED						
Started at Jun 6, 2019 6:09:07 PM -0700						
Completed at Jun 6, 2019 6:09:09 PM -0700						
Check summary						
Check name	Failed	Successful	Skipped	Canceled	Total	Executions
IoT policies overly permissive	0	2	0	0	2	Show

You can use the **Show executions for** filters to focus on types of actions or action states.

4. To see details for the task, in **Executions**, choose **Show**.

The screenshot shows the AWS Device Defender Audit Action executions page. At the top, it displays the path: Device Defender > Audit > Action executions > ff82164a6439e6024e83b4fc104817d7 >. Below this, a dark header bar contains the text "MITIGATION ACTION EXECUTION TASK" and the ID "ff82164a6439e6024e83b4fc104817d7". The main content area has a heading "IoT policies overly permissive". Underneath, there's a section titled "Action executions (4)" with a "Show executions for" dropdown set to "All actions" and "All status". A table below lists four completed actions from June 6, 2019, at 6:09:08 PM -0700, each with an "sns_publish" finding.

Started at	Status	Action	Finding
Jun 6, 2019 6:09:08 PM -0700	Completed	sns_publish	053cff17-1da4-4479-996b-8b...
Jun 6, 2019 6:09:08 PM -0700	Completed	replace_default_policy_version	053cff17-1da4-4479-996b-8b...
Jun 6, 2019 6:09:08 PM -0700	Completed	replace_default_policy_version	2b966f76-b499-4986-836c-f8...

To use the AWS CLI to list your started tasks

1. Use [ListAuditMitigationActionsTasks](#) to view your audit mitigation actions tasks. You can provide filters to narrow the results. If you want to view details of the task, make note of the task ID.
2. Use [ListAuditMitigationActionsExecutions](#) to view execution details for a particular audit mitigation actions task.
3. Use [DescribeAuditMitigationActionsTask](#) to view details about the task, such as the parameters specified when it was started.

To use the AWS CLI to cancel a running audit mitigation actions task

1. Use the [ListAuditMitigationActionsTasks](#) command to find the task ID for the task whose execution you want to cancel. You can provide filters to narrow the results.
2. Use the [ListDetectMitigationActionsExecutions](#) command, using the task ID, to cancel your audit mitigation actions task. You cannot cancel tasks that have been completed. When you cancel a task, remaining actions are not applied, but mitigation actions that were already applied are not rolled back.

Permissions

For each mitigation action that you define, you must provide the role used to apply that action.

Permissions for mitigation actions

Action type	Permissions policy template
UPDATE_DEVICE_CERTIFICATE	{ "Version": "2012-10-17", "Statement": [

Action type	Permissions policy template
	<pre>{ "Effect": "Allow", "Action": ["iot:UpdateCertificate"], "Resource": ["*"] }</pre>
UPDATE_CA_CERTIFICATE	<pre>{ "Version": "2012-10-17", "Statement": [{ "Effect": "Allow", "Action": ["iot:UpdateCACertificate"], "Resource": ["*"] }] }</pre>
ADD_THINGS_TO_THING_GROUP	<pre>{ "Version": "2012-10-17", "Statement": [{ "Effect": "Allow", "Action": ["iot>ListPrincipalThings", "iot>AddThingToThingGroup"], "Resource": ["*"] }] }</pre>

Action type	Permissions policy template
REPLACE_DEFAULT_POLICY_VERSION	<pre>{ "Version": "2012-10-17", "Statement": [{ "Effect": "Allow", "Action": ["iot:CreatePolicyVersion"], "Resource": ["*"] }] }</pre>
ENABLE_IOT_LOGGING	<pre>{ "Version": "2012-10-17", "Statement": [{ "Effect": "Allow", "Action": ["iot:SetV2LoggingOptions"], "Resource": ["*"] }, { "Effect": "Allow", "Action": ["iam:PassRole"], "Resource": ["<IAM role ARN used for setting up logging>"] }] }</pre>

Action type	Permissions policy template	
PUBLISH_FINDING_TO_SNS	<pre>{ "Version": "2012-10-17", "Statement": [{ "Effect": "Allow", "Action": ["sns:Publish"], "Resource": ["<The SNS topic to which the finding is published>"] }] }</pre>	

For all mitigation action types, use the following trust policy template:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "",
            "Effect": "Allow",
            "Principal": {
                "Service": "iot.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

Mitigation action commands

You can use these mitigation action commands to define a set of actions for your AWS account that you can later apply to one or more sets of audit findings. There are three command categories:

- Those used to define and manage actions.
- Those used to start and manage the application of those actions to Audit findings.
- Those used to start and manage the application of those actions to Detect alarms.

Mitigation action commands

Define and manage actions	Start and manage Audit execution	Start and manage Detect execution
CreateMitigationAction	CancelAuditMitigationActionsTask	CancelDetectMitigationActionsTask
DeleteMitigationAction	DescribeAuditMitigationActionsTask	DescribeDetectMitigationActionsTask

Define and manage actions	Start and manage Audit execution	Start and manage Detect execution
DescribeMitigationAction	ListAuditMitigationActionsTasks	ListDetectMitigationActionsTasks
ListMitigationActions	StartAuditMitigationActionsTask	StartDetectMitigationActionsTask
UpdateMitigationAction	ListAuditMitigationActionsExecutions	ListDetectMitigationActionsExecutions

Using AWS IoT Device Defender with other AWS services

Using AWS IoT Device Defender with devices running AWS IoT Greengrass

AWS IoT Greengrass provides pre-built integration with AWS IoT Device Defender to monitor device behaviors on an ongoing basis.

- [Integrate Device Defender with AWS IoT Greengrass V1](#)
- [Integrate Device Defender with AWS IoT Greengrass V2](#)

Using AWS IoT Device Defender with FreeRTOS and embedded devices

To use AWS IoT Device Defender on a FreeRTOS device, your device must have the [FreeRTOS Embedded C SDK](#) or the [AWS IoT Device Defender library](#) installed. The FreeRTOS Embedded C SDK includes the AWS IoT Device Defender library. For information about how to integrate AWS IoT Device Defender with your FreeRTOS devices, see the following demos:

- [AWS IoT Device Defender for FreeRTOS standard metrics and custom metrics demos](#)
- [Using MQTT agent to submit metrics to AWS IoT Device Defender](#)
- [Using the MQTT core library to submit metrics to AWS IoT Device Defender](#)

To use AWS IoT Device Defender on an embedded device without FreeRTOS, your device must have the [AWS IoT Embedded C SDK](#) or [AWS IoT Device Defender library](#). The AWS IoT Embedded C SDK includes the AWS IoT Device Defender library. For information about how to integrate AWS IoT Device Defender with your embedded devices, see the following demos, [AWS IoT Device Defender for AWS IoT Embedded SDK standard and custom metrics demos](#)

Security best practices for device agents

Least Privilege

The agent process should be granted only the minimum permissions required to perform its duties.

Basic mechanisms

- Agent should be run as non-root user.

- Agent should run as a dedicated user, in its own group.
- User/groups should be granted read-only permissions on the resources required to gather and transmit metrics.
- Example: read-only on /proc /sys for the sample agent.
- For an example of how to set up a process to run with reduced permissions, see the setup instructions that are included with the Python sample agent.

There are a number of well-known Linux mechanisms that can help you further restrict or isolate your agent process:

Advanced mechanisms

- [CGroups](#)
- [SELinux](#)
- [Chroot](#)
- [Linux Namespaces](#)

Operational Resiliency

An agent process must be resilient to unexpected operational errors and exceptions and must not crash or exit permanently. The code needs to gracefully handle exceptions and, as a precaution, it must be configured to automatically restart in case of unexpected termination (for example, due to system restarts or uncaught exceptions).

Least Dependencies

An agent must use the least possible number of dependencies (that is, third-party libraries) in its implementation. If use of a library is justified due to the complexity of a task (for example, transport layer security), use only well-maintained dependencies and establish a mechanism to keep them up to date. If the added dependencies contain functionality not used by the agent and active by default (for example, opening ports, domain sockets), disable them in your code or by means of the library's configuration files.

Process Isolation

An agent process must only contain functionality required for performing device metric gathering and transmission. It must not piggyback on other system processes as a container or implement functionality for other out of scope use cases. In addition, the agent process must refrain from creating inbound communication channels such as domain socket and network service ports that would allow local or remote processes to interfere with its operation and impact its integrity and isolation.

Stealthiness

An agent process must not be named with keywords such as security, monitoring, or audit indicating its purpose and security value. Generic code names or random and unique-per-device process names are preferred. The same principle must be followed in naming the directory in which the agent's binaries reside and any names and values of process arguments.

Least Information Shared

Any agent artifacts deployed to devices must not contain sensitive information such as privileged credentials, debugging and dead code, or inline comments or documentation files that reveal details about server-side processing of agent-gathered metrics or other details about backend systems.

Transport Layer Security

To establish TLS secure channels for data transmission, an agent process must enforce all client-side validations, such as certificate chain and domain name validation, at the application level, if not enabled by default. Furthermore, an agent must use a root certificate store that contains trusted authorities and does not contain certificates belonging to compromised certificate issuers.

Secure Deployment

Any agent deployment mechanism, such as code push or sync and repositories containing its binaries, source code and any configuration files (including trusted root certificates), must be access-controlled to prevent unauthorized code injection or tampering. If the deployment mechanism relies on network communication, then use cryptographic methods to protect the integrity of deployment artifacts in transit.

Further Reading

- [Security in AWS IoT \(p. 278\)](#)
- [Understanding the AWS IoT Security Model](#)
- [Redhat: A Bite of Python](#)
- [10 common security gotchas in Python and how to avoid them](#)
- [What Is Least Privilege & Why Do You Need It?](#)
- [OWASP Embedded Security Top 10](#)
- [OWASP IoT Project](#)

Device Advisor

Device Advisor is a cloud-based, fully managed test capability for validating IoT devices during device software development. Device Advisor provides pre-built tests that you can use to validate IoT devices for reliable and secure connectivity with AWS IoT Core, before deploying devices to production. Device Advisor's pre-built tests help you validate your device software against best practices for usage of [TLS](#), [MQTT](#), [Device Shadow](#), and [IoT Jobs](#). You can also download signed qualification reports to submit to the AWS Partner Network to get your device qualified for the [AWS Partner Device Catalog](#) without the need to send your device in and wait for it to be tested.

Note

Device Advisor is supported in us-east-1, us-west-2, ap-northeast-1, and eu-west-1 regions.
Device Advisor supports MQTT with X509 client certificates.

This chapter contains the following sections:

- [Setting up \(p. 948\)](#)
- [Getting started with Device Advisor in the console \(p. 952\)](#)
- [Device Advisor workflow \(p. 958\)](#)
- [Device Advisor detailed console workflow \(p. 962\)](#)
- [Device Advisor test cases \(p. 971\)](#)

Any device that has been built to connect to AWS IoT Core can take advantage of Device Advisor. You can access Device Advisor from the [AWS IoT console](#), or by using the AWS CLI or SDK. When you're ready to test your device, register it with AWS IoT Core and configure the device software with the Device Advisor endpoint. Then choose the prebuilt tests, configure them, run the tests on your device, and get the test results along with detailed logs or a qualification report.

Device Advisor is a test endpoint in the AWS cloud. You can test your devices by configuring them to connect to the test endpoint provided by the Device Advisor. After a device is configured to connect to the test endpoint, you can visit the Device Advisor's console or use the AWS SDK to choose the tests you want to run on your devices. Device Advisor then manages the full lifecycle of a test, including the provisioning of resources, scheduling of the test process, managing the state machine, recording the device behavior, logging the results, and providing the final results in form of a test report.

Setting up

Before you use Device Advisor for the first time, complete the following tasks.

Prerequisites

- [Create an IoT thing \(p. 948\)](#)
- [Create an IAM role to be used as your device role \(p. 949\)](#)
- [Create a custom-managed policy for an IAM user to use Device Advisor \(p. 950\)](#)
- [Create an IAM user to use Device Advisor \(p. 950\)](#)
- [Configure your device \(p. 951\)](#)

Create an IoT thing

First you will need to create a thing and attach a certificate to the thing. Use the following tutorial to create a thing: [Create a thing object](#).

Create an IAM role to be used as your device role

Note

You can quickly create the device role using the Device Advisor console. See [Getting started with the Device Advisor in the console](#) for the steps to set up your device role using the Device Advisor console.

1. Go to the [AWS IAM console](#) and log in to the account you use for Device Advisor testing.
2. In the left navigation pane, chose **Policies**.
3. Choose **Create policy**.
4. Under **Create policy**, do the following:
 1. For **Service**, choose **IoT**.
 2. Under **Actions**, either select actions based on the policy attached to the IoT thing or certificate created in the previous section (recommended), or search for the following actions in the **Filter** action box and select them.
 - Connect
 - Publish
 - Subscribe
 - Receive
 3. Under **Resources**, or best security practices, we recommend you restrict the **client**, **topic**, and **topicfilter** resources using the following steps:
 - a. Choose **Specify client resource ARN for the Connect action**.
 - i. Choose **Add ARN**.
 - ii. Specify the **region**, **accountId**, and **clientId** in the visual ARN editor, or manually specify the Amazon Resource Names (ARNs) of the IoT topics you want to use to run test cases. The **clientId** is the MQTT **clientId** your device uses to interact with Device Advisor.
 - iii. Choose **Add**.
 - b. Choose **Specify topic resource ARN for the Receive and 1 more action**.
 - i. Choose **Add ARN**.
 - ii. Specify the **region**, **accountId**, and **topic name** in the visual ARN editor or manually specify the ARNs of the IoT topics you want to use to run test cases. The **topic name** is the MQTT **topic** your device use to publish messages to.
 - iii. Choose **Add**.
 - c. Choose **Specify topicfilter resource ARN for the Subscribe action**.
 - i. Choose **Add ARN**.
 - ii. Specify the **region**, **accountId**, and **topic name** in the visual ARN editor or manually specify the ARNs of the IoT topics you want to use to run test cases. The **topic name** is the MQTT **topic** your device uses to subscribe to.
 - iii. Choose **Add**.
 5. Choose **Review policy**.
 6. Under **Review policy**, enter a **Name**.
 7. Choose **Create policy**.
 8. On the left navigation pane, Choose **Roles**.
 9. Choose **Create Role**.
 10. Under **Or select a service to view its use cases**, choose **IoT**.
 11. Under **Select your use case**, choose **IoT**.
 12. Choose **Next: Permissions**.

13. (Optional) Under **Set permissions boundary**, Choose **Use a permissions boundary to control the maximum role permissions**, and then choose the policy you just created.
14. Choose **Next: Tags**.
15. Choose **Next: Review**.
16. Enter a **Role name** and a **Role description**.
17. Choose **Create role**.
18. Navigate to the role you created.
19. In the **Permissions** tab, choose **Attach policies** and then choose the policy you created in Step 4
20. Choose **Attach policy**.
21. Choose **Trust relationships** tab and choose **Edit trust relationship**.
22. Enter this policy:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "AllowAwsIoTCoreDeviceAdvisor",  
            "Effect": "Allow",  
            "Principal": {  
                "Service": "iotdeviceadvisor.amazonaws.com"  
            },  
            "Action": "sts:AssumeRole"  
        }  
    ]  
}
```

23. Choose **Update Trust Policy**.

Create a custom-managed policy for an IAM user to use Device Advisor

1. Navigate to the IAM console at <https://console.aws.amazon.com/iam/>. If prompted, enter your AWS credentials to sign in.
2. In the left navigation pane, choose **Policies**.
3. Choose **Create Policy**, then choose the **JSON** tab.
4. Add the necessary permissions to use Device Advisor. The policy document can be found in the topic [Security best practices](#).
5. Choose **Review Policy**.
6. Enter a **Name** and **Description**.
7. Choose **Create Policy**.

Create an IAM user to use Device Advisor

Note

We recommend that you create an IAM user to use when you run Device Advisor tests. Using an IAM admin user to run Device Advisor tests, while allowed, is not recommended.

1. Navigate to the IAM console at <https://console.aws.amazon.com/iam/>. If prompted, enter your AWS credentials to sign in.

2. In the left navigation pane, Choose **Users**.
3. Choose **Add User**.
4. Enter a **User name**.
5. Select **Programmatic access**.
6. Choose **Next: Permissions**.
7. Choose **Attach existing policies directly**.
8. Enter the name of the custom-managed policy the you created in the search box, and then select the check box for **Policy name**.
9. Choose **Next: Tags**.
10. Choose **Next: Review**.
11. Choose **Create user**.
12. Choose **Close**.

Device Advisor requires access to your AWS resources (things, certificates, and endpoints) on your behalf. Your IAM user must have the necessary permissions. Device Advisor will also publish logs to Amazon CloudWatch if you attach the necessary permissions policy to your IAM user.

Configure your device

Device Advisor uses the server name indication (SNI) TLS extension to apply TLS configurations. Devices must use this extension when connecting and pass a server name that is identical to the Device Advisor test endpoint.

Device Advisor allows the TLS connection when a test is in the Running state and denies the TLS connection before and after each test run. For this reason, we also recommend using the device connect retry mechanism to have a fully automated testing experience with Device Advisor. If you run a test suite with more than one test case (for instance TLS connect, MQTT connect, and MQTT publish) then we recommend that you have a mechanism built for your device to try connecting to our test endpoint every five seconds. You can then run multiple test cases, in sequence, in an automated manner.

Note

To make your device software ready for testing, we recommend that you have an SDK that can connect to AWS IoT Core and update the SDK with the Device Advisor test endpoint provided for your account.

Device Advisor supports two types of endpoints: Account-level endpoints and Device-level endpoints. Choose the endpoint that best fits your use case. To simultaneously run multiple test suites using different devices, use a Device-level endpoint. Run the following command to get the Device-level endpoint:

```
aws iotdeviceadvisor get-endpoint --thing-arn your-thing-arn
```

or

```
aws iotdeviceadvisor get-endpoint --certificate-arn your-certificate-arn
```

To run one test suite at a time, choose an Account-level endpoint. Run the following command to get the Account-level endpoint:

```
aws iotdeviceadvisor get-endpoint
```

Getting started with Device Advisor in the console

This tutorial helps you quickly get started with Device Advisor on the console. Device Advisor offers features such as required tests and signed qualification reports to qualify and list devices in the [AWS Partner Device Catalog](#) as detailed in the [AWS IoT Core qualification program](#).

For more information about using Device Advisor, see [Device Advisor workflow \(p. 958\)](#) and [Device Advisor detailed console workflow \(p. 962\)](#).

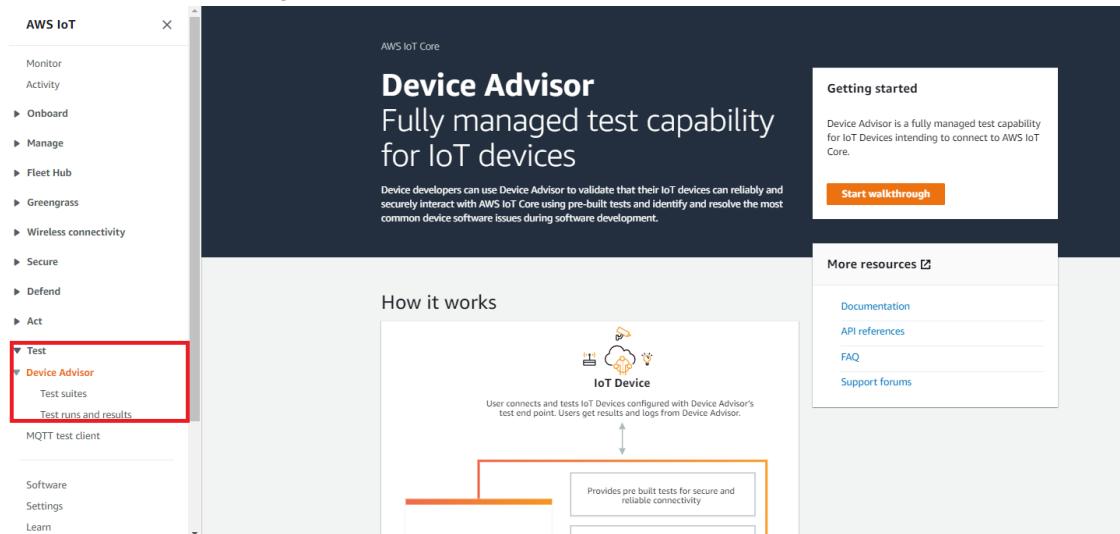
To complete this tutorial, follow the steps outlined in [Setting up \(p. 948\)](#).

Note

Device Advisor is supported in us-east-1, us-west-2, ap-northeast-1, and eu-west-1 regions.

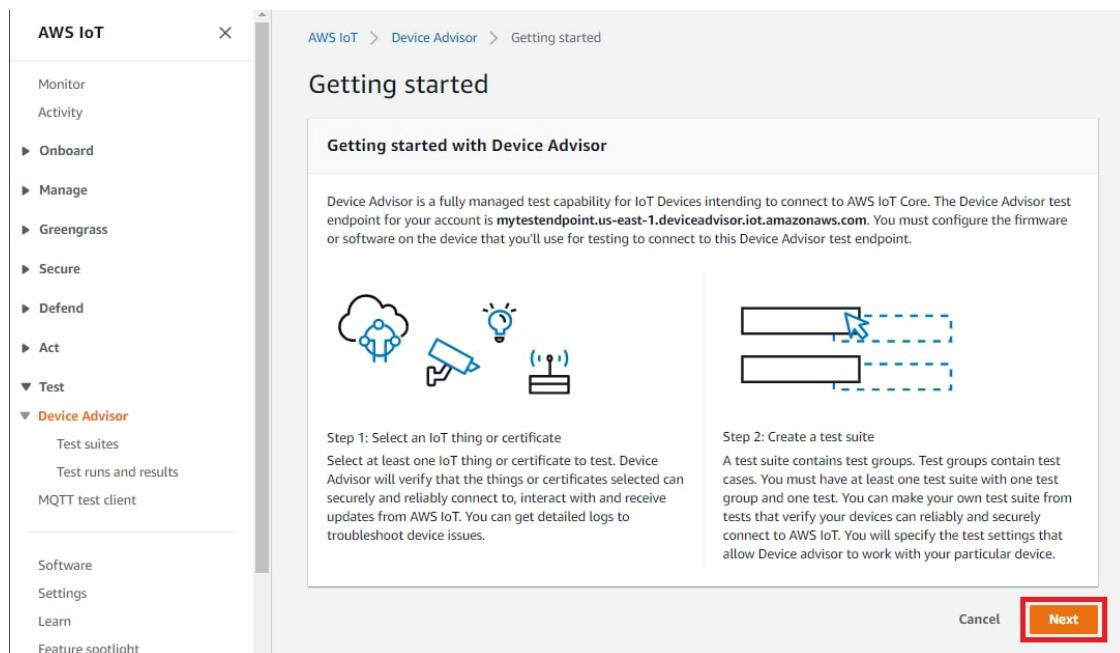
Getting started

1. In the [AWS IoT console](#), in the navigation pane, expand **Test**, and then **Device Advisor**, and then choose **Start walkthrough**.



2. The **Getting started with Device Advisor** page gives an overview of the steps required to create a test suite and run tests against your device. You can also find the Device Advisor test endpoint for your account. You must configure the firmware or software on the device that you'll use for testing to connect to this test endpoint.

To complete this tutorial, you need to [create a thing and certificate](#).



After you've reviewed the information, choose **Next**.

- In **Step 1**, select an AWS IoT thing or certificate to test using Device Advisor. If you don't have any existing things or certificates, see [Setting up](#).

In the **Test endpoint** section, select the endpoint that best fits our use case. If you plan to run multiple test suites simultaneously using the same AWS account, select **Device-level endpoint**. Otherwise, to run one test suite at a time, select **Account-level endpoint**.

Configure your test device with the selected Device Advisor's test endpoint.

Choose **Next**.

Step 1	Step 2	Step 3	Step 4
Select an IoT thing or certificate	Create test suite	Select a device role	Review

Select an IoT thing or certificate

Test device

Select at least one IoT thing or certificate to test using Device Advisor. A list of your things and certificates is shown below. Your test devices must be configured with the correct test endpoint. Refer to [Configure your test device](#) for more details.

Things Certificates

Choose a thing for this test suite. To create a new thing, go to [Things](#)

Choose a certificate for this test suite. To create a new certificate, go to [Certificates](#)

Things (1)

Name: MyThing

Test endpoint

Choose the endpoint that best fits your situation. If you want to simultaneously run multiple test suites then use 'Device-level endpoint'; if you want to run only one test suite at a time then choose the 'Account-level endpoint'.

Account-level endpoint Device-level endpoint

Using this endpoint, you can only run one test suite at a time.

Using this endpoint, you can run multiple test suites simultaneously.

Copy and paste this endpoint to your test device.
tb8dts41394y919yc3t2ub.gamla.us-west-2.advisor.iot.awss.dev

Cancel Next

- In **Step 2**, you create and configure a custom test suite. A custom test suite must have at least one test group, and each test group must have at least one test case. We've added the **MQTT Connect** test case for you to get started.

Choose **Test suite properties**. You must supply the test suite properties when you create your test suite.

The screenshot shows the 'Create test suite' wizard at Step 2. It includes sections for 'Test cases', 'Start', and 'Configure'. A red box highlights the 'Test suite properties' button in the top right corner.

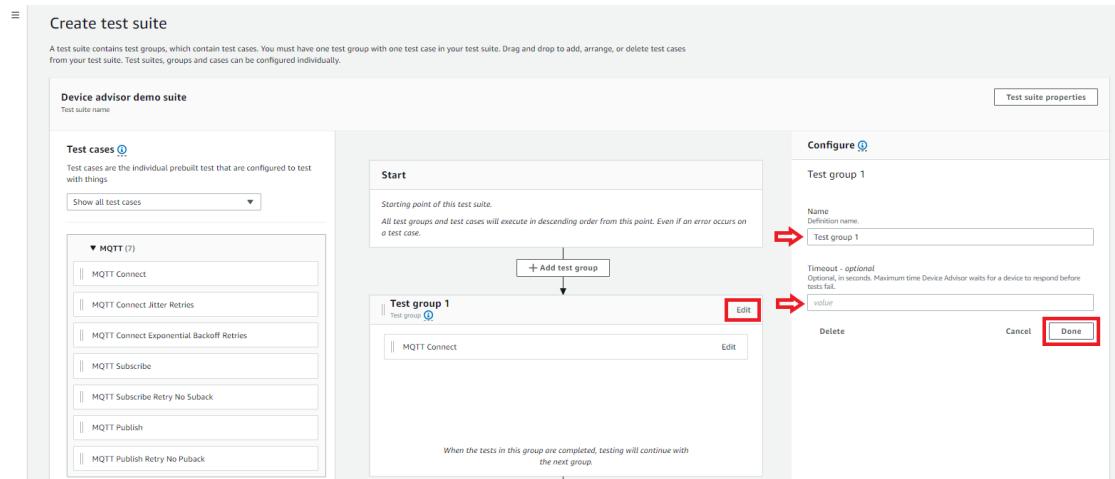
You can configure the following suite-level properties:

- **Test suite name:** Enter a name for your test suite.
- **Timeout (optional):** The timeout in seconds for each test case in the current test suite. If you don't specify a timeout value, the default value is used.
- **Tags (optional):** Add tags to the test suite that you're going to create.

The dialog box is titled 'Test suite properties'. It contains fields for 'Test suite name' (set to 'Device Advisor demo suite'), 'Timeout - optional' (set to '300'), and a 'Tags' section with a 'Add new tag' button. At the bottom are 'Cancel' and 'Update properties' buttons.

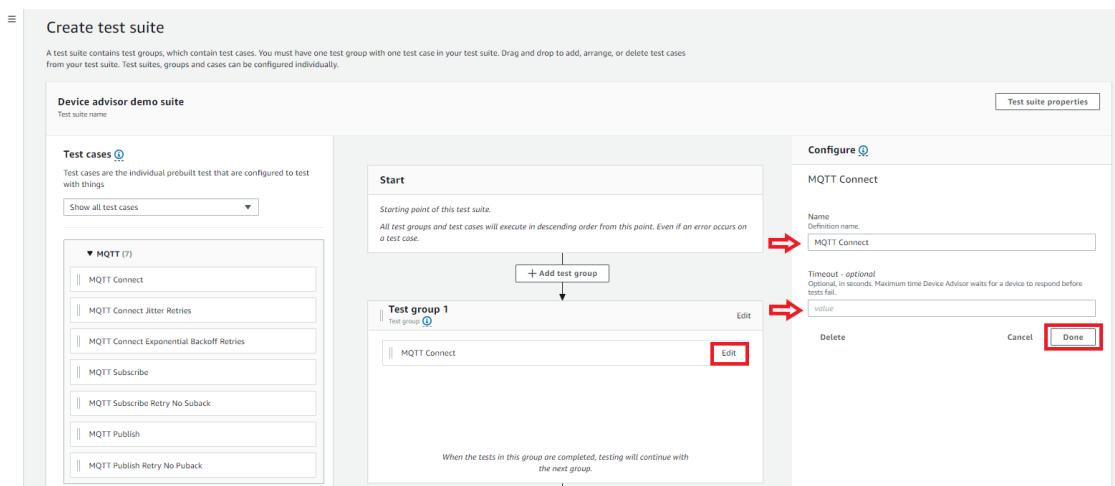
When you've finished, choose **Update properties**.

5. (Optional) You can update the test suite group configuration by choosing **Edit** next to the test group name.
 - **Name:** Enter a custom name for the test suite group.
 - **Timeout (optional):** The timeout in seconds for each test case in the current test suite. If you don't specify a timeout value, the default value is used.



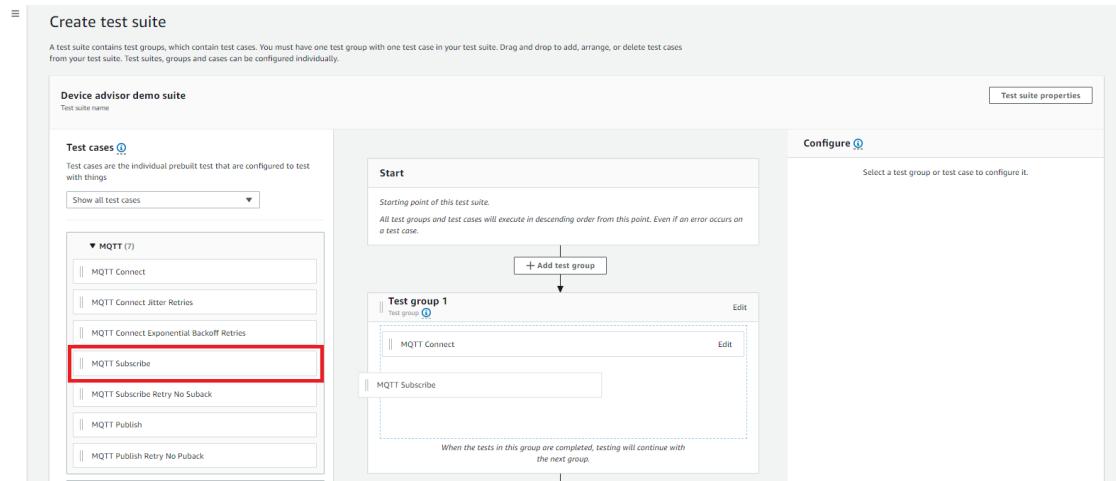
Choose Done.

6. (Optional) You can update the test case configuration by choosing **Edit** next to the test case name.
 - **Name:** Enter a custom name for the test suite group.
 - **Timeout (optional):** The timeout in seconds for the selected test case. If you don't specify a timeout value, the default value is used.



Choose Done.

7. (Optional) To add more test groups to the test suite, choose **Add test group** and follow the instructions in Step 5..
8. (Optional) To add more test cases, drag the test cases displayed under **Test cases** into any of your test groups.



9. Test groups and test cases can be reordered by selecting and dragging the listed test cases. Device Advisor runs tests in the order in which your test cases are listed.

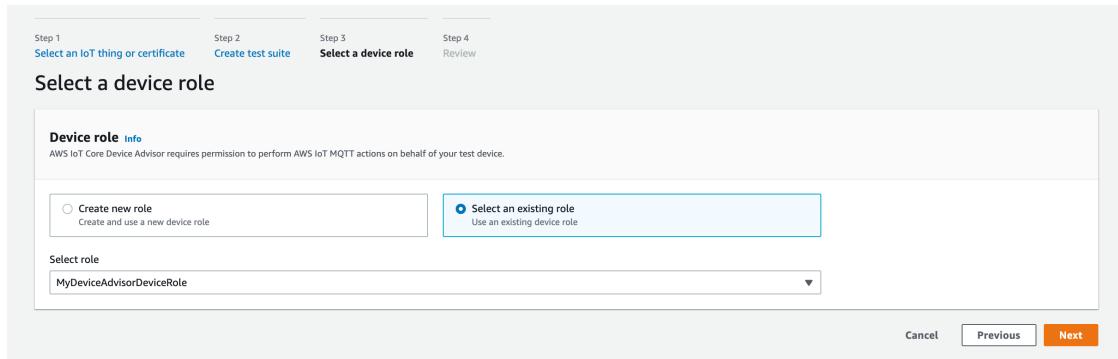
After you've configured your test suite, choose **Next**.

10. In **Step 3** you can configure a device role which Device Advisor will use to perform AWS IoT MQTT actions on behalf of your test device. If you selected the **MQTT Connect** test case only, the **Connect** action will be selected automatically since that permission is required on device role to run this test suite. If you selected other test cases, the corresponding actions will be selected.

Provide the resource values for each of the selected actions. For example, for the **Connect** action, provide the client id with which your device will connect to the Device Advisor endpoint. You can provide multiple values by using commas to separate the values, and you can provide prefix values using a wildcard (*) character. For example, to provide permission to publish on any topic beginning with **MyTopic**, you can provide “**MyTopic***” as the resource value.

Action	Resource type	Resource
<input checked="" type="checkbox"/> Connect	ClientId	MyClient1
<input type="checkbox"/> Publish	Topic	Specify topics to publish to, e.g. MyTopic, MyTopic*
<input type="checkbox"/> Subscribe	TopicFilter	Specify topic filters to subscribe to, e.g. MyTopic, MyTopic*
<input type="checkbox"/> Receive	Topic	Specify topics to receive from e.g. MyTopic, MyTopic*

If you have already created a device role previously from [Setting up](#), and you would like to use that role, choose **Select an existing role** and choose your device role under **Select role**.



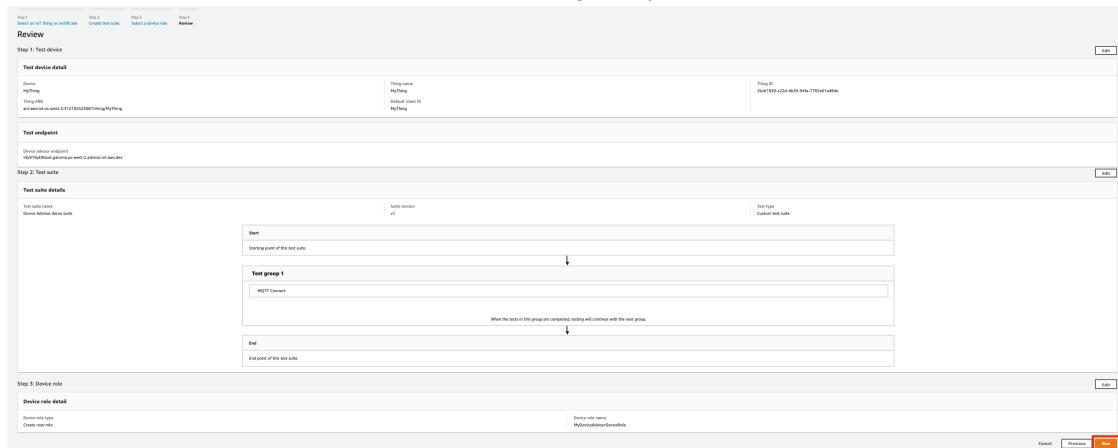
Configure your device role using one of the two provided options and choose **Next**.

11. **Step 4** shows an overview of the selected test device, test endpoint, test suite, and test device role that you've configured. If you want to make changes to a section, choose the **Edit** button above the section you want to edit.

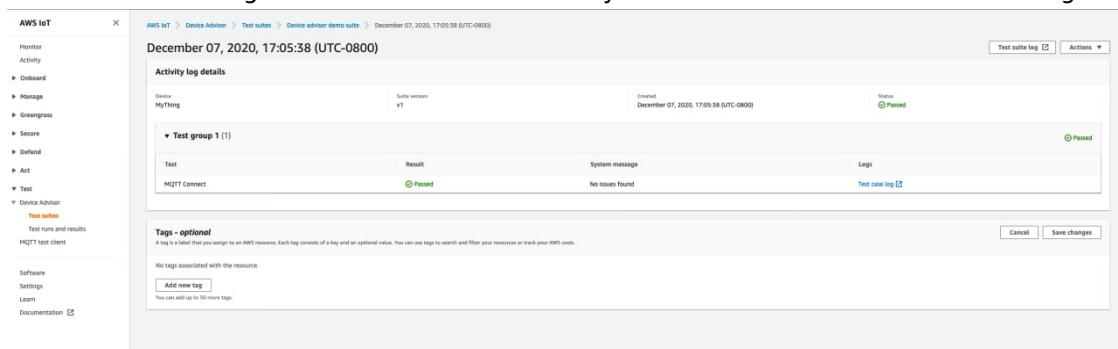
Note

For best results, you can connect your selected test device to the Device Advisor test endpoint before starting the test suite run. We recommend that you have a mechanism built for your device to try connecting to our test endpoint every five seconds for one to two minutes.

To create the test suite and run the selected tests against your device, choose **Run**.



12. In the navigation pane, expand **Test**, **Device Advisor**, and then choose **Test runs and results** to view the run details and logs. Select the test suite run that you started to view the run details and logs.



13. To access the CloudWatch logs for the suite run:

- Choose **Test suite log** to view the CloudWatch logs for the test suite run.
 - Choose **Test case log** for any test case to view test case-specific CloudWatch logs.
14. Based on your test results, [troubleshoot](#) your device until all tests pass.

Device Advisor workflow

This tutorial provides instructions on how to create a custom test suite and run tests against the device you want to test in the console. After the tests are complete, you can view the test results and detailed logs.

Tutorials

- [Prerequisites \(p. 958\)](#)
- [Create a test suite definition \(p. 958\)](#)
- [Get a test suite definition \(p. 960\)](#)
- [Get a test endpoint \(p. 960\)](#)
- [Start a test suite run \(p. 960\)](#)
- [Get a test suite run \(p. 961\)](#)
- [Stop a test suite run \(p. 961\)](#)
- [Get a qualification report for a successful qualification test suite run \(p. 961\)](#)

Prerequisites

To complete this tutorial, complete the steps outlined in [Setting up \(p. 948\)](#).

Create a test suite definition

First, [install an AWS SDK](#).

rootGroup syntax

A root group is a JSON string that specifies which test cases are included in your test suite as well as any necessary configurations for those test cases. Use the root group to structure and order your test suite in any way you like. The hierarchy of a test suite is:

```
test suite # test group(s) # test case(s)
```

A test suite must have at least one test group, and each test group must have at least one test case. Device Advisor runs tests in the order in which you define the test groups and test cases.

Each root group follows this basic structure:

```
{  
    "configuration": { // for all tests in the test suite  
        """: ""  
    }  
    "tests": [  
        {"name": ""  
        "configuration": { // for all sub-groups in this test group  
            """: ""  
        },  
        "tests": [  
            {"name": ""  
            "configuration": { // for all sub-sub-groups in this test case  
                """: ""  
            }  
        }  
    ]  
}
```

```
        "configuration": { // for all test cases in this test group
            "": ""
        },
        "test": {
            "id": ""
            "version": ""
        }
    }
}
}]
```

A block that contains a "name", "configuration", and "tests" is referred to as a "group definition". A block that contains a "name", "configuration", and "test" is referred to as a "test case definition". Each "test" block that contains an "id" and "version" is referred to as a "test case".

For information on how to fill in the "id" and "version" fields for each test case ("test" block), see [Device Advisor test cases \(p. 971\)](#). That section also contains information on the available "configuration" settings.

The following block is an example of a root group configuration that specifies the "MQTT Connect Happy Case" and "MQTT Connect Exponential Backoff Retries" test cases, along with descriptions of the configuration fields.

```

{
    "configuration": {}, // Suite-level configuration
    "tests": [ // Group definitions should be provided here
        {
            "name": "My_MQTT_Connect_Group", // Group definition name
            "configuration": {} // Group definition-level configuration,
            "tests": [ // Test case definitions should be provided here
                {
                    "name": "My_MQTT_Connect_Happy_Case", // Test case definition name
                    "configuration": {
                        "EXECUTION_TIMEOUT": 300 // Test case definition-level
configuration, in seconds
                    },
                    "test": {
                        "id": "MQTT_Connect", // test case id
                        "version": "0.0.0" // test case version
                    }
                },
                {
                    "name": "My_MQTT_Connect_Jitter_Backoff_Retries", // Test case definition name
                    "configuration": {
                        "EXECUTION_TIMEOUT": 600 // Test case definition-level
configuration, in seconds
                    },
                    "test": {
                        "id": "MQTT_Connect_Jitter_Backoff_Retries", // test case id
                        "version": "0.0.0" // test case version
                    }
                }
            ]
        }
    ]
}

```

You must supply the root group configuration when you create your test suite definition. Save the `suiteDefinitionId` that is returned in the response object. This ID is used to retrieve your test suite definition information and to run your test suite.

Here is a Java SDK example:

```
response = iotDeviceAdvisorClient.createSuiteDefinition(
```

```
CreateSuiteDefinitionRequest.builder()
    .suiteDefinitionConfiguration(SuiteDefinitionConfiguration.builder()
        .suiteDefinitionName("your-suite-definition-name")
        .devices(
            DeviceUnderTest.builder()
                .thingArn("your-test-device-thing-arn")
                .certificateArn("your-test-device-certificate-arn")
                .build()
        )
        .rootGroup("your-root-group-configuration")
        .devicePermissionRoleArn("your-device-permission-role-arn")
        .build()
    )
    .build()
)
```

Get a test suite definition

After you start a test suite run, you can check its progress and its results with the `GetSuiteRun` API.

SDK example:

```
// Using the SDK, call the GetSuiteRun API.

response = iotDeviceAdvisorClient.GetSuiteRun(
    GetSuiteRunRequest.builder()
        .suiteDefinitionId("your-suite-definition-id")
        .suiteRunId("your-suite-run-id")
    .build())
```

Get a test endpoint

You can use `GetTestEndpoint` API to get the test endpoint used by your device. While choosing the endpoint, select the endpoint that best fits the situation. To simultaneously run multiple test suites, use `Device-level endpoint` by providing a `thing ARN` or a `certificate ARN`. To run a single test suite, choose the `Account-level endpoint` by providing no arguments.

SDK example:

```
response = iotDeviceAdvisorClient.getEndpoint(GetEndpointRequest.builder()
    .certificateArn("your-test-device-certificate-arn")
    .thingArn("your-test-device-thing-arn")
    .build())
```

Start a test suite run

After you've successfully created a test suite definition and configured your test device to connect to your Device Advisor test endpoint, run your test suite with the `StartSuiteRun` API. Use either `certificateArn` or `thingArn` to run the test suite. If both are configured, the certificate will be used if it belongs to the thing.

For `.parallelRun()`, use `true` if you use Device-level endpoint to run multiple test suites in parallel using one AWS account.

SDK example:

```
response = iotDeviceAdvisorClient.startSuiteRun(StartSuiteRunRequest.builder()
    .suiteDefinitionId("your-suite-definition-id")
    .suiteRunConfiguration(SuiteRunConfiguration.builder()
```

```
.primaryDevice(DeviceUnderTest.builder()
    .certificateArn("your-test-device-certificate-arn")
    .thingArn("your-test-device-thing-arn")
    .build())
.parallelRun(true | false)
.build()
.build()
```

Save the `suiteRunId` that is returned in the response. You will use this to retrieve the results of this test suite run.

Get a test suite run

After you create your test suite definition, you receive the `suiteDefinitionId` in the response object of the `CreateSuiteDefinition` API.

You may see that there are new `id` fields within each of the group and test case definitions in the root group that is returned. This is expected; you can use these IDs to run a subset of your test suite definition.

Java SDK example:

```
response = iotDeviceAdvisorClient.GetSuiteDefinition(
    GetSuiteDefinitionRequest.builder()
        .suiteDefinitionId("your-suite-definition-id")
        .build()
)
```

Stop a test suite run

To stop a test suite run that is still in progress, you can call the `StopSuiteRun` API. After you call the `StopSuiteRun` API, the service will start the cleanup process. While the service is running the cleanup process, the test suite run status is updated to `Stopping`. The cleanup process will take several minutes, and once the process is complete, the test suite run status is updated to `Stopped`. After a test run has completely stopped you will be able to start another test suite run. You can periodically check the suite run status using the `GetSuiteRun` API as shown in the previous section.

SDK example:

```
// Using the SDK, call the StopSuiteRun API.

response = iotDeviceAdvisorClient.StopSuiteRun(
    StopSuiteRun.builder()
        .suiteDefinitionId("your-suite-definition-id")
        .suiteRunId("your-suite-run-id")
    .build()
```

Get a qualification report for a successful qualification test suite run

If you run a qualification test suite that completes, you can retrieve a qualification report by using the `GetSuiteRunReport` API. You can use this qualification report to qualify your device with the AWS IoT Core qualification program. To determine whether your test suite is a qualification test suite, check whether the `intendedForQualification` parameter is set to `true`. After you call the `GetSuiteRunReport` API, the download URL returned is available for you to download for 90 seconds. If more than 90 seconds elapse from the previous time you called the `GetSuiteRunReport` API, call the API again to retrieve a valid URL.

SDK example:

```
// Using the SDK, call the getSuiteRunReport API.  
  
response = iotDeviceAdvisorClient.getSuiteRunReport(  
    GetSuiteRunReportRequest.builder()  
        .suiteDefinitionId("your-suite-definition-id")  
        .suiteRunId("your-suite-run-id")  
        .build()  
)
```

Device Advisor detailed console workflow

In this tutorial, you'll create a custom test suite and run tests against the device you want to test in the console. After the tests are complete, you can view the test results and detailed logs.

Tutorials

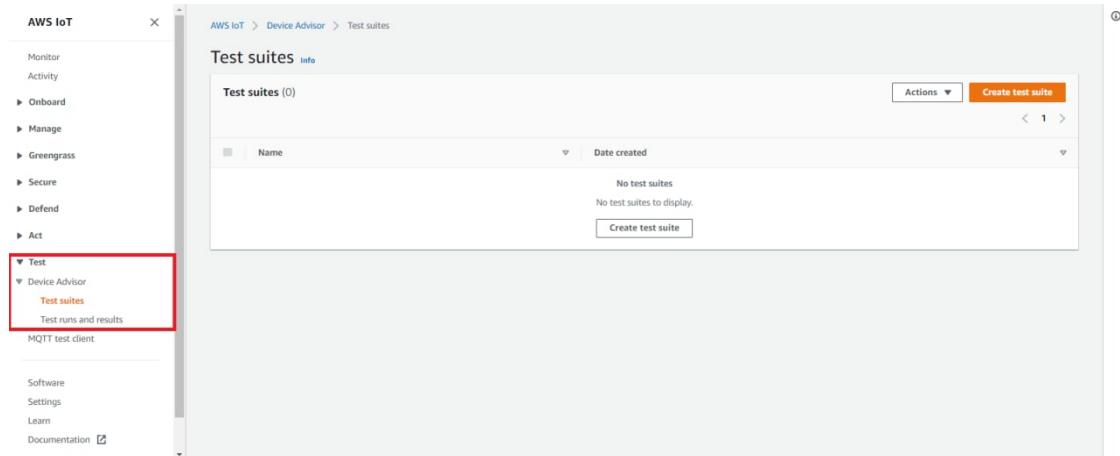
- [Prerequisites \(p. 962\)](#)
- [Create a test suite definition \(p. 962\)](#)
- [Start a test suite run \(p. 966\)](#)
- [Stop a test suite run \(optional\) \(p. 968\)](#)
- [View test suite run details and logs \(p. 969\)](#)
- [Download an AWS IoT qualification report \(p. 970\)](#)

Prerequisites

To complete this tutorial, you need to [create a thing and certificate](#).

Create a test suite definition

1. In the [AWS IoT console](#), in the navigation pane, expand **Test, Device Advisor** and then choose **Test suites**.



Choose **Create Test Suite**.

2. Select either **Use the AWS Qualification test suite** or **Create a new test suite**.

The screenshot shows the first step of a four-step wizard:

- Step 1 Choose test suite type**
- Step 2 Configure test suite**
- Step 3 Select a device role**
- Step 4 Review**

Choose test suite type

Prerequisites

Choose suite type

Choose "Create a new test suite" to debug your device's software. Your test suite must specify one or more prebuilt tests. After debugging your device, choose "Use the AWS IoT Core Qualification test suite" if you want to qualify your device for the AWS Partner Device Catalog per the AWS IoT Core Qualification Program.

Use the AWS IoT Core Qualification test suite
Qualify a device for the AWS Device Partner Catalog.

Create a new test suite
Troubleshoot and debug your device.

Cancel **Next**

Select **Use the AWS Qualification test suite** to qualify and list your device to the AWS Partner Device Catalog. By choosing this option, test cases required for qualification of your device to the AWS IoT Core qualification program are pre-selected. Test groups and test cases can't be added or removed. You will still need to configure the test suite properties.

Select **Create a new test suite** to create and configure a custom test suite. We recommend starting with this option for initial testing and troubleshooting. A custom test suite must have at least one test group, and each test group must have at least one test case. For the purpose of this tutorial, we'll select this option and choose **Next**.

The screenshot shows the second step of the wizard:

- Step 1 Choose test suite type**
- Step 2 Configure test suite**
- Step 3 Select a device role**
- Step 4 Review**

Configure test suite

A test suite contains test groups, which contain test cases. You must have one test group with one test case in your test suite. Drag and drop to add, arrange, or delete test cases from your test suite. Test suites, groups and cases can be configured individually.

Test suite November 14, 2021, 23:05:33 (UTC-0500) **Test suite properties**

Test cases		Start	Configure
<p>Test cases are the individual prebuilt test that are configured to test with things.</p> <p>Show all test cases</p> <p>MQTT (10)</p> <ul style="list-style-type: none"> MQTT Connect MQTT Connect Jitter Retries MQTT Connect Exponential Backoff Retries MQTT Reconnect Backoff Retries On Server Disconnect MQTT Subscribe MQTT Subscribe Retry No Suback 		<p>Starting point of this test suite.</p> <p>All test groups and test cases will execute in descending order from this point. Even if an error occurs on a test case.</p> <p>Start</p> <p>+ Add test group</p>	<p>Select a test group or test case to configure it.</p>
		<p>Test group 1</p> <p>Test group</p> <p>No test cases have been added to this test group.</p>	

3. Choose **Test suite properties**. You must create the test suite properties when you create your test suite.

The screenshot shows the 'Configure test suite' step of the AWS IoT Device Advisor setup. It displays a tree view of test cases under a specific test suite. A red box highlights the 'Test suite properties' button in the top right corner of the configuration panel.

Under **Test suite properties**, fill out the following:

- **Test suite name:** You can create the suite with a custom name.
- **Timeout (optional):** The timeout in seconds for each test case in the current test suite. If you don't specify a timeout value, the default value is used.
- **Tags (optional):** Add tags to the test suite.

When you've finished, choose **Update properties**.

4. To modify the group level configuration, under **Test group 1**, choose **Edit**. Then, enter a **Name** to give the group a custom name.

Optionally, you can also enter a **Timeout** value in seconds under the selected test group. If you don't specify a timeout value, the default value is used.

The screenshot shows the 'Configure test suite' step of the AWS IoT Device Advisor setup. It displays a tree view of test cases under a specific test suite. A red box highlights the 'Edit' button for 'Test group 1'. A second red box highlights the 'Name' field in the 'Configure' dialog for 'Test group 1'. A third red box highlights the 'value' field for 'Timeout - optional'.

Choose **Done**.

5. Drag one of the available test cases from **Test cases** into the test group.

6. To modify the test case level configuration for the test case that you added to your test group, choose **Edit**. Then, enter a **Name** to give the group a custom name.

Optionally, you can also enter a **Timeout** value in seconds under the selected test group. If you don't specify a timeout value, the default value is used.

Choose Done.

Note

To add more test groups to the test suite, choose **Add test group**. Follow the preceding steps to create and configure more test groups, or to add more test cases to one or more test groups. Test groups and test cases can be reordered by choosing and dragging a test case to the desired position. Device Advisor runs tests in the order in which you define the test groups and test cases.

7. Choose **Next**.
8. In **Step 3**, configure a device role which Device Advisor will use to perform AWS IoT MQTT actions on behalf of your test device.

If you selected **MQTT Connect** test case only in **Step 2**, the **Connect** action will be checked automatically since that permission is required on device role to run this test suite. If you selected other test cases, the corresponding required actions will be checked. Ensure that the resource values values for each of the actions is provided. For example, for the **Connect** action, provide the client id that your device will be connecting to the Device Advisor endpoint with. You can provide multiple values by using commas to separate the values, and you can provide prefix values using a wildcard (*) character as well. For example, to provide permission to publish on any topic beginning with **MyTopic**, you can provide "MyTopic*" as the resource value.

If you already created a device role previously and would like to use that role, select **Select an existing role** and choose your device role under **Select role**.

Configure your device role using one of the two provided options and choose **Next**.

9. In **Step 4**, make sure the configuration provided in each of the steps is accurate. To edit configuration provided for a particular step, choose **Edit** for the corresponding step.

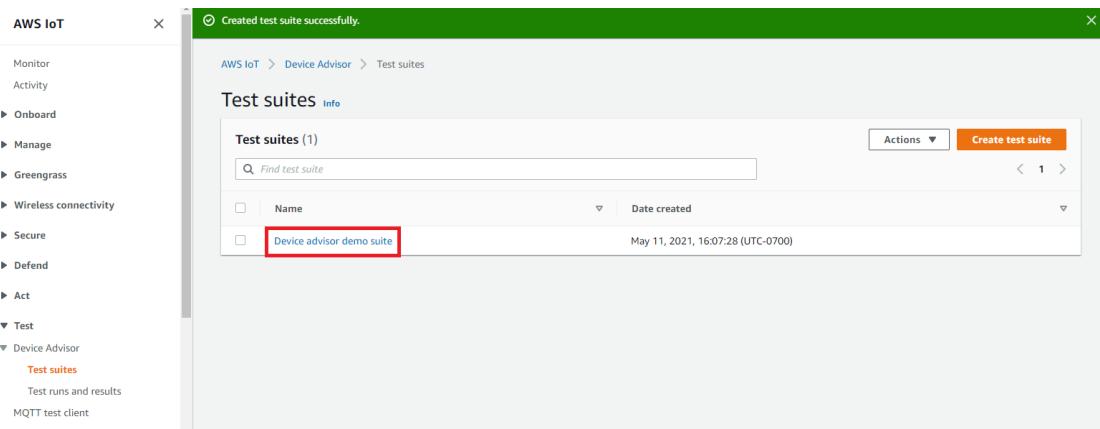
After you verify the configuration, choose **Create test suite**.

The test suite should be created successfully and you'll be redirected to the **Test suites** page where you can view all the test suite that have been created.

If the test suite creation failed, make sure the test suite, test groups, test cases, and device role have been configured according to the previous instructions.

Start a test suite run

1. In the [AWS IoT console](#), in the navigation pane, expand **Test, Device Advisor**, and then choose **Test suites**.
2. Choose the test suite for which you'd like to view the test suite details.



The test suite detail page displays all of the information related to the test suite.

3. Choose Actions, then Run test suite.

This screenshot shows the "Device Advisor demo suite" detail page. The "Actions" menu on the right has "Run test suite" highlighted with a red box. Other options in the menu include "Edit" and "Delete". The page includes sections for "Test suite details" (with suite version v1, created on November 05, 2021, and test type Custom test suite), "Activity Log" (empty), and "Test suite summary" (also empty). A "Start" button is at the bottom.

4. Under Run configuration, you'll need to select an AWS IoT thing or certificate to test using Device Advisor. If you don't have any existing things or certificates, first [create AWS IoT Core resources \(p. 948\)](#).

In **Test endpoint** section, select the endpoint that best fits your case. If you plan to run multiple test suites simultaneously using the same AWS account in the future, select **Device-level endpoint**. Otherwise, if you plan to only run one test suite at a time, select **Account-level endpoint**.

Configure your test device with the selected Device Advisor's test endpoint.

After you select a thing or certificate and choose a Device Advisor endpoint, choose **Run test**.

- Choose **Go to results** on the top banner for viewing the test run details.

Stop a test suite run (optional)

- In the **AWS IoT console**, in the navigation pane, expand **Test**, **Device Advisor**, and then choose **Test runs and results**.
- Choose the test suite in progress that you want to stop.

3. Choose Actions, then Stop test suite.

The screenshot shows the AWS IoT Device Advisor Test suites interface. On the left is a navigation pane with options like Monitor, Activity, Onboard, Manage, Test, and Device Advisor. Under Device Advisor, 'Test suites' is selected. The main area displays a test suite named 'Device advisor demo suite' created on May 11, 2021, at 16:15:43 (UTC-0700). The status is 'In Progress'. Below the suite name, there's an 'Activity log details' section showing a single test group 'Test group 1 (1)'. The first test, 'MQTT Connect', is also in progress. At the top right of the main area, there are buttons for 'Test suite log' and 'Actions'. The 'Actions' button has a red border around it, and its dropdown menu includes 'Run test suite' and 'Stop test suite', with 'Stop test suite' being highlighted in red.

4. The cleanup process will take several minutes to complete. While the cleanup process runs, the test run status will be STOPPING. Wait for the cleanup process to complete and for the test suite status to change to the STOPPED status before starting a new suite run.

This screenshot shows the same test suite after it has been stopped. The status is now 'Stopped'. The 'Activity log details' section shows the same test group and test, but the result is now 'Stopped'. The 'Logs' column contains a link labeled 'Test case log'. The rest of the interface is identical to the previous screenshot, with the 'Actions' button still having a red border.

View test suite run details and logs

1. In the [AWS IoT console](#), in the navigation pane, expand **Test**, **Device Advisor** and then choose **Test runs and results**.

This page displays:

- Number of IoT things
- Number of IoT certificates
- Number of test suites currently running
- All the test suite runs that have been created

2. Choose the test suite for which you'd like to view the run details and logs.

Name	Timestamp	Test suite version	Status	Passed	Failed	Duration
Device Advisor demo suite	December 07, 2020, 11:16:46 (UTC-0800)	v1	In Progress	-	-	-

The run summary page displays the status of the current test suite run. This page automatically refreshes every 10 seconds. We recommend that you have a mechanism built for your device to try connecting to our test endpoint every five seconds for one to two minutes. Then you can run multiple test cases in sequence in an automated manner.

3. To access the CloudWatch logs for the test suite run, choose **Test suite log**.

To access CloudWatch logs for any test case, choose **Test case log**.

4. Based on your test results, **troubleshoot** your device until all tests pass.

Download an AWS IoT qualification report

If you chose the **Use the AWS IoT Qualification test suite** option while creating a test suite and were able to run a qualification test suite, you can download a qualification report by choosing **Download qualification report** in the test run summary page.

The screenshot shows the AWS IoT Device Advisor Test Suite interface. The left sidebar includes options like Monitor, Activity, Dashboard, Manage, Greengrass, Secure, Define, Act, Test, Device Advisor, and Test suites. The main content area displays a qualification report for the 'AWS IoT Core Qualification demo suite' on December 07, 2020, at 23:33:16 (UTC-0800). The report details a 'Qualification Program' with a status of 'Passed'. It lists several test cases: MQTT Connect, MQTT Subscribe, MQTT Publish, TLS Connect, TLS Unsecure Server Cert, and TLS Incorrect Subject Name Server Cert, all of which passed. A 'Tags - optional' section is present, and there are 'Cancel' and 'Save changes' buttons at the bottom.

Device Advisor test cases

Device Advisor provides prebuilt tests in five categories.

- **TLS**
- **Permissions and policies**
- **MQTT**
- **Shadow**
- **Job Execution**

Note

Your device needs to pass the following tests to qualify as per [AWS Device Qualification Program](#)

- **TLS Incorrect Subject Name Server Cert** ("Incorrect Subject Common Name (CN) / Subject Alternative Name (SAN)")
- **TLS Unsecure Server Cert** ("Not Signed By Recognized CA")
- **TLS Connect** ("TLS Connect")
- **MQTT Connect** ("Device send CONNECT to AWS IoT Core (Happy case)")
- **MQTT Subscribe** ("Can Subscribe (Happy Case)")
- **MQTT Publish** ("QoS0 (Happy Case)")

TLS

You use these tests to determine if the transport layer security protocol (TLS) between your devices and AWS IoT is secure.

Cipher suites

"TLS Connect"

Validates if the device under test can complete TLS handshake to AWS IoT. This test doesn't validate the MQTT implementation of the client device.

API test case definition:

Note

EXECUTION_TIMEOUT has a default value of 5 minutes. For best results, we recommend a timeout value of 2 minutes.

```
"tests": [
  {
    "name": "my_tls_connect_test",
    "configuration": {
      // optional:
      "EXECUTION_TIMEOUT": "300", //in seconds
    },
    "test": {
      "id": "TLS_Connect",
      "version": "0.0.0"
    }
  }
]
```

Example Test case outputs:

- **Pass** — The device under test completed TLS handshake with AWS IoT.
- **Pass with warnings** — The device under test completed TLS handshake with AWS IoT, but there were TLS warning messages from the device or AWS IoT.
- **Fail** — The device under test failed to complete TLS handshake with AWS IoT due to handshake error.

"TLS Device Support for AWS IoT recommended Cipher Suites"

Validates that the cipher suites in the TLS Client Hello message from the device under test contains [AWS IoT recommended cipher suites \(p. 362\)](#). It provides additional insights into cipher suites supported by the device.

API test case definition:

Note

EXECUTION_TIMEOUT has a default value of 5 minutes. We recommend a timeout value of 2 minutes.

```
"tests": [
  {
    "name": "my_tls_support_awst_iot_cipher_suites_test",
    "configuration": {
      // optional:
      "EXECUTION_TIMEOUT": "300", // in seconds
    },
    "test": {
      "id": "TLS_Support_AWS_IoT_Cipher_Suites",
      "version": "0.0.0"
    }
  }
]
```

Example Test case outputs:

- **Pass** — The device under test cipher suites contain at least one AWS IoT recommended cipher suite and don't contain any unsupported cipher suites.
- **Pass with warnings** — The device cipher suites contain at least one AWS IoT cipher suite but 1) don't contain any of the recommended cipher suites, or 2) contain cipher suites not supported by AWS IoT. We suggest verifying that unsupported cipher suites are safe.

- **Fail** — The device under test cipher suites don't contain any of the AWS IoT supported cipher suites.

Bad server certificate

"Not Signed By Recognized CA"

Validates that the device under test closes the connection if it's presented with a server certificate that doesn't have a valid signature from the ATS CA. A device should only connect to an endpoint that presents a valid certificate.

API test case definition:

Note

EXECUTION_TIMEOUT has a default value of 5 minutes. We recommend a timeout value of 2 minutes.

```
"tests": [
  {
    "name": "my_tls_unsecure_server_cert_test",
    "configuration": {
      // optional:
      "EXECUTION_TIMEOUT": "300",  //in seconds
    },
    "test": {
      "id": "TLS_Unsecure_Server_Cert",
      "version": "0.0.0"
    }
  }
]
```

Example Test case outputs:

- **Pass** — The device under test closed the connection.
- **Fail** — The device under test completed TLS handshake with AWS IoT.

"Incorrect Subject Common Name (CN) / Subject Alternative Name (SAN)"

Validates that the device under test closes the connection if it's presented with a server certificate for a domain name that is different than the one requested.

API test case definition:

Note

EXECUTION_TIMEOUT has a default value of 5 minutes. We recommend a timeout value of 2 minutes.

```
"tests": [
  {
    "name": "my_tls_incorrect_subject_name_cert_test",
    "configuration": {
      // optional:
      "EXECUTION_TIMEOUT": "300",  // in seconds
    },
    "test": {
      "id": "TLS_Incorrect_Subject_Name_Server_Cert",
      "version": "0.0.0"
    }
  }
]
```

Example Test case outputs:

- **Pass** — The device under test closed the connection.
- **Fail** — The device under test completed TLS handshake with AWS IoT.

Permissions and policies

You can use the following tests to determine if the policies attached to your devices' certificates follow standard best practices.

"Device certificate attached policies don't contain wildcards"

Validates if the permission policies associated with a device follow best practices and do not grant the device more permissions than needed.

API test case definition:

Note

EXECUTION_TIMEOUT has a default value of 1 minute. We recommend setting a timeout of at least 30 seconds.

```
"tests": [
  {
    "name": "my_security_device_policies",
    "configuration": {
      // optional:
      "EXECUTION_TIMEOUT": "60"      // in seconds
    },
    "test": {
      "id": "Security_Device_Policies",
      "version": "0.0.0"
    }
  }
]
```

MQTT

CONNECT, DISCONNECT, and RECONNECT

"Device send CONNECT to AWS IoT Core (Happy case)"

Validates that the device under test sends a CONNECT request.

API test case definition:

Note

EXECUTION_TIMEOUT has a default value of 5 minutes. We recommend a timeout value of 2 minutes.

```
"tests": [
  {
    "name": "my_mqtt_connect_test",
    "configuration": {
      // optional:
      "EXECUTION_TIMEOUT": "300",    // in seconds
    },
    "test": {
      "id": "MQTT_Connect",
      "version": "0.0.0"
    }
}
```

```
        }
    ]
```

"Device can return PUBACK to an arbitrary topic for QoS1"

This test case will check if device (client) can return a PUBACK message if it received a publish message from the broker after subscribing to a topic with QoS1.

API test case definition:

Note

EXECUTION_TIMEOUT has a default value of 5 minutes. We recommend a timeout value of 2 minutes.

```
"tests": [
  {
    "name": "my_mqtt_client_puback_qos1",
    "configuration": {
      // optional:
      "TRIGGER_TOPIC": "myTopic",
      "EXECUTION_TIMEOUT": "300", // in seconds
      "PAYLOAD_FOR_PUBLISH_VALIDATION": "custom payload"
    },
    "test": {
      "id": "MQTT_Client_Puback_Qos1",
      "version": "0.0.0"
    }
  }
]
```

"Device connect retries with jitter backoff - No CONNACK response"

Validates that the device under test uses the proper jitter backoff when reconnecting with the broker for at least five times. The broker logs the timestamp of the device under test's CONNECT request, performs packet validation, pauses without sending a CONNACK to the device under test, and waits for the device under test to resend the request. The sixth connection attempt is allowed to pass through and CONNACK is allowed to flow back to the device under test.

The preceding process is performed again. In total, this test case requires the device to connect at least 12 times in total. The collected timestamps are used to validate that jitter backoff is used by the device under test. If the device under test has a strictly exponential backoff delay, this test case will pass with warnings.

We recommend implementation of the [Exponential Backoff And Jitter](#) mechanism on the device under test to pass this test case.

API test case definition:

Note

EXECUTION_TIMEOUT has a default value of 5 minutes. We recommend a timeout value of 4 minutes.

```
"tests": [
  {
    "name": "my_mqtt_jitter_backoff_retries_test",
    "configuration": {
      // optional:
      "EXECUTION_TIMEOUT": "300", // in seconds
    },
    "test": {
      "id": "MQTT_Connect_Jitter_Backoff_Retries",
```

```

        "version": "0.0.0"
    }
]

```

"Device connect retries with exponential backoff - No CONNACK response"

Validates that the device under test uses the proper exponential backoff when reconnecting with the broker for at least five times. The broker logs the timestamp of the device under test's CONNECT request, performs packet validation, pauses without sending a CONNACK to the client device, and waits for the device under test to resend the request. The collected timestamps are used to validate that an exponential backoff is used by the device under test.

We recommend implementation of the [Exponential Backoff And Jitter](#) mechanism on the device under test to pass this test case.

API test case definition:

Note

EXECUTION_TIMEOUT has a default value of 5 minutes. We recommend a timeout value of 4 minutes.

```

"tests": [
{
    "name": "my_mqtt_exponential_backoff_retries_test",
    "configuration": {
        // optional:
        "EXECUTION_TIMEOUT": "600", // in seconds
    },
    "test": {
        "id": "MQTT_Connect_Exponential_Backoff_Retries",
        "version": "0.0.0"
    }
}
]

```

"Device re-connect with jitter backoff - After server disconnect"

Validates if a device under test uses necessary jitter and backoff while reconnecting after it's been disconnected from the server. Device Advisor disconnects the device from the server for at least five times and observes the device's behavior for MQTT reconnection. Device Advisor logs the timestamp of the CONNECT request for the device under test, performs packet validation, pauses without sending a CONNACK to the client device, and waits for the device under test to resend the request. The collected timestamps are used to validate that the device under test uses jitter and backoff while reconnecting. If the device under test has a strictly exponential backoff or doesn't implement a proper jitter backoff mechanism, this test case will pass with warnings. If the device under test has implemented either a linear backoff or a constant backoff mechanism, the test will fail.

To pass this test case, we recommend implementing the [Exponential Backoff And Jitter](#) mechanism on the device under test.

API test case definition:

Note

EXECUTION_TIMEOUT has a default value of 5 minutes. We recommend a timeout value of 4 minutes.

```

"tests": [
{
    "name": "my_mqtt_reconnect_backoff_retries_on_server_disconnect",
    "configuration": {

```

```

        // optional:
        "EXECUTION_TIMEOUT": "300",    // in seconds
        "RECONNECTION_ATTEMPTS": 5
    },
    "test": {
        "id": "MQTT_Reconnect_Backoff_Retries_On_Server_Disconnect",
        "version": "0.0.0"
    }
}
]

```

The number of reconnection attempts to validate for backoff can be changed by specifying the RECONNECTION_ATTEMPTS. The number must be between five and ten. The default value is five.

Keep-Alive

"Mqtt No Ack PingResp"

This test case validates if the device under test disconnects when it doesn't receive a ping response. As part of this test case, Device Advisor blocks responses sent from AWS IoT Core for publish, subscribe, and ping requests. It also validates if the device under test disconnects the MQTT connection.

API test case definition:

Note

EXECUTION_TIMEOUT has a default value of 5 minutes. We recommend a timeout greater than 1.5 times the keepAliveTime value.

```

"tests": [
{
    "name": "Mqtt No Ack PingResp",
    "configuration": {
        //optional:
        "EXECUTION_TIMEOUT": "306",    // in seconds
    },
    "test": {
        "id": "MQTT_No_Ack_PingResp",
        "version": "0.0.0"
    }
}
]

```

Publish

"QoS0 (Happy Case)"

Validates that the device under test publishes a message with QoS0. You can also validate the topic of the message by specifying this topic value in the test settings.

API test case definition:

Note

EXECUTION_TIMEOUT has a default value of 5 minutes. We recommend a timeout value of 2 minutes.

```

"tests": [
{
    "name": "my_mqtt_publish_test",
    "configuration": {

```

```
// optional:  
"EXECUTION_TIMEOUT": "300", // in seconds  
"TOPIC_FOR_PUBLISH_VALIDATION": "my_TOPIC_FOR_PUBLISH_VALIDATION",  
"PAYLOAD_FOR_PUBLISH_VALIDATION": "my_PAYLOAD_FOR_PUBLISH_VALIDATION",  
,  
"test":{  
    "id":"MQTT_Publish",  
    "version":"0.0.0"  
}  
}  
]
```

"QoS1 publish retry - No PUBACK"

Validates that the device under test republishes a message sent with QoS1, if the broker doesn't send PUBACK. You can also validate the topic of the message by specifying this topic in the test settings. The client device must not disconnect before republishing the message. This test also validates that the republished message has the same packet identifier as the original.

API test case definition:

Note

EXECUTION_TIMEOUT has a default value of 5 minutes. It is recommended for at least 4 minutes.

```
"tests": [  
  {  
    "name": "my_mqtt_publish_retry_test",  
    "configuration": {  
      // optional:  
      "EXECUTION_TIMEOUT": "300", // in seconds  
      "TOPIC_FOR_PUBLISH_VALIDATION": "my_TOPIC_FOR_PUBLISH_VALIDATION",  
      "PAYLOAD_FOR_PUBLISH_VALIDATION": "my_PAYLOAD_FOR_PUBLISH_VALIDATION",  
    },  
    "test": {  
      "id": "MQTT_Publish_Retry_No_Puback",  
      "version": "0.0.0"  
    }  
  }  
]
```

Subscribe

"Can Subscribe (Happy Case)"

Validates that the device under test subscribes to MQTT topics. You can also validate the topic that the device under test subscribes to by specifying this topic in the test settings.

API test case definition:

Note

EXECUTION_TIMEOUT has a default value of 5 minutes. We recommend a timeout value of 2 minutes.

```
"tests": [  
  {  
    "name": "my_mqtt_subscribe_test",  
    "configuration": {  
      // optional:  
      "EXECUTION_TIMEOUT": "300", // in seconds  
      "TOPIC_LIST_FOR_SUBSCRIPTION_VALIDATION_ID":  
      [ "my_TOPIC_FOR_PUBLISH_VALIDATION_a", "my_TOPIC_FOR_PUBLISH_VALIDATION_b" ]  
    }  
  }  
]
```

```

        },
        "test": {
            "id": "MQTT_Subscribe",
            "version": "0.0.0"
        }
    ]
}
]

```

"Subscribe Retry - No SUBACK"

Validates that the device under test retries a failed subscription to MQTT topics. The server then waits and doesn't send a SUBACK. If the client device doesn't retry the subscription, the test fails. The client device must retry the failed subscription with the same packet Id. You can also validate the topic that the device under test subscribes to by specifying this topic in the test settings.

API test case definition:

Note

EXECUTION_TIMEOUT has a default value of 5 minutes. We recommend a timeout value of 4 minutes.

```

"tests": [
{
    "name": "my_mqtt_subscribe_retry_test",
    "configuration": {
        "EXECUTION_TIMEOUT": "300", // in seconds
        // optional:
        "TOPIC_LIST_FOR_SUBSCRIPTION_VALIDATION_ID": [
            "myTOPIC_FOR_PUBLISH_VALIDATION_a", "my_TOPIC_FOR_PUBLISH_VALIDATION_b"
        ],
        "test": {
            "id": "MQTT_Subscribe_Retry_No_Suback",
            "version": "0.0.0"
        }
    }
}
]

```

Shadow

Use these test to verify your devices under test use AWS IoT Device Shadow service correctly. See [AWS IoT Device Shadow service \(p. 591\)](#) for more information. If these test cases are configured in your test suite, then providing a thing is required when starting the suite run.

Publish

"Device publishes state after it connects (Happy case)"

Validates if a device can publish its state after it connects to AWS IoT Core

API test case definition:

Note

EXECUTION_TIMEOUT has a default value of 5 minutes. We recommend a timeout value of 2 minutes.

```

"tests": [
{
    "name": "my_shadow_publish_reported_state",
    "configuration": {
}
]
}
]

```

```
// optional:  
"EXECUTION_TIMEOUT": "300", // in seconds  
"SHADOW_NAME": "SHADOW_NAME",  
"REPORTED_STATE": {  
    "STATE_ATTRIBUTE": "STATE_VALUE"  
},  
"test":{  
    "id":"Shadow_Publish_Reported_State",  
    "version":"0.0.0"  
}  
}  
]
```

The REPORTED_STATE can be provided for additional validation on your device's exact shadow state, after it connects. By default, this test case validates your device publishing state.

If **SHADOW_NAME** is not provided, the test case looks for messages published to topic prefixes of the Unnamed (classic) shadow type by default. Provide a shadow name if your device uses the named shadow type. See [Using shadows in devices](#) for more information.

Update

"Device updates reported state to desired state (Happy case)"

Validates if your device reads all update messages received and synchronizes the device's state to match the desired state properties. Your device should publish its latest reported state after synchronizing. If your device already has an existing shadow before running the test, make sure the desired state configured for the test case and the existing reported state do not already match. You can identify Shadow update messages sent by Device Advisor by looking at the **ClientToken** field in the Shadow document as it will be `DeviceAdvisorShadowTestCaseSetup`.

API test case definition:

Note

`EXECUTION_TIMEOUT` has a default value of 5 minutes. We recommend a timeout value of 2 minutes.

```
"tests": [  
    {  
        "name": "my_shadow_update_reported_state",  
        "configuration": {  
            "DESIRED_STATE": {  
                "STATE_ATTRIBUTE": "STATE_VALUE"  
            },  
            // optional:  
            "EXECUTION_TIMEOUT": "300", // in seconds  
            "SHADOW_NAME": "SHADOW_NAME"  
        },  
        "test":{  
            "id":"Shadow_Update_Reported_State",  
            "version":"0.0.0"  
        }  
    }  
]
```

The `DESIRED_STATE` should have at least one attribute and associated value.

If `SHADOW_NAME` is not provided, then the test case looks for messages published to topic prefixes of the Unnamed (classic) shadow type by default. Provide a shadow name if your device uses the named shadow type. See [Using shadows in devices](#) for more information.

Job Execution

"Device can complete a job execution"

This test case helps you validate if your device is able to receive updates using AWS IoT Jobs, and publish the status of successful updates. For more information on AWS IoT Jobs, see [Jobs](#).

API test case definition:

Note

EXECUTION_TIMEOUT has a default value of 5 minutes. We recommend a timeout value of 2 minutes.

```
{
  "tests": [
    {
      "name": "my_job_execution",
      "configuration": {
        // Document is a JSON formatted string
        "JOB_DOCUMENT": "{\n          \"operation\": \"reboot\",\n          \"files\": {\n            \"fileName\": \"install.py\",\n            \"url\": \"$aws:iot:s3-presigned-url:https://\n            s3.amazonaws.com/bucket-name/key\"\n          }\n        }",
        // Document_SOURCE is an S3 link to the job document
        // Document and document are optional but can't be both null.
        "JOB_DOCUMENT_SOURCE": "https://s3.amazonaws.com/bucket-name/key",
        // optional:
        "EXECUTION_TIMEOUT": "300", // in seconds
        // JobId is used to create test job, if not provided, test case will create
        // a random Id
        "JOB_JOBID": "String",
        // Role Arn is used to presign Url, which will replace the placeholder in
        // Job document
        "JOB_PRESIGN_ROLE_ARN": "String",
        // Presigned Url expiration time, must be 60 - 3600, default value is 3600
        "JOB_PRESIGN_EXPIRES_IN_SEC": "Long"
      },
      "test": {
        "id": "Job_Execution",
        "version": "0.0.0"
      }
    }
  ]
}
```

For more information on creating and using job documents see [job document](#).

Event messages

This section contains information about messages published by AWS IoT when things or jobs are updated or changed. For information about the AWS IoT Events service that allows you to create detectors to monitor your devices for failures or changes in operation, and to trigger actions when they occur, see [AWS IoT Events](#).

How event messages are generated

AWS IoT publishes event messages when certain events occur. For example, events are generated by the registry when things are added, updated, or deleted. Each event causes a single event message to be sent. Event messages are published over MQTT with a JSON payload. The content of the payload depends on the type of event.

Note

Event messages are guaranteed to be published once. It is possible for them to be published more than once. The ordering of event messages is not guaranteed.

Policy for receiving event messages

To receive event messages, your device must use an appropriate policy that allows it to connect to the AWS IoT device gateway and subscribe to MQTT event topics. You must also subscribe to the appropriate topic filters.

The following is an example of the policy required for receiving lifecycle events:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:Subscribe",  
                "iot:Receive"  
            ],  
            "Resource": [  
                "arn:aws:iot:region:account:/$aws/events/*"  
            ]  
        }  
    ]  
}
```

Enable events for AWS IoT

Before subscribers to the reserved topics can receive messages, you must enable event messages from the AWS Management Console or by using the API or CLI.

- To enable event messages, go to the [Settings](#) tab of the AWS IoT console and then, in the **Event-based messages** section, choose **Manage events**. You can specify the events that you want to manage.
- To control which event types are published by using the API or CLI, call the [UpdateEventConfigurations](#) API or use the **update-event-configurations** CLI command. For example:

```
aws iot update-event-configurations --event-configurations "{\"THING\":{\"Enabled\": true}}"
```

Note

All quotation marks ("") are escaped with a backslash (\).

You can get the current event configuration by calling the [DescribeEventConfigurations API](#) or by using the **describe-event-configurations** CLI command. For example:

```
aws iot describe-event-configurations
```

Registry events

The registry can publish event messages when things, thing types, and thing groups are created, updated, or deleted. These events, however, are not available by default. For information about how to turn on these events, see [Enable events for AWS IoT \(p. 982\)](#).

The registry can provide the following event types:

- [Thing events \(p. 983\)](#)
- [Thing type events \(p. 984\)](#)
- [Thing group events \(p. 986\)](#)

Thing events

Thing Created/Updated/Deleted

The registry publishes the following event messages when things are created, updated, or deleted:

- \$aws/events/thing/*thingName*/created
- \$aws/events/thing/*thingName*/updated
- \$aws/events/thing/*thingName*/deleted

The messages contain the following example payload:

```
{
    "eventType" : "THING_EVENT",
    "eventId" : "f5ae9b94-8b8e-4d8e-8c8f-b3266dd89853",
    "timestamp" : 1234567890123,
    "operation" : "CREATED|UPDATED|DELETED",
    "accountId" : "123456789012",
    "thingId" : "b604f69c-aa9a-4d4a-829e-c480e958a0b5",
    "thingName" : "MyThing",
    "versionNumber" : 1,
    "thingTypeName" : null,
    "attributes": {
        "attribute3": "value3",
        "attribute1": "value1",
        "attribute2": "value2"
    }
}
```

The payloads contain the following attributes:

eventType

Set to "THING_EVENT".

eventId

A unique event ID (string).

timestamp

The UNIX timestamp of when the event occurred.

operation

The operation that triggered the event. Valid values are:

- CREATED
- UPDATED
- DELETED

accountId

Your AWS account ID.

thingId

The ID of the thing being created, updated, or deleted.

thingName

The name of the thing being created, updated, or deleted.

versionNumber

The version of the thing being created, updated, or deleted. This value is set to 1 when a thing is created. It is incremented by 1 each time the thing is updated.

thingTypeName

The thing type associated with the thing, if one exists. Otherwise, null.

attributes

A collection of name-value pairs associated with the thing.

Thing type events

Thing type related events:

- [Thing Type Created/Deprecated/Undeleted/Deleted \(p. 984\)](#)
- [Thing Type Associated or Disassociated with a Thing \(p. 985\)](#)

Thing Type Created/Deprecated/Undeleted/Deleted

The registry publishes the following event messages when thing types are created, deprecated, undeleted, or deleted:

- `$aws/events/thingType/thingTypeName/created`
- `$aws/events/thingType/thingTypeName/updated`
- `$aws/events/thingType/thingTypeName/deleted`

The message contains the following example payload:

```
{  
    "eventType" : "THING_TYPE_EVENT",  
}
```

```
"eventId" : "8827376c-4b05-49a3-9b3b-733729df7ed5",
"timestamp" : 1234567890123,
"operation" : "CREATED|UPDATED|DELETED",
"accountId" : "123456789012",
"thingTypeId" : "c530ae83-32aa-4592-94d3-da29879d1aac",
"thingTypeName" : "MyThingType",
"isDeprecated" : false|true,
"deprecationDate" : null,
"searchableAttributes" : [ "attribute1", "attribute2", "attribute3" ],
"description" : "My thing type"
}
```

The payloads contain the following attributes:

eventType

Set to "THING_TYPE_EVENT".

eventId

A unique event ID (string).

timestamp

The UNIX timestamp of when the event occurred.

operation

The operation that triggered the event. Valid values are:

- CREATED
- UPDATED
- DELETED

accountId

Your AWS account ID.

thingTypeId

The ID of the thing type being created, deprecated, or deleted.

thingTypeName

The name of the thing type being created, deprecated, or deleted.

isDeprecated

true if the thing type is deprecated. Otherwise, false.

deprecationDate

The UNIX timestamp for when the thing type was deprecated.

searchableAttributes

A collection of name-value pairs associated with the thing type that can be used for searching.

description

A description of the thing type.

Thing Type Associated or Disassociated with a Thing

The registry publishes the following event messages when a thing type is associated or disassociated with a thing.

- `$aws/events/thingTypeAssociation/thing/thingName/typeName`

The messages contain the following example payload:

```
{  
    "eventId" : "87f8e095-531c-47b3-aab5-5171364d138d",  
    "eventType" : "THING_TYPE_ASSOCIATION_EVENT",  
    "operation" : "CREATED|DELETED",  
    "thingId" : "b604f69c-aa9a-4d4a-829e-c480e958a0b5",  
    "thingName": "myThing",  
    "thingTypeName" : "MyThingType",  
    "timestamp" : 1234567890123,  
}
```

The payloads contain the following attributes:

`eventId`

A unique event ID (string).

`eventType`

Set to "THING_TYPE_ASSOCIATION_EVENT".

`operation`

The operation that triggered the event. Valid values are:

- `CREATED`
- `DELETED`

`thingId`

The ID of the thing whose type association was changed.

`thingName`

The name of the thing whose type association was changed.

`thingTypeName`

The thing type associated with, or no longer associated with, the thing.

`timestamp`

The UNIX timestamp of when the event occurred.

Thing group events

Thing group related events:

- [Thing Group Created/Updated/Deleted \(p. 986\)](#)
- [Thing Added to or Removed from a Thing Group \(p. 988\)](#)
- [Thing Group Added to or Deleted from a Thing Group \(p. 989\)](#)

Thing Group Created/Updated/Deleted

The registry publishes the following event messages when a thing group is created, updated, or deleted.

- `$aws/events/thingGroup/groupName/created`
- `$aws/events/thingGroup/groupName/updated`

- \$aws/events/thingGroup/*groupName*/deleted

The following is an example of an updated payload. Payloads for created and deleted messages are similar.

```
{
  "eventType": "THING_GROUP_EVENT",
  "eventId": "8b9ea8626aea1e42100f3f32b975899",
  "timestamp": 1603995417409,
  "operation": "UPDATED",
  "accountId": "571EXAMPLE833",
  "thingGroupId": "8757eec8-bb37-4cca-a6fa-403b003d139f",
  "thingGroupName": "Tg_level5",
  "versionNumber": 3,
  "parentGroupName": "Tg_level4",
  "parentGroupId": "5fce366a-7875-4c0e-870b-79d8d1dce119",
  "description": "New description for Tg_level5",
  "rootToParentThingGroups": [
    {
      "groupArn": "arn:aws:iot:us-west-2:571EXAMPLE833:thinggroup/TgTopLevel",
      "groupId": "36aa0482-f80d-4e13-9bff-1c0a75c055f6"
    },
    {
      "groupArn": "arn:aws:iot:us-west-2:571EXAMPLE833:thinggroup/Tg_level1",
      "groupId": "bc1643e1-5a85-4eac-b45a-92509cbe2a77"
    },
    {
      "groupArn": "arn:aws:iot:us-west-2:571EXAMPLE833:thinggroup/Tg_level2",
      "groupId": "0476f3d2-9beb-48bb-ae2c-ea8bd6458158"
    },
    {
      "groupArn": "arn:aws:iot:us-west-2:571EXAMPLE833:thinggroup/Tg_level3",
      "groupId": "1d9d4ffe-a6b0-48d6-9de6-2e54d1eae78f"
    },
    {
      "groupArn": "arn:aws:iot:us-west-2:571EXAMPLE833:thinggroup/Tg_level4",
      "groupId": "5fce366a-7875-4c0e-870b-79d8d1dce119"
    }
  ],
  "attributes": {
    "attribute1": "value1",
    "attribute3": "value3",
    "attribute2": "value2"
  },
  "dynamicGroupMappingId": null
}
```

The payloads contain the following attributes:

eventType

Set to "THING_GROUP_EVENT".

eventId

A unique event ID (string).

timestamp

The UNIX timestamp of when the event occurred.

operation

The operation that triggered the event. Valid values are:

- CREATED
- UPDATED
- DELETED

accountId

Your AWS account ID.

thingGroupId

The ID of the thing group being created, updated, or deleted.

thingGroupName

The name of the thing group being created, updated, or deleted.

versionNumber

The version of the thing group. This value is set to 1 when a thing group is created. It is incremented by 1 each time the thing group is updated.

parentGroupName

The name of the parent thing group, if one exists.

parentGroupId

The ID of the parent thing group, if one exists.

description

A description of the thing group.

rootToParentThingGroups

An array of information about the parent thing group. There is one element for each parent thing group, starting from the root thing group and continuing to the thing group's parent. Each entry contains the thing group's groupArn and groupId.

attributes

A collection of name-value pairs associated with the thing group.

Thing Added to or Removed from a Thing Group

The registry publishes the following event messages when a thing is added to or removed from a thing group.

- \$aws/events/thingGroupMembership/thingGroup/*thingGroupName*/thing/*thingName*/added
- \$aws/events/thingGroupMembership/thingGroup/*thingGroupName*/thing/*thingName*/removed

The messages contain the following example payload:

```
{  
    "eventType" : "THING_GROUP_MEMBERSHIP_EVENT",  
    "eventId" : "d684bd5f-6f6e-48e1-950c-766ac7f02fd1",  
    "timestamp" : 1234567890123,  
    "operation" : "ADDED|REMOVED",  
    "accountId" : "123456789012",  
    "groupArn" : "arn:aws:iot:ap-northeast-2:123456789012:thinggroup/MyChildThingGroup",  
    "groupId" : "06838589-373f-4312-b1f2-53f2192291c4",  
    "thingArn" : "arn:aws:iot:ap-northeast-2:123456789012:thing/MyThing",  
}
```

```
        "thingId" : "b604f69c-aa9a-4d4a-829e-c480e958a0b5",
        "membershipId" : "8505ebf8-4d32-4286-80e9-c23a4a16bbd8"
    }
```

The payloads contain the following attributes:

eventType

Set to "THING_GROUP_MEMBERSHIP_EVENT".

eventId

The event ID.

timestamp

The UNIX timestamp for when the event occurred.

operation

ADDED when a thing is added to a thing group. REMOVED when a thing is removed from a thing group.

accountId

Your AWS account ID.

groupArn

The ARN of the thing group.

groupId

The ID of the group.

thingArn

The ARN of the thing that was added or removed from the thing group.

thingId

The ID of the thing that was added or removed from the thing group.

membershipId

An ID that represents the relationship between the thing and the thing group. This value is generated when you add a thing to a thing group.

Thing Group Added to or Deleted from a Thing Group

The registry publishes the following event messages when a thing group is added to or removed from another thing group.

- \$aws/events/thingGroupHierarchy/thingGroup/*parentThingGroupName*/childThingGroup/*childThingGroupName*/added
- \$aws/events/thingGroupHierarchy/thingGroup/*parentThingGroupName*/childThingGroup/*childThingGroupName*/removed

The message contains the following example payload:

```
{
    "eventType" : "THING_GROUP_HIERARCHY_EVENT",
    "eventId" : "264192c7-b573-46ef-ab7b-489fcfd47da41",
    "timestamp" : 1234567890123,
    "operation" : "ADDED|REMOVED",
```

```
"accountId" : "123456789012",
"thingGroupId" : "8f82a106-6b1d-4331-8984-a84db5f6f8cb",
"thingGroupName" : "MyRootThingGroup",
"childGroupId" : "06838589-373f-4312-b1f2-53f2192291c4",
"childGroupName" : "MyChildThingGroup"
}
```

The payloads contain the following attributes:

eventType

Set to "THING_GROUP_HIERARCHY_EVENT".

eventId

The event ID.

timestamp

The UNIX timestamp for when the event occurred.

operation

ADDED when a thing is added to a thing group. REMOVED when a thing is removed from a thing group.

accountId

Your AWS account ID.

thingGroupId

The ID of the parent thing group.

thingGroupName

The name of the parent thing group.

childGroupId

The ID of the child thing group.

childGroupName

The name of the child thing group.

Jobs events

The AWS IoT Jobs service publishes to reserved topics on the MQTT protocol when jobs are pending, completed, or canceled, and when a device reports success or failure when running a job. Devices or management and monitoring applications can track the status of jobs by subscribing to these topics.

How to enable jobs events

Response messages from the AWS IoT Jobs service don't pass through the message broker and they can't be subscribed to by other clients or rules. To subscribe to job activity-related messages, use the `notify` and `notify-next` topics. For information about jobs topics, see [Job topics \(p. 101\)](#).

To be notified of jobs updates, enable these jobs events by using the AWS Management Console, or by using the API or CLI. For more information, see [Enable events for AWS IoT \(p. 982\)](#).

How jobs events work

Because it can take some time to cancel or delete a job, two messages are sent to indicate the start and end of a request. For example, when a cancellation request starts, a message is sent to the `$aws/`

events/job/jobID/cancellation_in_progress topic. When the cancellation request is complete, a message is sent to the \$aws/events/job/jobID/canceled topic.

A similar process occurs for a job deletion request. Management and monitoring applications can subscribe to these topics to keep track of the status of jobs. For more information about publishing and subscribing to MQTT topics, see [the section called "Device communication protocols" \(p. 77\)](#).

Job event types

The following shows the different types of jobs events:

Job Completed/Canceled/Deleted

The AWS IoT Jobs service publishes a message on an MQTT topic when a job is completed, canceled, deleted, or when cancellation or deletion are in progress:

- \$aws/events/job/*jobID*/completed
- \$aws/events/job/*jobID*/canceled
- \$aws/events/job/*jobID*/deleted
- \$aws/events/job/*jobID*/cancellation_in_progress
- \$aws/events/job/*jobID*/deletion_in_progress

The completed message contains the following example payload:

```
{  
  "eventType": "JOB",  
  "eventId": "7364ffd1-8b65-4824-85d5-6c14686c97c6",  
  "timestamp": 1234567890,  
  "operation": "completed",  
  "jobId": "27450507-bf6f-4012-92af-bb8a1c8c4484",  
  "status": "COMPLETED",  
  "targetSelection": "SNAPSHOT|CONTINUOUS",  
  "targets": [  
    "arn:aws:iot:us-east-1:123456789012:thing/a39f6f91-70cf-4bd2-a381-9c66df1a80d0",  
    "arn:aws:iot:us-east-1:123456789012:thinggroup/2fc4c0a4-6e45-4525-  
a238-0fe8d3dd21bb"  
  ],  
  "description": "My Job Description",  
  "completedAt": 1234567890123,  
  "createdAt": 1234567890123,  
  "lastUpdatedAt": 1234567890123,  
  "jobProcessDetails": {  
    "numberOfCanceledThings": 0,  
    "numberOfRejectedThings": 0,  
    "numberOfFailedThings": 0,  
    "numberOfRemovedThings": 0,  
    "numberOfSucceededThings": 3  
  }  
}
```

The canceled message contains the following example payload.

```
{  
  "eventType": "JOB",  
  "eventId": "568d2ade-2e9c-46e6-a115-18afa1286b06",  
  "timestamp": 1234567890,  
  "operation": "canceled",  
  "jobId": "4d2a531a-da2e-47bb-8b9e-ff5adcd53ef0",  
  "status": "CANCELED",  
  "targetSelection": "SNAPSHOT|CONTINUOUS",  
  "targets": [  
  ]  
}
```

```

    "arn:aws:iot:us-east-1:123456789012:thing/Thing0-947b9c0c-ff10-4a80-b4b3-
cd33d0145a0f",
    "arn:aws:iot:us-east-1:123456789012:thinggroup/ThingGroup1-95c644d5-1621-41a6-9aa5-
ad2de581d18f"
],
"description": "My job description",
"createdAt": 1234567890123,
"lastUpdatedAt": 1234567890123
}

```

The deleted message contains the following example payload.

```

{
    "eventType": "JOB",
    "eventId": "568d2ade-2e9c-46e6-a115-18afa1286b06",
    "timestamp": 1234567890,
    "operation": "deleted",
    "jobId": "4d2a531a-da2e-47bb-8b9e-ff5adcd53ef0",
    "status": "DELETED",
    "targetSelection": "SNAPSHOT|CONTINUOUS",
    "targets": [
        "arn:aws:iot:us-east-1:123456789012:thing/Thing0-947b9c0c-ff10-4a80-b4b3-
cd33d0145a0f",
        "arn:aws:iot:us-east-1:123456789012:thinggroup/
ThingGroup1-95c644d5-1621-41a6-9aa5-ad2de581d18f"
    ],
    "description": "My job description",
    "createdAt": 1234567890123,
    "lastUpdatedAt": 1234567890123,
    "comment": "Comment for this operation"
}

```

The cancellation_in_progress message contains the following example payload:

```

{
    "eventType": "JOB",
    "eventId": "568d2ade-2e9c-46e6-a115-18afa1286b06",
    "timestamp": 1234567890,
    "operation": "cancellation_in_progress",
    "jobId": "4d2a531a-da2e-47bb-8b9e-ff5adcd53ef0",
    "status": "CANCELLATION_IN_PROGRESS",
    "targetSelection": "SNAPSHOT|CONTINUOUS",
    "targets": [
        "arn:aws:iot:us-east-1:123456789012:thing/Thing0-947b9c0c-ff10-4a80-b4b3-
cd33d0145a0f",
        "arn:aws:iot:us-east-1:123456789012:thinggroup/
ThingGroup1-95c644d5-1621-41a6-9aa5-ad2de581d18f"
    ],
    "description": "My job description",
    "createdAt": 1234567890123,
    "lastUpdatedAt": 1234567890123,
    "comment": "Comment for this operation"
}

```

The deletion_in_progress message contains the following example payload:

```

{
    "eventType": "JOB",
    "eventId": "568d2ade-2e9c-46e6-a115-18afa1286b06",
    "timestamp": 1234567890,
    "operation": "deletion_in_progress",
    "jobId": "4d2a531a-da2e-47bb-8b9e-ff5adcd53ef0",
    "status": "DELETION_IN_PROGRESS"
}

```

```
        "status": "DELETION_IN_PROGRESS",
        "targetSelection": "SNAPSHOT|CONTINUOUS",
        "targets": [
            "arn:aws:iot:us-east-1:123456789012:thing/Thing0-947b9c0c-ff10-4a80-b4b3-
cd33d0145a0f",
            "arn:aws:iot:us-east-1:123456789012:thinggroup/
ThingGroup1-95c644d5-1621-41a6-9aa5-ad2de581d18f"
        ],
        "description": "My job description",
        "createdAt": 1234567890123,
        "lastUpdatedAt": 1234567890123,
        "comment": "Comment for this operation"
    }
```

Job Execution Terminal Status

The AWS IoT Jobs service publishes a message when a device updates a job execution to terminal status:

- \$aws/events/jobExecution/*jobID*/succeeded
- \$aws/events/jobExecution/*jobID*/failed
- \$aws/events/jobExecution/*jobID*/rejected
- \$aws/events/jobExecution/*jobID*/canceled
- \$aws/events/jobExecution/*jobID*/timed_out
- \$aws/events/jobExecution/*jobID*/removed
- \$aws/events/jobExecution/*jobID*/deleted

The message contains the following example payload:

```
{
    "eventType": "JOB_EXECUTION",
    "eventId": "cca89fa5-8a7f-4ced-8c20-5e653afb3572",
    "timestamp": 1234567890,
    "operation": "succeeded|failed|rejected|canceled|removed|timed_out",
    "jobId": "154b39e5-60b0-48a4-9b73-f6f8dd032d27",
    "thingArn": "arn:aws:iot:us-east-1:123456789012:myThing/6d639fbc-8f85-4a90-924d-
a2867f8366a7",
    "status": "SUCCEEDED|FAILED|REJECTED|CANCELED|REMOVED|TIMED_OUT",
    "statusDetails": {
        "key": "value"
    }
}
```

Lifecycle events

AWS IoT can publish lifecycle events on the MQTT topics. These events are not available by default. For information about how to turn on these events, see [Enable events for AWS IoT \(p. 982\)](#).

This section describes the AWS IoT lifecycle event messages.

Note

Lifecycle messages might be sent out of order. You might receive duplicate messages.

Connect/Disconnect events

AWS IoT publishes a message to the following MQTT topics when a client connects or disconnects:

- \$aws/events/presence/connected/*clientId* – A client connected to the message broker.

- `$aws/events/presence/disconnected/clientId` – A client disconnected from the message broker.

The following is a list of JSON elements that are contained in the connection/disconnection messages published to the `$aws/events/presence/connected/clientId` topic.

clientId

The client ID of the connecting or disconnecting client.

Note

Client IDs that contain # or + do not receive lifecycle events.

clientInitiatedDisconnect

True if the client initiated the disconnect. Otherwise, false. Found in disconnection messages only.

disconnectReason

The reason why the client is disconnecting. Found in disconnect messages only. The following table contains valid values.

Disconnect reason	Description
AUTH_ERROR	The client failed to authenticate or authorization failed.
CLIENT_INITIATED_DISCONNECT	The client indicates that it will disconnect. The client can do this by sending either a MQTT DISCONNECT control packet or a Close frame if the client is using a WebSocket connection.
CLIENT_ERROR	The client did something wrong that causes it to disconnect. For example, a client will be disconnected for sending more than 1 MQTT CONNECT packet on the same connection or if the client attempts to publish with a payload that exceeds the payload limit.
CONNECTION_LOST	The client-server connection is cut off. This can happen during a period of high network latency or when the internet connection is lost.
DUPLICATE_CLIENTID	The client is using a client ID that is already in use. In this case, the client that is already connected will be disconnected with this disconnect reason.
FORBIDDEN_ACCESS	The client is not allowed to be connected. For example, a client with a denied IP address will fail to connect.
MOTT_KEEP_ALIVE_TIMEOUT	If there is no client-server communication for 1.5x of the client's keep-alive time, the client is disconnected.
SERVER_ERROR	Disconnected due to unexpected server issues.
SERVER_INITIATED_DISCONNECT	Server intentionally disconnects a client for operational reasons.

Disconnect reason	Description
THROTTLED	The client is disconnected for exceeding a throttling limit.
WEBSOCKET_TTL_EXPIRATION	The client is disconnected because a WebSocket has been connected longer than its time-to-live value.

eventType

The type of event. Valid values are `connected` or `disconnected`.

ipAddress

The IP address of the connecting client. This can be in IPv4 or IPv6 format. Found in connection messages only.

principalIdentifier

The credential used to authenticate. For TLS mutual authentication certificates, this is the certificate ID. For other connections, this is IAM credentials.

sessionIdentifier

A globally unique identifier in AWS IoT that exists for the life of the session.

timestamp

An approximation of when the event occurred, expressed in milliseconds since the Unix epoch. The accuracy of the timestamp is +/- 2 minutes.

versionNumber

The version number for the lifecycle event. This is a monotonically increasing long integer value for each client ID connection. The version number can be used by a subscriber to infer the order of lifecycle events.

Note

The connect and disconnect messages for a client connection have the same version number.

The version number might skip values and is not guaranteed to be consistently increasing by 1 for each event.

If a client is not connected for approximately one hour, the version number is reset to 0. For persistent sessions, the version number is reset to 0 after a client has been disconnected longer than the configured time-to-live (TTL) for the persistent session.

A connect message has the following structure.

```
{
  "clientId": "186b5",
  "timestamp": 1573002230757,
  "eventType": "connected",
  "sessionIdentifier": "a4666d2a7d844ae4ac5d7b38c9cb7967",
  "principalIdentifier": "12345678901234567890123456789012",
  "ipAddress": "192.0.2.0",
  "versionNumber": 0
}
```

A disconnect message has the following structure.

```
{  
    "clientId": "186b5",  
    "timestamp": 1573002340451,  
    "eventType": "disconnected",  
    "sessionIdentifier": "a4666d2a7d844ae4ac5d7b38c9cb7967",  
    "principalIdentifier": "12345678901234567890123456789012",  
    "clientInitiatedDisconnect": true,  
    "disconnectReason": "CLIENT_INITIATED_DISCONNECT",  
    "versionNumber": 0  
}
```

Handling client disconnections

The best practice is to always have a wait state implemented for lifecycle events, including Last Will and Testament (LWT) messages. When a disconnect message is received, your code should wait a period of time and verify a device is still offline before taking action. One way to do this is by using [SQS Delay Queues](#). When a client receives a LWT or a lifecycle event, you can enqueue a message (for example, for 5 seconds). When that message becomes available and is processed (by Lambda or another service), you can first check if the device is still offline before taking further action.

Subscribe/Unsubscribe events

AWS IoT publishes a message to the following MQTT topic when a client subscribes or unsubscribes to an MQTT topic:

```
$aws/events/subscriptions/subscribed/clientId
```

or

```
$aws/events/subscriptions/unsubscribed/clientId
```

Where *clientId* is the MQTT client ID that connects to the AWS IoT message broker.

The message published to this topic has the following structure:

```
{  
    "clientId": "186b5",  
    "timestamp": 1460065214626,  
    "eventType": "subscribed" | "unsubscribed",  
    "sessionIdentifier": "00000000-0000-0000-000000000000",  
    "principalIdentifier": "000000000000/ABCDEFGHIJKLMNPQRSTUVWXYZ:some-user/  
ABCDEFGHIJKLMNPQRSTUVWXYZ:some-user",  
    "topics" : ["foo/bar","device/data","dog/cat"]  
}
```

The following is a list of JSON elements that are contained in the subscribed and unsubscribed messages published to the `$aws/events/subscriptions/subscribed/clientId` and `$aws/events/subscriptions/unsubscribed/clientId` topics.

clientId

The client ID of the subscribing or unsubscribing client.

Note

Client IDs that contain # or + do not receive lifecycle events.

eventType

The type of event. Valid values are `subscribed` or `unsubscribed`.

principalIdentifier

The credential used to authenticate. For TLS mutual authentication certificates, this is the certificate ID. For other connections, this is IAM credentials.

sessionId

A globally unique identifier in AWS IoT that exists for the life of the session.

timestamp

An approximation of when the event occurred, expressed in milliseconds since the Unix epoch. The accuracy of the timestamp is +/- 2 minutes.

topics

An array of the MQTT topics to which the client has subscribed.

Note

Lifecycle messages might be sent out of order. You might receive duplicate messages.

AWS IoT Core for LoRaWAN

AWS IoT Core for LoRaWAN is a fully managed LoRaWAN network server (LNS) that provides gateway management using the Configuration and Update Server (CUPS) and Firmware Updates Over-The-Air (FUOTA) capabilities. You can replace your private LNS with AWS IoT Core for LoRaWAN and connect your Long Range Wide Area Network (LoRaWAN) devices and gateways to AWS IoT Core. By doing so, you'll reduce the maintenance, operational costs, setup time, and overhead costs.

Introduction

LoRaWAN devices are long-range, low-power, battery-operated devices that use the LoRaWAN protocol to operate in a license-free radio spectrum. LoRaWAN is a Low Power Wide Area Network (LPWAN) communication protocol that is built on LoRa. LoRa is the physical layer protocol that enables low power, wide-area communication between devices.

You can onboard your LoRaWAN devices the same way you would onboard other IoT devices to AWS IoT. To connect your LoRaWAN devices to AWS IoT, you must use a LoRaWAN gateway. The gateway acts as a bridge to connect your device to AWS IoT Core for LoRaWAN and to exchange messages. AWS IoT Core for LoRaWAN uses the AWS IoT rules engine to route the messages from your LoRaWAN devices to other AWS IoT services.

To reduce development effort and quickly onboard your devices to AWS IoT Core for LoRaWAN, we recommend that you use LoRaWAN-certified end devices. For more information, see the [AWS IoT Core for LoRaWAN product overview page](#). For information about getting your devices LoRaWAN certified, see [Certifying LoRaWAN products](#).

How to use AWS IoT Core for LoRaWAN

You can quickly onboard your LoRaWAN devices and gateways to AWS IoT Core for LoRaWAN by using the console or the AWS IoT Wireless API.

Using the console

To onboard your LoRaWAN devices and gateways by using the AWS Management Console, sign in to the AWS Management Console and navigate to the [AWS IoT Core for LoRaWAN](#) page in the AWS IoT console. You can then use the **Intro** section to add your gateways and devices to AWS IoT Core for LoRaWAN. For more information, see [Using the console to onboard your device and gateway to AWS IoT Core for LoRaWAN \(p. 1002\)](#).

Using the API or CLI

You can onboard both LoRaWAN and Sidewalk devices by using the [AWS IoT Wireless API](#). The AWS IoT Wireless API that AWS IoT Core for LoRaWAN is built on is supported by the AWS SDK. For more information, see [AWS SDKs and Toolkits](#).

You can use the AWS CLI to run commands for onboarding and managing your LoRaWAN gateways and devices. For more information, see [AWS IoT Wireless CLI reference](#).

AWS IoT Core for LoRaWAN Regions and endpoints

AWS IoT Core for LoRaWAN provides support for control plane and data plane API endpoints that are specific to your AWS Region. The data plane API endpoints are specific to your AWS account and AWS Region. For more information about the AWS IoT Core for LoRaWAN endpoints, see [AWS IoT Core for LoRaWAN Endpoints](#) in the *AWS General Reference*.

In the Regions US East (N. Virginia) and Europe (Ireland), you can connect your devices to AWS IoT Core for LoRaWAN through AWS PrivateLink in your virtual private cloud (VPC), instead of connecting over the public internet. For more information, see [Connecting to AWS IoT Core for LoRaWAN through a VPC interface endpoint \(p. 1021\)](#).

AWS IoT Core for LoRaWAN has quotas that apply to device data that is transmitted between the devices and the maximum TPS for the AWS IoT Wireless API operations. For more information, see [AWS IoT Core for LoRaWAN quotas](#) in the *AWS General Reference*

AWS IoT Core for LoRaWAN pricing

When you sign up for AWS, you can get started with AWS IoT Core for LoRaWAN for no charge by using the [AWS Free Tier](#).

For more information about general product overview and pricing, see [AWS IoT Core pricing](#).

What is AWS IoT Core for LoRaWAN?

AWS IoT Core for LoRaWAN replaces a private LoRaWAN network server (LNS) by connecting your LoRaWAN devices and gateways to AWS. Using the AWS IoT rules engine, you can route messages received from LoRaWAN devices, where they can be formatted and sent to other AWS IoT services. To secure device communications with AWS IoT, AWS IoT Core for LoRaWAN uses X.509 certificates.

AWS IoT Core for LoRaWAN manages the service and device policies that AWS IoT Core requires to communicate with the LoRaWAN gateways and devices. AWS IoT Core for LoRaWAN also manages the destinations that describe the AWS IoT rules that send device data to other services.

With AWS IoT Core for LoRaWAN, you can:

- Onboard and connect LoRaWAN devices and gateways to AWS IoT without the need to set up and manage a private LNS.
- Connect LoRaWAN devices that comply to 1.0.x or 1.1 LoRaWAN specifications standardized by LoRa Alliance. These devices can operate in class A, class B, or class C mode.
- Use LoRaWAN gateways that support LoRa Basics Station version 2.0.4 or later. All gateways that are qualified for AWS IoT Core for LoRaWAN run a compatible version of LoRa Basics Station.
- Monitor signal strength, bandwidth, and spreading factor by using AWS IoT Core for LoRaWAN's adaptive data rate, and optimize the data rate if needed.
- Update LoRaWAN gateways' firmware using the CUPS service and the firmware of LoRaWAN devices using Firmware Updates Over-The-Air (FUOTA).

Topics

- [What is LoRaWAN? \(p. 1000\)](#)
- [How AWS IoT Core for LoRaWAN works \(p. 1000\)](#)

What is LoRaWAN?

The [LoRa Alliance](#) describes LoRaWAN as, "*a Low Power, Wide Area (LPWA) networking protocol designed to wirelessly connect battery operated ‘things’ to the internet in regional, national or global networks, and targets key Internet of Things (IoT) requirements such as bi-directional communication, end-to-end security, mobility and localization services.*".

LoRa and LoRaWAN

The LoRaWAN protocol is a Low Power Wide Area Networking (LPWAN) communication protocol that functions on LoRa. The LoRaWAN specification is open so anyone can set up and operate a LoRa network.

LoRa is a wireless audio frequency technology that operates in a license-free radio frequency spectrum. LoRa is a physical layer protocol that uses spread spectrum modulation and supports long-range communication at the cost of a narrow bandwidth. It uses a narrow band waveform with a central frequency to send data, which makes it robust to interference.

Characteristics of LoRaWAN technology

- Long range communication up to 10 miles in line of sight.
- Long battery duration of up to 10 years. For enhanced battery life, you can operate your devices in class A or class B mode, which requires increased downlink latency.
- Low cost for devices and maintenance.
- License-free radio spectrum but region-specific regulations apply.
- Low power but has a limited payload size of 51 bytes to 241 bytes depending on the data rate. The data rate can be 0,3 Kbit/s – 27 Kbit/s data rate with a 222 maximal payload size.

Learn more about LoRaWAN

The following links contain helpful information about the LoRaWAN technology and about LoRa Basics Station, which is the software that runs on your LoRaWAN gateways for connecting end devices to AWS IoT Core for LoRaWAN.

- [The Things Fundamentals on LoRaWAN](#)

The Things Fundamentals on LoRaWAN contains an introductory video that covers the fundamentals of LoRaWAN and a series of chapters that'll help you learn about LoRa and LoRaWAN.

- [What is LoRaWAN](#)

LoRa Alliance provides a technical overview of LoRa and LoRaWAN, including a summary of the LoRaWAN specifications in different Regions.

- [LoRa Basics Station](#)

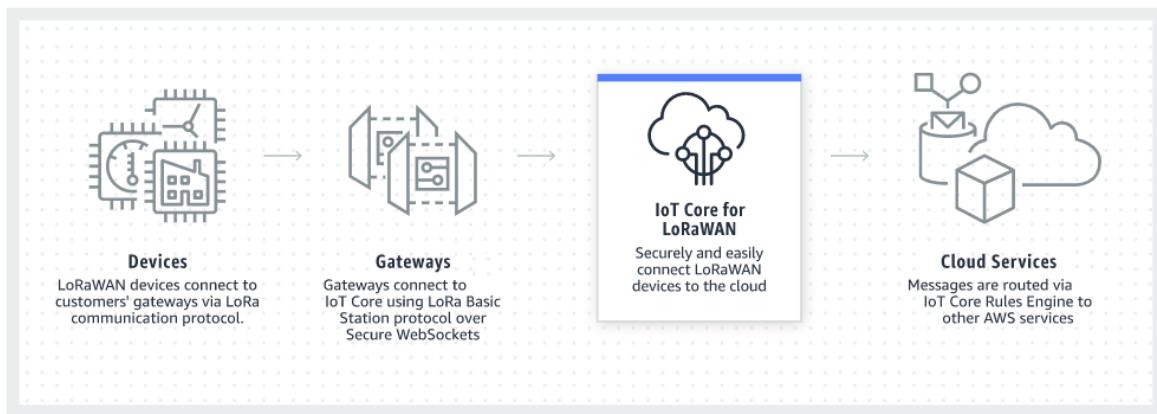
Semtech Corporation provides helpful concepts about LoRa basics for gateways and end nodes. LoRa Basics Station, an open source software that runs on your LoRaWAN gateway, is maintained and distributed through Semtech Corporation's [GitHub](#) repository. You can also learn about the LNS and CUPS protocols that describe how to exchange LoRaWAN data and perform configuration updates.

How AWS IoT Core for LoRaWAN works

The LoRaWAN network architecture is deployed in a star of stars topology in which gateways relay information between end devices and the LoRaWAN network server (LNS).

AWS IoT Core for LoRaWAN helps you connect and manage wireless LoRaWAN (low-power long-range Wide Area Network) devices and replaces the need for you to develop and operate an LNS. Long range WAN (LoRaWAN) devices and gateways can connect to AWS IoT Core by using AWS IoT Core for LoRaWAN.

The following shows how a LoRaWAN device interacts with AWS IoT Core for LoRaWAN. It also shows how AWS IoT Core for LoRaWAN replaces an LNS and communicates with other AWS services in the AWS Cloud.



LoRaWAN devices communicate with AWS IoT Core through LoRaWAN gateways. AWS IoT Core for LoRaWAN manages the service and device policies that AWS IoT Core requires to manage and communicate with the LoRaWAN gateways and devices. AWS IoT Core for LoRaWAN also manages the destinations that describe the AWS IoT rules that send device data to other services.

Get started using AWS IoT Core for LoRaWAN

1. Select the wireless devices and LoRaWAN gateways that you'll need.

The [AWS Partner Device Catalog](#) contains gateways and developer kits that are qualified for use with AWS IoT Core for LoRaWAN. For more information, see [Using qualified gateways from the AWS Partner Device Catalog \(p. 1031\)](#).

2. Add your wireless devices and LoRaWAN gateways to AWS IoT Core for LoRaWAN.

[Connecting gateways and devices to AWS IoT Core for LoRaWAN \(p. 1002\)](#) gives you information about how to describe your resources and add your wireless devices and LoRaWAN gateways to AWS IoT Core for LoRaWAN. You'll also learn how to configure the other AWS IoT Core for LoRaWAN resources that you'll need to manage these devices and send their data to AWS services.

3. Complete your AWS IoT Core for LoRaWAN solution.

Start with [our sample AWS IoT Core for LoRaWAN solution](#) and make it yours.

AWS IoT Core for LoRaWAN resources

The following resources will help you get familiar with the LoRaWAN technology and AWS IoT Core for LoRaWAN.

- [Getting Started with AWS IoT Core for LoRaWAN](#)

The following video describes how AWS IoT Core for LoRaWAN works and walks you through the process of adding LoRaWAN gateways from the AWS Management Console.

- [AWS IoT Core for LoRaWAN workshop](#)

The workshop covers fundamentals of LoRaWAN technology and its implementation with AWS IoT Core for LoRaWAN. You can also use the workshop to walk through labs that show how to connect your gateway and device to AWS IoT Core for LoRaWAN for building a sample IoT solution.

Connecting gateways and devices to AWS IoT Core for LoRaWAN

AWS IoT Core for LoRaWAN helps you connect and manage wireless LoRaWAN (low-power long-range Wide Area Network) devices and replaces the need for you to develop and operate a LNS. Long range WAN (LoRaWAN) devices and gateways can connect to AWS IoT Core by using AWS IoT Core for LoRaWAN.

Naming conventions for your devices, gateways, profiles, and destinations

Before you get started with AWS IoT Core for LoRaWAN and creating the resources, consider the naming convention of your devices, gateways, and destination.

AWS IoT Core for LoRaWAN assigns unique IDs to the resources you create for wireless devices, gateways, and profiles; however, you can also give your resources more descriptive names to make it easier to identify them. Before you add devices, gateways, profiles, and destinations to AWS IoT Core for LoRaWAN, consider how you'll name them to make them easier to manage.

You can also add tags to the resources you create. Before you add your LoRaWAN devices, consider how you might use tags to identify and manage your AWS IoT Core for LoRaWAN resources. Tags can be modified after you add them.

For more information about naming and tagging, see [Describe your AWS IoT Core for LoRaWAN resources \(p. 1003\)](#).

Mapping of device data to service data

The data from LoRaWAN wireless devices is often encoded to optimize bandwidth. These encoded messages arrive at AWS IoT Core for LoRaWAN in a format that might not be easily used by other AWS services. AWS IoT Core for LoRaWAN uses AWS IoT rules that can use AWS Lambda functions to process and decode the device messages to a format that other AWS services can use.

To transform device data and send it to other AWS services, you need to know:

- The format and contents of the data that the wireless devices send.
- The service to which you want to send the data.
- The format that service requires.

Using that information, you can create the AWS IoT rule that performs the conversion and sends the converted data to the AWS services that will use it.

Using the console to onboard your device and gateway to AWS IoT Core for LoRaWAN

You can use the console interface or the API to add your LoRaWAN gateway and devices. If you're using AWS IoT Core for LoRaWAN for the first time, we recommend that you use the console. The console

interface is most practical when managing a few AWS IoT Core for LoRaWAN resources at a time. When managing large numbers of AWS IoT Core for LoRaWAN resources, consider creating more automated solutions by using the AWS IoT Wireless API.

Much of the data that you enter when configuring AWS IoT Core for LoRaWAN resources is provided by the devices' vendors and is specific to the LoRaWAN specifications they support. The following topics describe how you can describe your AWS IoT Core for LoRaWAN resources and use the console or the API to add your gateways and devices.

Topics

- [Describe your AWS IoT Core for LoRaWAN resources \(p. 1003\)](#)
- [Onboard your gateways to AWS IoT Core for LoRaWAN \(p. 1004\)](#)
- [Onboard your devices to AWS IoT Core for LoRaWAN \(p. 1010\)](#)

Describe your AWS IoT Core for LoRaWAN resources

If you're using AWS IoT Core for LoRaWAN for the first time, you can add your first LoRaWAN gateway and device by using the [AWS IoT Core for LoRaWAN Intro](#) page of the AWS IoT console.

Before you get started with creating the resources, consider the naming convention of your devices, gateways, and destination. AWS IoT Core for LoRaWAN provides several options to identify the resources you create. While AWS IoT Core for LoRaWAN resources are given a unique ID when they're created, this ID is not descriptive nor can it be changed after the resource is created. You can also assign a name, add a description, and attach tags and tag values to most AWS IoT Core for LoRaWAN resources to make it more convenient to select, identify, and manage your AWS IoT Core for LoRaWAN resources.

• [Resource names \(p. 1003\)](#)

For gateways, devices, and profiles, the resource name is an optional field that you can change after the resource is created. The name appears in the lists displayed on the resource hub pages.

For destinations, you provide a name that is unique in your AWS account and AWS Region. You can't change the destination name after you create the destination resource.

While a name can have up to 256 characters, the display space in the resource hub is limited. Make sure that the distinguishing part of the name appears in the first 20 to 30 characters, if possible.

• [Resource tags \(p. 1004\)](#)

Tags are key-value pairs of metadata that can be attached to AWS resources. You choose both tag keys and their corresponding values.

Gateways, destinations, and profiles can have up to 50 tags attached to them. Devices don't support tags.

Resource names

AWS IoT Core for LoRaWAN resource support for name

Resource	Name field support	
Destination	Name is unique ID of resource and can't be changed.	
Device	Name is optional descriptor of resource and can be changed.	

Resource	Name field support
Gateway	Name is optional descriptor of resource and can be changed.
Profile	Name is optional descriptor of resource and can be changed.

The name field appears in resource hub lists of resources; however, the space is limited and so only the first 15-30 characters of the name might be visible.

When selecting names for your resources, consider how you want them to identify the resources and how they'll be displayed in the console.

Description

Destination, device, and gateway resources also support a description field, which can accept up to 2,048 characters. The description field appears only in the individual resource's detail page. While the description field can hold a lot of information, because it only appears in the resource's detail page, it isn't convenient for scanning in the context of multiple resources.

Resource tags

AWS IoT Core for LoRaWAN resource support for AWS tags

Resource	AWS tag support
Destination	Up to 50 AWS tags can be added to the resource.
Device	This resource doesn't support AWS tags.
Gateway	Up to 50 AWS tags can be added to the resource.
Profile	Up to 50 AWS tags can be added to the resource.

Tags are words or phrases that act as metadata that you can use to identify and organize your AWS resources. You can think of the tag key as a category of information and the tag value as a specific value in that category.

For example, you might have a tag value of *color* and then give some resources a value of *blue* for that tag and others a value of *red*. With that, you could use the [Tag editor](#) in the AWS console to find the resources with a *color* tag value of *blue*.

For more information about tagging and tagging strategies, see [Tag editor](#).

Onboard your gateways to AWS IoT Core for LoRaWAN

If you're using AWS IoT Core for LoRaWAN for the first time, you can add your first LoRaWAN gateway and device by using the console.

Before onboarding your gateway

Before you onboard your gateway to AWS IoT Core for LoRaWAN, we recommend that you:

- Use gateways that are qualified for use with AWS IoT Core for LoRaWAN. These gateways connect to AWS IoT Core without any additional configuration settings and have a compatible version of the [LoRa Basics Station](#) software running on them. For more information, see [Managing gateways with AWS IoT Core for LoRaWAN \(p. 1030\)](#).
- Consider the naming convention of the resources that you create so that you can more easily manage them. For more information, see [Describe your AWS IoT Core for LoRaWAN resources \(p. 1003\)](#).
- Have the configuration parameters that are unique to each gateway ready to enter in advance, which makes entering the data into the console go more smoothly. The wireless gateway configuration parameters that AWS IoT requires to communicate with and manage the gateway include the gateway's EUI and its LoRa frequency band.

For onboarding your gateways to AWS IoT Core for LoRaWAN:

- [Consider frequency band selection and add necessary IAM role \(p. 1005\)](#)
- [Add a gateway to AWS IoT Core for LoRaWAN \(p. 1006\)](#)
- [Connect your LoRaWAN gateway and verify its connection status \(p. 1009\)](#)

Consider frequency band selection and add necessary IAM role

Before you add your gateway to AWS IoT Core for LoRaWAN, we recommend that you consider the frequency band in which your gateway will be operating and add the necessary IAM role for connecting your gateway to AWS IoT Core for LoRaWAN.

Note

If you're adding your gateway using the console, click **Create role** in the console to create the necessary IAM role so you can then skip these steps. You need to perform these steps only if you're using the CLI to create the gateway.

Consider selection of LoRa frequency bands for your gateways and device connection

AWS IoT Core for LoRaWAN supports EU863-870, US902-928, AU915, and AS923-1 frequency bands, which you can use to connect your gateways and devices that are physically present in countries that support the frequency ranges and characteristics of these bands. The EU863-870 and US902-928 bands are commonly used in Europe and North America, respectively. The AS923-1 band is commonly used in Australia, New Zealand, Japan, and Singapore among other countries. The AU915 is used in Australia and Argentina among other countries. For more information about which frequency band to use in your region or country, see [LoRaWAN® Regional Parameters](#).

LoRa Alliance publishes LoRaWAN specifications and regional parameter documents that are available for download from the LoRa Alliance website. The LoRa Alliance regional parameters help companies decide which frequency band to use in their region or country. AWS IoT Core for LoRaWAN's frequency band implementation follows the recommendation in the regional parameters specification document. These regional parameters are grouped into a set of radio parameters, along with a frequency allocation that is adapted to the Industrial, Scientific, and Medical (ISM) band. We recommend that you work with the compliance teams to ensure that you meet any applicable regulatory requirements.

Add an IAM role to allow the Configuration and Update Server (CUPS) to manage gateway credentials

This procedure describes how to add an IAM role that will allow the Configuration and Update Server (CUPS) to manage gateway credentials. Make sure you perform this procedure before a LoRaWAN gateway tries to connect with AWS IoT Core for LoRaWAN; however, you need to do this only once.

Add the IAM role to allow the Configuration and Update Server (CUPS) to manage gateway credentials

1. Open the [Roles hub of the IAM console](#) and choose **Create role**.
2. If you think that you might have already added the **IoTWirelessGatewayCertManagerRole** role, in the search bar, enter **IoTWirelessGatewayCertManagerRole**.

If you see an **IoTWirelessGatewayCertManagerRole** role in the search results, you have the necessary IAM role. You can leave the procedure now.

If the search results are empty, you don't have the necessary IAM role. Continue the procedure to add it.

3. In **Select type of trusted entity**, choose **Another AWS account**.
4. In **Account ID**, enter your AWS account ID, and then choose **Next: Permissions**.
5. In the search box, enter **AWSIoTWirelessGatewayCertManager**.
6. In the list of search results, select the policy named **AWSIoTWirelessGatewayCertManager**.
7. Choose **Next: Tags**, and then choose **Next: Review**.
8. In **Role name**, enter **IoTWirelessGatewayCertManagerRole**, and then choose **Create role**.
9. To edit the new role, in the confirmation message, choose **IoTWirelessGatewayCertManagerRole**.
10. In **Summary**, choose the **Trust relationships** tab, and then choose **Edit trust relationship**.
11. In **Policy Document**, change the **Principal** property to look like this example.

```
"Principal": {  
    "Service": "iotwireless.amazonaws.com"  
},
```

After you change the **Principal** property, the complete policy document should look like this example.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Principal": {  
                "Service": "iotwireless.amazonaws.com"  
            },  
            "Action": "sts:AssumeRole",  
            "Condition": {}  
        }  
    ]  
}
```

12. To save your changes and exit, choose **Update Trust Policy**.

You've now created the **IoTWirelessGatewayCertManagerRole**. You won't need to do this again.

If you performed this procedure while you were adding a gateway, you can close this window and the IAM console and return to the AWS IoT console to finish adding the gateway.

Add a gateway to AWS IoT Core for LoRaWAN

You can add your gateway to AWS IoT Core for LoRaWAN by using the console or the CLI.

Before adding your gateway, we recommend that you consider the factors mentioned in the **Before onboarding your gateway** section of [Onboard your gateways to AWS IoT Core for LoRaWAN \(p. 1004\)](#).

If you're adding your gateway for the first time, we recommend that you use the console. If you want to add your gateway by using the CLI instead, you must have already created the necessary IAM role so that the gateway can connect with AWS IoT Core for LoRaWAN. For information about how to create the role, see [Add an IAM role to allow the Configuration and Update Server \(CUPS\) to manage gateway credentials \(p. 1005\)](#).

Add a gateway using the console

Navigate to the [AWS IoT Core for LoRaWAN Intro](#) page of the AWS IoT console and choose **Get started**, and then choose **Add gateway**. If you've already added a gateway, choose **View gateway** to view the gateway that you added. If you would like to add more gateways, choose **Add gateway**.

1. Provide gateway details and frequency band information

Use the **Gateway details** section to provide information about the device configuration data such as the Gateway's EUI and the frequency band configuration.

- **Gateway's EUI**

The EUI (Extended Unique Identifier) of the individual gateway device. The EUI is a 16-digit alphanumeric code, such as c0ee40ffff29df10, that uniquely identifies a gateway in your LoRaWAN network. This information is specific to your gateway model and you can find it on your gateway device or in its user manual.

Note

The Gateway's EUI is different from the Wi-Fi MAC address that you may see printed on your gateway device. The EUI follows a EUI-64 standard that uniquely identifies your gateway and therefore cannot be reused in other AWS accounts and regions.

- **Frequency band (RFRegion)**

The gateway's frequency band. You can choose from US915, EU868, AU915, or AS923-1, depending on what your gateway supports and which country or region the gateway is physically connecting from. For more information about the bands, see [Consider selection of LoRa frequency bands for your gateways and device connection \(p. 1005\)](#).

2. Specify your wireless gateway configuration data (optional)

These fields are optional and you can use them to provide additional information about the gateway and its configuration.

- **Name, Description, and Tags for your gateway**

The information in these optional fields comes from how you organize and describe the elements in your wireless system. You can assign a **Name** to the gateway, use the **Description** field to provide information about the gateway, and use **Tags** to add key-value pairs of metadata about the gateway. For more information on naming and describing your resources, see [Describe your AWS IoT Core for LoRaWAN resources \(p. 1003\)](#).

- **LoRaWAN configuration using subbands and filters**

Optionally, you can also specify LoRaWAN configuration data such as the subbands that you want to use and filters that can control the flow of traffic. For this tutorial, you can skip these fields. For more information, see [Configure your gateway's subbands and filtering capabilities \(p. 1031\)](#).

3. Associate an AWS IoT thing with the gateway

Specify whether to create an AWS IoT thing and associate it with the gateway. Things in AWS IoT can make it easier to search and manage your devices. Associating a thing with your gateway lets the gateway access other AWS IoT Core features.

4. Create and download the gateway certificate

To authenticate your gateway so that it can securely communicate with AWS IoT, your LoRaWAN gateway must present a private key and certificate to AWS IoT Core for LoRaWAN. Create a **Gateway certificate** so that AWS IoT can verify your gateway's identity by using the X.509 Standard.

Click the **Create certificate** button and download the certificate files. You'll use them later to configure your gateway.

5. Copy the CUPS and LNS endpoints and download certificates

Your LoRaWAN gateway must connect to a CUPS or LNS endpoint when establishing a connection to AWS IoT Core for LoRaWAN. We recommend that you use the CUPS endpoint as it also provides configuration management. To verify the authenticity of AWS IoT Core for LoRaWAN endpoints, your gateway will use a trust certificate for each of the CUPS and LNS endpoints.

Click the **Copy** button to copy the CUPS and LNS endpoints. You'll need this information later to configure your gateway. Then click the **Download server trust certificates** button to download the trust certificates for the CUPS and LNS endpoints.

6. Create the IAM role for the gateway permissions

You need to add an IAM role that allows the Configuration and Update Server (CUPS) to manage gateway credentials. You must do this before a LoRaWAN gateway tries to connect with AWS IoT Core for LoRaWAN; however, you need to do it only once.

To create the **IoTWirelessGatewayCertManager** IAM role for your account, click the **Create role** button. If the role already exists, select it from the dropdown list.

Click **Submit** to complete the gateway creation.

Add a gateway by using the API

If you're adding a gateway for the first time by using the API or CLI, you must add the **IoTWirelessGatewayCertManager** IAM role so that the gateway can connect with AWS IoT Core for LoRaWAN. For information about how to create the role, see the following section [Add an IAM role to allow the Configuration and Update Server \(CUPS\) to manage gateway credentials \(p. 1005\)](#).

The following lists describe the API actions that perform the tasks associated with adding, updating, or deleting a LoRaWAN gateway.

AWS IoT Wireless API actions for AWS IoT Core for LoRaWAN gateways

- [CreateWirelessGateway](#)
- [GetWirelessGateway](#)
- [ListWirelessGateways](#)
- [UpdateWirelessGateway](#)
- [DeleteWirelessGateway](#)

For the complete list of the actions and data types available to create and manage AWS IoT Core for LoRaWAN resources, see the [AWS IoT Wireless API reference](#).

How to use the AWS CLI to add a gateway

You can use the AWS CLI to create a wireless gateway by using the `create-wireless-gateway` command. The following example creates a wireless LoRaWAN device gateway. You can also provide an `input.json` file that will contain additional details such as the gateway certificate and provisioning credentials.

Note

You can also perform this procedure with the API by using the methods in the AWS API that correspond to the CLI commands shown here.

```
aws iotwireless create-wireless-gateway \
--lorawan GatewayEui="a1b2c3d4567890ab",RfRegion="US915" \
--name "myFirstLoRaWANGateway" \
--description "Using my first LoRaWAN gateway"
--cli-input-json input.json
```

For information about the CLIs that you can use, see [AWS CLI reference](#)

Connect your LoRaWAN gateway and verify its connection status

Before you can check the gateway connection status, you must have already added your gateway and connected it to AWS IoT Core for LoRaWAN. For information about how to add your gateway, see [Add a gateway to AWS IoT Core for LoRaWAN \(p. 1006\)](#).

Connect your gateway to AWS IoT Core for LoRaWAN

After you've added your gateway, connect to the configuration interface of your gateway to enter the configuration information and trust certificates.

After adding the gateway's information to AWS IoT Core for LoRaWAN, add some AWS IoT Core for LoRaWAN information to the gateway device. The documentation provided by the gateway's vendor should describe the process for uploading the certificate files to the gateway and configuring the gateway device to communicate with AWS IoT Core for LoRaWAN.

Gateways qualified for use with AWS IoT Core for LoRaWAN

For instructions on how to configure your LoRaWAN gateway, refer to the [configure gateway device](#) section of the AWS IoT Core for LoRaWAN workshop. Here, you'll find information about instructions for connecting gateways that are qualified for use with AWS IoT Core for LoRaWAN.

Gateways that support CUPS protocol

The following instructions show how you can connect your gateways that support the CUPS protocol.

1. Upload the following files that you obtained when adding your gateway.
 - Gateway device certificate and private key files.
 - Trust certificate file for CUPS endpoint, `cups.trust`.
2. Specify the CUPS endpoint URL that you obtained previously. The endpoint will be of the format `prefix.cups.lorawan.region.amazonaws.com:443`.

For details about how to obtain this information, see [Add a gateway to AWS IoT Core for LoRaWAN \(p. 1006\)](#).

Gateways that support LNS protocol

The following instructions show how you can connect your gateways that support the LNS protocol.

1. Upload the following files that you obtained when adding your gateway.
 - Gateway device certificate and private key files.
 - Trust certificate file for LNS endpoint, `lns.trust`.
2. Specify the LNS endpoint URL that you obtained previously. The endpoint will be of the format `prefix.lns.lorawan.region.amazonaws.com:443`.

For details about how to obtain this information, see [Add a gateway to AWS IoT Core for LoRaWAN \(p. 1006\)](#).

After that you've connected your gateway to AWS IoT Core for LoRaWAN, you can check the status of your connection and get information about when the last uplink was received by using the console or the API.

Check gateway connection status using the console

To check the connection status using the console, navigate to the [Gateways](#) page of the AWS IoT console and choose the gateway you've added. In the **LoRaWAN specific details** section of the Gateway details page, you'll see the connection status and the date and time the last uplink was received.

Check gateway connection status using the API

To check the connection status using the API, use the `GetWirelessGatewayStatistics` API. This API doesn't have a request body and only contains a response body that shows whether the gateway is connected and when the last uplink was received.

```
HTTP/1.1 200
Content-type: application/json

{
    "ConnectionStatus": "Connected",
    "LastUplinkReceivedAt": "2021-03-24T23:13:08.476015749Z",
    "WirelessGatewayId": "30cbdcf3-86de-4291-bfab-5bfa2b12bad5"
}
```

Onboard your devices to AWS IoT Core for LoRaWAN

After you have onboarded your gateway to AWS IoT Core for LoRaWAN and verified its connection status, you can onboard your wireless devices. For information about how to onboard your gateways, see [Onboard your gateways to AWS IoT Core for LoRaWAN \(p. 1004\)](#).

LoRaWAN devices use a LoRaWAN protocol to exchange data with cloud-hosted applications. AWS IoT Core for LoRaWAN supports devices that comply to 1.0.x or 1.1 LoRaWAN specifications standardized by LoRa Alliance.

A LoRaWAN device typically contains one or more sensors and actors. The devices send uplink telemetry data through LoRaWAN gateways to AWS IoT Core for LoRaWAN. Cloud-hosted applications can control the sensors by sending downlink commands to LoRaWAN devices through LoRaWAN gateways.

Before onboarding your wireless device

Before you onboard your wireless device to AWS IoT Core for LoRaWAN, you need to have the following information ready in advance:

- **LoRaWAN specification and wireless device configuration**

Having the configuration parameters that are unique to each device ready in advance makes entering the data into the console go more smoothly. The specific parameters that you need to enter depend on the LoRaWAN specification that the device uses. For the complete listing of its specifications and configuration parameters, see each device's documentation.

- **Device name and description (optional)**

The information in these optional fields comes from how you organize and describe the elements in your wireless system. For more information about naming and describing your resources, see [Describe your AWS IoT Core for LoRaWAN resources \(p. 1003\)](#).

- **Device and service profiles**

Have some wireless device configuration parameters ready that are shared by many devices and can be stored in AWS IoT Core for LoRaWAN as device and service profiles. The configuration parameters are found in the device's documentation or on the device itself. You'll want to identify a device profile that matches the configuration parameters of the device, or create one if necessary, before you add the device. For more information, see [Add profiles to AWS IoT Core for LoRaWAN \(p. 1013\)](#).

- **AWS IoT Core for LoRaWAN destination**

Each device must be assigned to a destination that will process its messages to send to AWS IoT and other services. The AWS IoT rules that process and send the device messages are specific to the device's message format. To process the messages from the device and send them to the correct service, identify the destination you'll create to use with the device's messages and assign it to the device.

To onboard your wireless device to AWS IoT Core for LoRaWAN

- [Add your wireless device to AWS IoT Core for LoRaWAN \(p. 1011\)](#)
- [Add profiles to AWS IoT Core for LoRaWAN \(p. 1013\)](#)
- [Add destinations to AWS IoT Core for LoRaWAN \(p. 1015\)](#)
- [Create rules to process LoRaWAN device messages \(p. 1018\)](#)
- [Connect your LoRaWAN device and verify its connection status \(p. 1020\)](#)

Add your wireless device to AWS IoT Core for LoRaWAN

If you're adding your wireless device for the first time, we recommend that you use the console. Navigate to the [AWS IoT Core for LoRaWAN Intro](#) page of the AWS IoT console, choose **Get started**, and then choose **Add device**. If you've already added a device, choose **View device** to view the gateway that you added. If you would like to add more devices, choose **Add device**.

Alternatively, you can also add wireless devices from the [Devices](#) page of the AWS IoT console.

Add your wireless device specification to AWS IoT Core for LoRaWAN using the console

Choose a **Wireless device specification** based on your activation method and the LoRaWAN version. Once selected, your data is encrypted with a key that AWS owns and manages for you.

OTAA and ABP activation modes

Before your LoRaWAN device can send uplink data, you must complete a process called *activation* or *join procedure*. To activate your device, you can either use OTAA (Over the air activation) or ABP (Activation by personalization).

ABP doesn't require a join procedure and uses static keys. When you use OTAA, your LoRaWAN device sends a join request and the Network Server can allow the request. We recommend that you use OTAA to activate your device because new session keys are generated for each activation, which makes it more secure.

LoRaWAN version

When you use OTAA, your LoRaWAN device and cloud-hosted applications share the root keys. These root keys depend on whether you're using version v1.0.x or v1.1. v1.0.x has only one root key, **AppKey** (Application Key) whereas v1.1 has two root keys, **AppKey** (Application Key) and **NwkKey** (Network Key). The session keys are derived based on the root keys for each activation. Both the **NwkKey** and **AppKey** are 32-digit hexadecimal values that your wireless vendor provided.

Wireless Device EUIs

After you select the **Wireless device specification**, you see the EUI (Extended Unique Identifier) parameters for the wireless device displayed on the console. You can find this information from the documentation for the device or the wireless vendor.

- **DevEUI**: 16-digit hexadeciml value that is unique to your device and found on the device label or its documentation.
- **AppEUI**: 16-digit hexadeciml value that is unique to the join server and found in the device documentation. In LoRaWAN version v1.1, the **AppEUI** is called as **JoinEUI**.

For more information about the unique identifiers, session keys, and root keys, refer to the [LoRa Alliance documentation](#).

Add your wireless device specification to AWS IoT Core for LoRaWAN by using the API

If you're adding a wireless device using the API, you must create your device profile and service profile first before creating the wireless device. You'll use the device profile and service profile ID when creating the wireless device. For information about how to create these profiles using the API, see [Add a device profile by using the API \(p. 1013\)](#).

The following lists describe the API actions that perform the tasks associated with adding, updating, or deleting a service profile.

AWS IoT Wireless API actions for service profiles

- [CreateWirelessDevice](#)
- [GetWirelessDevice](#)
- [ListWirelessDevices](#)
- [UpdateWirelessDevice](#)
- [DeleteWirelessDevice](#)

For the complete list of the actions and data types available to create and manage AWS IoT Core for LoRaWAN resources, see the [AWS IoT Wireless API reference](#).

How to use the AWS CLI to create a wireless device

You can use the AWS CLI to create a wireless device by using the [create-wireless-device](#) command. The following example creates a wireless device by using an input.json file to input the parameters.

Note

You can also perform this procedure with the API by using the methods in the AWS API that correspond to the CLI commands shown here.

Contents of input.json

```
{  
    "Description": "My LoRaWAN wireless device"  
    "DestinationName": "IoTWirelessDestination"  
    "LoRaWAN": {  
        "DeviceProfileId": "ab0c23d3-b001-45ef-6a01-2bc3de4f5333",  
        "ServiceProfileId": "fe98dc76-cd12-001e-2d34-5550432da100",  
        "OtaaV1_1": {  
            "AppKey": "3f4ca100e2fc675ea123f4eb12c4a012",  
            "JoinEui": "b4c231a359bc2e3d",  
            "NwkKey": "01c3f004a2d6efffe32c4eda14bcd2b4"  
        }  
    }  
}
```

```
  },
  "DevEui": "ac12efc654d23fc2"
},
"Name": "SampleIoTWirelessThing"
"Type": LoRaWAN
}
```

You can provide this file as input to the `create-wireless-device` command.

```
aws iotwireless create-wireless-device \
--cli-input-json file://input.json
```

For information about the CLIs that you can use, see [AWS CLI reference](#)

Add profiles to AWS IoT Core for LoRaWAN

Device and service profiles can be defined to describe common device configurations. These profiles describe configuration parameters that are shared by devices to make it easier to add those devices. AWS IoT Core for LoRaWAN supports device profiles and service profiles.

The configuration parameters and the values to enter into these profiles are provided by the device's manufacturer.

Add device profiles

Device profiles define the device capabilities and boot parameters that the network server uses to set the LoRaWAN radio access service. It includes selection of parameters such as LoRa frequency band, LoRa regional parameters version, and MAC version of the device. To learn about the different frequency bands, see [Consider selection of LoRa frequency bands for your gateways and device connection \(p. 1005\)](#).

Add a device profile by using the console

If you're adding a wireless device by using the console as described in [Add your wireless device specification to AWS IoT Core for LoRaWAN using the console \(p. 1011\)](#), after you've added the wireless device specification, you can add your device profile. Alternatively, you can also add wireless devices from the [Profiles](#) page of the AWS IoT console on the [LoRaWAN](#) tab.

You can choose from default device profiles or create a new device profile. We recommend that you use the default device profiles. If your application requires you to create a device profile, provide a **Device profile name**, select the **Frequency band (RfRegion)** that you're using for the device and gateway, and keep the other settings to the default values, unless specified otherwise in the device documentation.

Add a device profile by using the API

If you're adding a wireless device by using the API, you must create your device profile before creating the wireless device.

The following lists describe the API actions that perform the tasks associated with adding, updating, or deleting a service profile.

AWS IoT Wireless API actions for service profiles

- [CreateDeviceProfile](#)
- [GetDeviceProfile](#)
- [ListDeviceProfiles](#)
- [UpdateDeviceProfile](#)
- [DeleteDeviceProfile](#)

For the complete list of the actions and data types available to create and manage AWS IoT Core for LoRaWAN resources, see the [AWS IoT Wireless API reference](#).

How to use the AWS CLI to create a device profile

You can use the AWS CLI to create a device profile by using the `create-device-profile` command. The following example creates a device profile.

```
aws iotwireless create-device-profile
```

Running this command automatically creates a device profile with an ID that you can use when creating the wireless device. You can now create the service profile using the following API and then create the wireless device by using the device and service profiles.

```
{  
    "Arn": "arn:aws:iotwireless:us-east-1:123456789012:DeviceProfile/12345678-a1b2-3c45-67d8-e90fa1b2c34d",  
    "Id": "12345678-a1b2-3c45-67d8-e90fa1b2c34d"  
}
```

For information about the CLIs that you can use, see [AWS CLI reference](#)

Add service profiles

Service profiles describe the communication parameters the device needs to communicate with the application server.

Add a service profile using the console

If you're adding a wireless device using the console as described in [Add your wireless device specification to AWS IoT Core for LoRaWAN using the console \(p. 1011\)](#), after you've added the device profile, you can add your service profile. Alternatively, you can also add wireless devices from the [Profiles](#) page of the AWS IoT console on the [LoRaWAN](#) tab.

We recommend that you leave the setting **AddGWMetaData** enabled so that you'll receive additional gateway metadata for each payload, such as RSSI and SNR for the data transmission.

Add a service profile using the API

If you're adding a wireless device using the API, you must first create your service profile before creating the wireless device.

The following lists describe the API actions that perform the tasks associated with adding, updating, or deleting a service profile.

AWS IoT Wireless API actions for service profiles

- [CreateServiceProfile](#)
- [GetServiceProfile](#)
- [ListServiceProfiles](#)
- [UpdateServiceProfile](#)
- [DeleteServiceProfile](#)

For the complete list of the actions and data types available to create and manage AWS IoT Core for LoRaWAN resources, see the [AWS IoT Wireless API reference](#).

How to use the AWS CLI to create a service profile

You can use the AWS CLI to create a service by using the [create-service-profile](#) command. The following example creates a service profile.

```
aws iotwireless create-service-profile
```

Running this command automatically creates a service profile with an ID that you can use when creating the wireless device. You can now create the wireless device by using the device and service profiles.

```
{  
    "Arn": "arn:aws:iotwireless:us-east-1:123456789012:ServiceProfile/12345678-a1b2-3c45-67d8-e90fa1b2c34d",  
    "Id": "12345678-a1b2-3c45-67d8-e90fa1b2c34d"  
}
```

Add destinations to AWS IoT Core for LoRaWAN

AWS IoT Core for LoRaWAN destinations describe the AWS IoT rule that processes a device's data for use by AWS services.

Because most LoRaWAN devices don't send data to AWS IoT Core for LoRaWAN in a format that can be used by AWS services, an AWS IoT rule must process it first. The AWS IoT rule contains the SQL statement that interprets the device's data and the topic rule actions that send the result of the SQL statement to the services that will use it.

If you're adding your destination for the first time, we recommend that you use the console.

Add a destination using the console

If you're adding a wireless device using the console as described in [Add your wireless device specification to AWS IoT Core for LoRaWAN using the console \(p. 1011\)](#), after you've already added the wireless device specification and profiles to AWS IoT Core for LoRaWAN as described previously, you can go ahead and add a destination.

Alternatively, you can also add an AWS IoT Core for LoRaWAN destination from the [Destinations](#) page of the AWS IoT console.

To process a device's data, specify the following fields when creating an AWS IoT Core for LoRaWAN destination, and then choose **Add destination**.

- **Destination details**

Enter a **Destination name** and an optional description for your destination.

- **Rule name**

The AWS IoT rule that is configured to evaluate messages sent by your device and process the device's data. The rule name will be mapped to your destination. The destination requires the rule to process the messages that it receives. You can choose for the messages to be processed by either invoking an AWS IoT rule or by publishing to the AWS IoT message broker.

- If you choose **Enter a rule name**, enter a name, and then choose **Copy** to copy the rule name that you'll enter when creating the AWS IoT rule. You can either choose **Create rule** to create the rule now or navigate to the [Rules Hub](#) of the AWS IoT console and create a rule with that name.

You can also enter a rule and use the **Advanced** setting to specify a topic name. The topic name is provided during rule invocation and is accessed by using the topic expression inside the rule. For more information about AWS IoT rules, see [Rules for AWS IoT \(p. 443\)](#).

- If you choose **Publish to AWS IoT message broker**, enter a topic name. You can then copy the MQTT topic name and multiple subscribers can subscribe to this topic to receive messages published to that topic. For more information, see [MQTT topics \(p. 93\)](#).

For more information about AWS IoT rules for destinations, see [Create rules to process LoRaWAN device messages \(p. 1018\)](#).

- **Role name**

The IAM role that grants the device's data permission to access the rule named in **Rule name**. In the console, you can create a new service role or select an existing service role. If you're creating a new service role, you can either enter a role name (for example, `IoTWirelessDestinationRole`), or leave it blank for AWS IoT Core for LoRaWAN to generate a new role name. AWS IoT Core for LoRaWAN will then automatically create the IAM role with the appropriate permissions on your behalf.

For more information about IAM roles, see [Using IAM roles](#).

Add a destination by using the API

If you want to add a destination using the CLI instead, you must have already created the rule and IAM role for your destination. For more information about the details that a destination requires in the role, see [Create an IAM role for your destinations \(p. 1016\)](#).

The following list contains the API actions that perform the tasks associated with adding, updating, or deleting a destination.

AWS IoT Wireless API actions for destinations

- [CreateDestination](#)
- [GetDestination](#)
- [ListDestinations](#)
- [UpdateDestination](#)
- [DeleteDestination](#)

For the complete list of the actions and data types available to create and manage AWS IoT Core for LoRaWAN resources, see the [AWS IoT Wireless API reference](#).

How to use the AWS CLI to add a destination

You can use the AWS CLI to add a destination by using the `create-destination` command. The following example shows how to create a destination by entering a rule name by using `RuleName` as the value for the `expression-type` parameter. If you want to specify a topic name for publishing or subscribing to the message broker, change the `expression-type` parameter's value to `MqttTopicId`.

```
aws iotwireless create-destination \
--name IoTWirelessDestination \
--expression-type RuleName \
--expression IoTWirelessRule \
--role-arn arn:aws:iam::123456789012:role/IoTWirelessDestinationRole
```

Running this command creates a destination with the specified destination name, rule name, and role name. For information about rule and role names for destinations, see [Create rules to process LoRaWAN device messages \(p. 1018\)](#) and [Create an IAM role for your destinations \(p. 1016\)](#).

For information about the CLIs that you can use, see [AWS CLI reference](#).

Create an IAM role for your destinations

AWS IoT Core for LoRaWAN destinations require IAM roles that give AWS IoT Core for LoRaWAN the permissions necessary to send data to the AWS IoT rule. If such a role is not already defined, you must define it so that it will appear in the list of roles.

When you use the console to add a destination, AWS IoT Core for LoRaWAN automatically creates an IAM role for you, as described previously in this topic. When you add a destination using the API or CLI, you must create the IAM role for your destination.

To create an IAM policy for your AWS IoT Core for LoRaWAN destination role

1. Open the [Policies hub of the IAM console](#).
2. Choose **Create policy**, and choose the **JSON** tab.
3. In the editor, delete any content from the editor and paste this policy document.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:DescribeEndpoint",  
                "iot:Publish"  
            ],  
            "Resource": "*"  
        }  
    ]  
}
```

4. Choose **Review policy**, and in **Name**, enter a name for this policy. You'll need this name to use in the next procedure.

You can also describe this policy in **Description**, if you want.

5. Choose **Create policy**.

To create an IAM role for an AWS IoT Core for LoRaWAN destination

1. Open the [Roles hub of the IAM console](#) and choose **Create role**.
2. In **Select type of trusted entity**, choose **Another AWS account**.
3. In **Account ID**, enter your AWS account ID, and then choose **Next: Permissions**.
4. In the search box, enter the name of the IAM policy that you created in the previous procedure.
5. In the search results, check the IAM policy that you created in the previous procedure.
6. Choose **Next: Tags**, and then choose **Next: Review**.
7. In **Role name**, enter the name of this role, and then choose **Create role**.
8. In the confirmation message, choose the name of the role you created to edit the new role.
9. In **Summary**, choose the **Trust relationships** tab, and then choose **Edit trust relationship**.
10. In **Policy Document**, change the **Principal** property to look like this example.

```
"Principal": {  
    "Service": "iotwireless.amazonaws.com"  
},
```

After you change the **Principal** property, the complete policy document should look like this example.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:DescribeEndpoint",  
                "iot:Publish"  
            ],  
            "Resource": "*"  
        }  
    ]  
}
```

```
"Principal": {  
    "Service": "iotwireless.amazonaws.com"  
},  
"Action": "sts:AssumeRole",  
"Condition": {}  
}  
]  
}
```

11. To save your changes and exit, choose **Update Trust Policy**.

With this role defined, you can find it in the list of roles when you configure your AWS IoT Core for LoRaWAN destinations.

Create rules to process LoRaWAN device messages

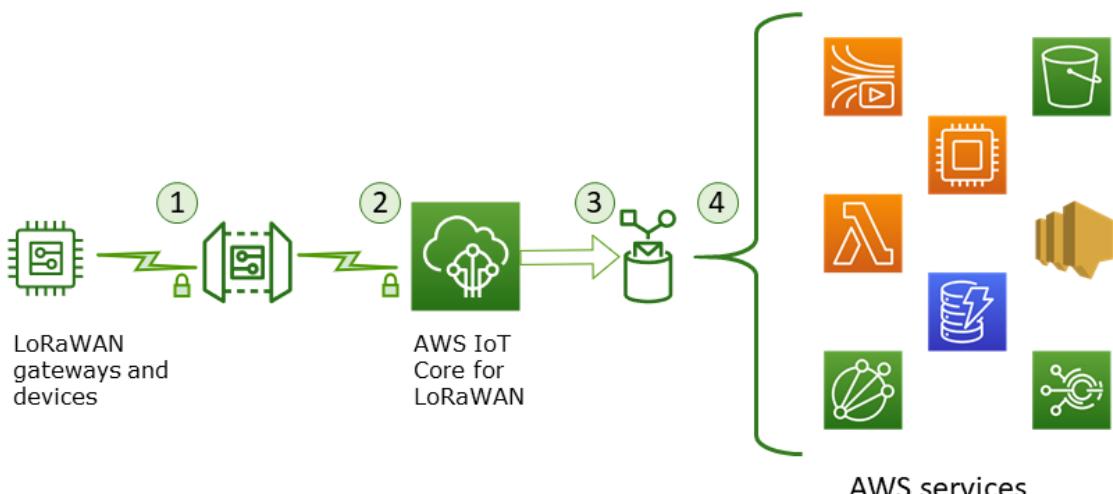
AWS IoT rules send device messages to other services. AWS IoT rules can also process the binary messages received from a LoRaWAN device to convert the messages to other formats that can make them easier for other services to use.

[AWS IoT Core for LoRaWAN destinations \(p. 1015\)](#) associate a wireless device with the rule that processes the device's message data to send to other services. The rule acts on the device's data as soon as AWS IoT Core for LoRaWAN receives it. [AWS IoT Core for LoRaWAN destinations \(p. 1015\)](#) can be shared by all devices whose messages have the same data format and that send their data to the same service.

How AWS IoT rules process device messages

How an AWS IoT rule processes a device's message data depends on the service that will receive the data, the format of the device's message data, and the data format that the service requires. Typically, the rule calls an AWS Lambda function to convert the device's message data to the format a service requires, and then sends the result to the service.

The following illustration shows how message data is secured and processed as it moves from the wireless device to an AWS service.



1. The LoRaWAN wireless device encrypts its binary messages using AES128 CTR mode before it transmits them.
2. AWS IoT Core for LoRaWAN decrypts the binary message and encodes the decrypted binary message payload as a base64 string.

3. The resulting base64-encoded message is sent as a binary message payload (a message payload that is not formatted as a JSON document) to the AWS IoT rule described in the destination assigned to the device.
4. The AWS IoT rule directs the message data to the service described in the rule's configuration.

The encrypted binary payload received from the wireless device is not altered or interpreted by AWS IoT Core for LoRaWAN. The decrypted binary message payload is encoded only as a base64 string. For services to access the data elements in the binary message payload, the data elements must be parsed out of the payload by a function called by the rule. The base64-encoded message payload is an ASCII string, so it could be stored as such to be parsed later.

Create rules for LoRaWAN

AWS IoT Core for LoRaWAN uses AWS IoT rules to securely send device messages directly to other AWS services without the need to use the message broker. By removing the message broker from the ingestion path, it reduces costs and optimizes the data flow.

For an AWS IoT Core for LoRaWAN rule to send device messages to other AWS services, it requires an AWS IoT Core for LoRaWAN destination and an AWS IoT rule assigned to that destination. The AWS IoT rule must contain a SQL query statement and at least one rule action.

Typically, the AWS IoT rule query statement consists of:

- A SQL SELECT clause that selects and formats the data from the message payload
- A topic filter (the FROM object in the rule query statement) that identifies the messages to use
- An optional conditional statement (a SQL WHERE clause) that specifies conditions on which to act

Here is an example of a rule query statement:

```
SELECT temperature FROM iot/topic' WHERE temperature > 50
```

When building AWS IoT rules to process payloads from LoRaWAN devices, you do not have to specify the FROM clause as part of the rule query object. The rule query statement must have the SQL SELECT clause and can optionally have the WHERE clause. If the query statement uses the FROM clause, it is ignored.

Here is an example of a rule query statement that can process payloads from LoRaWAN devices:

```
SELECT WirelessDeviceId, WirelessMetadata.LoRaWAN.FPort as FPort,  
WirelessMetadata.LoRaWAN.DevEui as DevEui,  
PayloadData
```

In this example, the PayloadData is a base64-encoded binary payload sent by your LoRaWAN device.

Here is an example rule query statement that can perform a binary decoding of the incoming payload and transform it into a different format such as JSON:

```
SELECT WirelessDeviceId, WirelessMetadata.LoRaWAN.FPort as FPort,  
WirelessMetadata.LoRaWAN.DevEui as DevEui,  
aws_lambda("arn:aws:lambda:<region>:<account>:function:<name>",<br>{<br>    "PayloadData":PayloadData,<br>    "Fport": WirelessMetadata.LoRaWAN.FPort
```

```
} ) as decodingoutput
```

For more information on using the SELECT AND WHERE clauses, see [AWS IoT SQL reference \(p. 529\)](#)

For information about AWS IoT rules and how to create and use them, see [Rules for AWS IoT \(p. 443\)](#) and [Creating AWS IoT rules to route device data to other services \(p. 182\)](#).

For information about creating and using AWS IoT Core for LoRaWAN destinations, see [Add destinations to AWS IoT Core for LoRaWAN \(p. 1015\)](#).

For information about using binary message payloads in a rule, see [Working with binary payloads \(p. 588\)](#).

For more information about the data security and encryption used to protect the message payload on its journey, see [Data protection in AWS IoT Core \(p. 361\)](#).

For a reference architecture that shows a binary decoding and implementation example for IoT rules, see [AWS IoT Core for LoRaWAN Solution Samples on GitHub](#).

Connect your LoRaWAN device and verify its connection status

Before you can check the device connection status, you must have already added your device and connected it to AWS IoT Core for LoRaWAN. For information about how to add your device, see [Add your wireless device to AWS IoT Core for LoRaWAN \(p. 1011\)](#).

After you've added your device, refer to your device's user manual to learn how to initiate sending an uplink message from your LoRaWAN device.

Check device connection status using the console

To check the connection status using the console, navigate to the **Devices** page of the AWS IoT console and choose the device you've added. In the **Details** section of the Wireless devices details page, you'll see the date and time the last uplink was received.

Check device connection status using the API

To check the connection status using the API, use the `GetWirelessDeviceStatistics` API. This API doesn't have a request body and only contains a response body that shows when the last uplink was received.

```
HTTP/1.1 200
Content-type: application/json

{
    "LastUplinkReceivedAt": "2021-03-24T23:13:08.476015749Z",
    "LoRaWAN": {
        "DataRate": 5,
        "DevEui": "647fda0000006420",
        "Frequency": 868100000
        "Gateways": [
            {
                "GatewayEui": "c0ee40fffff29df10",
                "Rssi": -67,
                "Snr": 9.75
            }
        ],
        "WirelessDeviceId": "30cbdcf3-86de-4291-bfab-5bfa2b12bad5"
    }
}
```

Next steps

Now that you have connected your device and verified the connection status, you can observe the format of the uplink metadata received from the device by using the [MQTT test client](#) on the **Test** page of the AWS IoT console. For more information, see [View format of uplink messages sent from LoRaWAN devices \(p. 1045\)](#).

Connecting to AWS IoT Core for LoRaWAN through a VPC interface endpoint

You can connect directly to AWS IoT Core for LoRaWAN through [Interface VPC endpoints \(AWS PrivateLink\)](#) in your Virtual Private Cloud (VPC) instead of connecting over the public internet. When you use a VPC interface endpoint, communication between your VPC and AWS IoT Core for LoRaWAN is conducted entirely and securely within the AWS network.

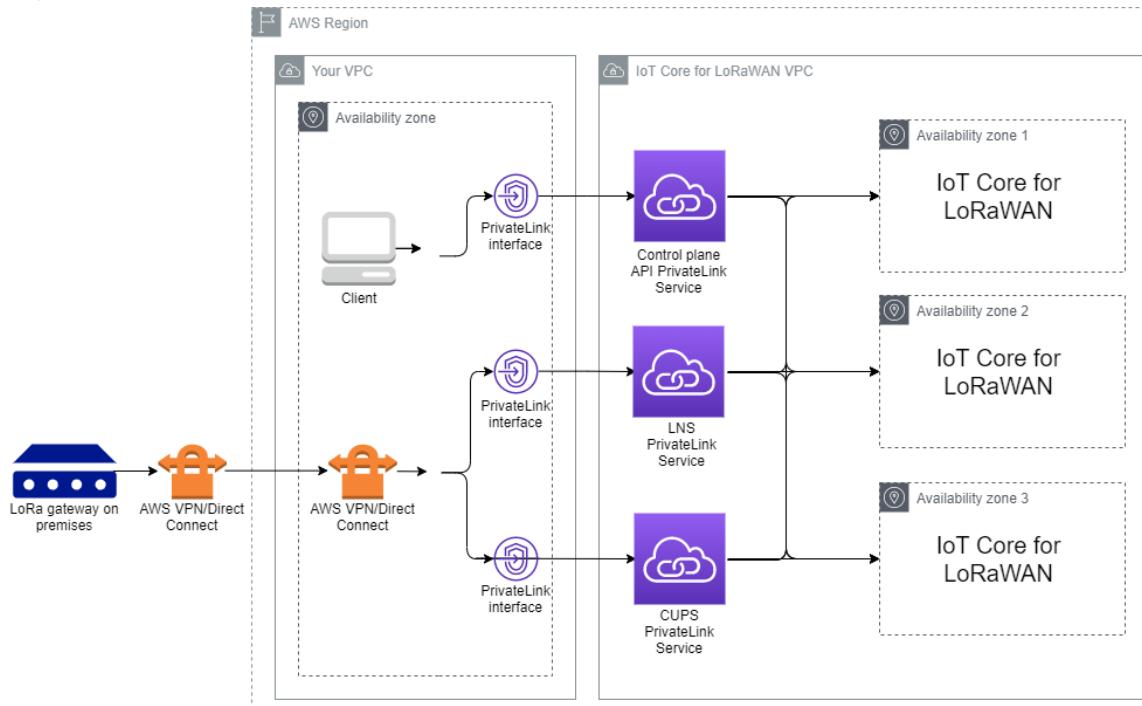
AWS IoT Core for LoRaWAN supports Amazon Virtual Private Cloud interface endpoints that are powered by AWS PrivateLink. Each VPC endpoint is represented by one or more Elastic Network Interfaces (ENIs) with private IP addresses in your VPC subnets.

For more information about VPC and endpoints, see [What is Amazon VPC](#).

For more information about AWS PrivateLink, see [AWS PrivateLink and VPC endpoints](#).

AWS IoT Core for LoRaWAN privatelink architecture

The following diagram shows the privatelink architecture of AWS IoT Core for LoRaWAN. The architecture uses a Transit Gateway and Route 53 Resolver to share the AWS PrivateLink interface endpoints between your VPC, the AWS IoT Core for LoRaWAN VPC, and an on-premises environment. You'll find a more detailed architecture diagram when setting up the connection to the VPC interface endpoints.



AWS IoT Core for LoRaWAN endpoints

AWS IoT Core for LoRaWAN has three public endpoints. Each public endpoint has a corresponding VPC interface endpoint. The public endpoints can be classified into control plane and data plane endpoints. For information about these endpoints, see [AWS IoT Core for LoRaWAN API endpoints](#).

Note

AWS PrivateLink support for the endpoints is available only in US East (N. Virginia) and Europe (Ireland).

- **Control plane API endpoints**

You can use control plane API endpoints to interact with the AWS IoT Wireless APIs. These endpoints can be accessed from a client that is hosted in your Amazon VPC by using AWS PrivateLink.

- **Data plane API endpoints**

Data plane API endpoints are LoRaWAN Network Server (LNS) and Configuration and Update Server (CUPS) endpoints that you can use to interact with the AWS IoT Core for LoRaWAN LNS and CUPS endpoints. These endpoints can be accessed from your LoRa gateways on premises by using AWS VPN or AWS Direct Connect. You get these endpoints when onboarding your gateway to AWS IoT Core for LoRaWAN. For more information, see [Add a gateway to AWS IoT Core for LoRaWAN \(p. 1006\)](#).

The following topics show how to onboard these endpoints.

Topics

- [Onboard AWS IoT Core for LoRaWAN control plane API endpoint \(p. 1022\)](#)
- [Onboard AWS IoT Core for LoRaWAN data plane API endpoints \(p. 1025\)](#)

Onboard AWS IoT Core for LoRaWAN control plane API endpoint

You can use AWS IoT Core for LoRaWAN control plane API endpoints to interact with the AWS IoT Wireless APIs. For example, you can use this endpoint to run the [SendDataToWirelessDevice](#) API to send data from AWS IoT to your LoRaWAN device. For more information, see [AWS IoT Core for LoRaWAN Control Plane API Endpoints](#).

You can use the client hosted in your Amazon VPC to access the control plane endpoints that are powered by AWS PrivateLink. You use these endpoints to connect to the AWS IoT Wireless API through an interface endpoint in your Virtual Private Cloud (VPC) instead of connecting over the public internet.

To onboard the control plane endpoint:

- [Create your Amazon VPC and subnet \(p. 1022\)](#)
- [Launch an Amazon EC2 instance in your subnet \(p. 1023\)](#)
- [Create Amazon VPC interface endpoint \(p. 1023\)](#)
- [Test your connection to the interface endpoint \(p. 1024\)](#)

Create your Amazon VPC and subnet

Before you can connect to the interface endpoint, you must create a VPC and subnet. You'll then launch an EC2 instance in your subnet, which you can use to connect to the interface endpoint.

To create your VPC:

1. Navigate to the [VPCs](#) page of the Amazon VPC console and choose **Create VPC**.
2. On the **Create VPC** page:
 - Enter a name for **VPC Name tag - optional** (for example, **VPC-A**).
 - Enter an IPv4 address range for your VPC in the **IPv4 CIDR block** (for example, **10.100.0.0/16**).
3. Keep the default values for other fields and choose **Create VPC**.

To create your subnet:

1. Navigate to the [Subnets](#) page of the Amazon VPC console and choose **Create subnet**.
2. On the **Create subnet** page:
 - For **VPC ID**, choose the VPC that you created earlier (for example, **VPC-A**).
 - Enter a name for **Subnet name** (for example, **Private subnet**).
 - Choose the **Availability Zone** for your subnet.
 - Enter your subnet's IP address block in the **IPv4 CIDR block** in CIDR format (for example, **10.100.0.0/24**).
3. To create your subnet and add it to your VPC, choose **Create subnet**.

For more information, see [Work with VPCs and subnets](#).

Launch an Amazon EC2 instance in your subnet

To launch your EC2 instance:

1. Navigate to the [Amazon EC2](#) console and choose **Launch Instance**.
2. For AMI, choose **Amazon Linux 2 AMI (HVM), SSD Volume Type** and then choose the **t2 micro** instance type. To configure the instance details, choose **Next**.
3. In the **Configure Instance Details** page:
 - For **Network**, choose the VPC that you created earlier (for example, **VPC-A**).
 - For **Subnet**, choose the subnet that you created earlier (for example, **Private subnet**).
 - For **IAM role**, choose the role **AWSIoTWirelessFullAccess** to grant AWS IoT Core for LoRaWAN full access policy. For more information, see [AWSIoTWirelessFullAccess policy summary](#).
 - For **Assume Private IP**, use an IP address, for example, **10.100.0.42**.
4. Choose **Next: Add Storage** and then choose **Next: Add Tags**. You can optionally add any tags to associate with your EC2 instance. Choose **Next: Configure Security Group**.
5. In the **Configure Security Group** page, configure the security group to allow:
 - Open **All TCP** for Source as **10.200.0.0/16**.
 - Open **All ICMP - IPV4** for Source as **10.200.0.0/16**.
6. To review the instance details and launch your EC2 instance, choose **Review and Launch**.

For more information, see [Get started with Amazon EC2 Linux instances](#).

Create Amazon VPC interface endpoint

You can create a VPC endpoint for your VPC, which can then be accessed by the EC2 API. To create the endpoint:

1. Navigate to the [VPC Endpoints](#) console and choose **Create Endpoint**.
2. In the **Create Endpoint** page, specify the following information.

- Choose **AWS services** for **Service category**.
- For **Service Name**, search by entering the keyword **iotwireless**. In the list of **iotwireless** services displayed, choose the control plane API endpoint for your Region. The endpoint will be in the format `com.amazonaws.region.iotwireless.api`.
- For **VPC** and **Subnets**, choose the VPC where you want to create the endpoint, and the Availability Zones (AZs) in which you want to create the endpoint network.

Note

The **iotwireless** service might not support all Availability Zones.

- For **Enable DNS name**, choose **Enable for this endpoint**.

Choosing this option will automatically resolve the DNS and create a route in Amazon Route 53 Public Data Plane so that the APIs you use later to test the connection will go through the **privatelink** endpoints.

- For **Security group**, choose the security groups you want to associate with the endpoint network interfaces.
- Optionally, you can add or remove tags. Tags are name-value pairs that you use to associate with your endpoint.

3. To create your VPC endpoint, choose **Create endpoint**.

Test your connection to the interface endpoint

You can use an SSH to access your Amazon EC2 instance and then use the AWS CLI to connect to the **privatelink** interface endpoints.

Before you connect to the interface endpoint, download the most recent AWS CLI version by following the instructions described in [Installing, updating, and uninstalling AWS CLI version 2 on Linux](#).

The following examples show how you can test your connection to the interface endpoint using the CLI.

```
aws iotwireless create-service-profile \
--endpoint-url https://api.iotwireless.region.amazonaws.com \
--name='test-privatelink'
```

The following shows an example of running the command.

```
Response:
{
  "Arn": "arn:aws:iotwireless:region:acct_number:ServiceProfile/1a2345ba-4c5d-67b0-ab67-e0c8342f2857",
  "Id": "1a2345ba-4c5d-67b0-ab67-e0c8342f2857"
}
```

Similarly, you can run the following commands to get the service profile information or list all service profiles.

```
aws iotwireless get-service-profile \
--endpoint-url https://api.iotwireless.region.amazonaws.com
--id="1a2345ba-4c5d-67b0-ab67-e0c8342f2857"
```

The following shows an example for the `list-device-profiles` command.

```
aws iotwireless list-device-profiles \
```

```
--endpoint-url https://api.iotwireless.region.amazonaws.com
```

Onboard AWS IoT Core for LoRaWAN data plane API endpoints

AWS IoT Core for LoRaWAN data plane endpoints consist of the following endpoints. You get these endpoints when adding your gateway to AWS IoT Core for LoRaWAN. For more information, see [Add a gateway to AWS IoT Core for LoRaWAN \(p. 1006\)](#).

- **LoRaWAN Network Server (LNS) endpoints**

The LNS endpoints are of the format *account-specific-prefix*.lns.lorawan.*region*.amazonaws.com. You can use this endpoint to establish a connection for exchanging LoRa uplink and downlink messages.

- **Configuration and Update Server (CUPS) endpoints**

The CUPS endpoints are of the format *account-specific-prefix*.cups.lorawan.*region*.amazonaws.com. You can use this endpoint for credentials management, remote configuration, and firmware update of gateways.

For more information, see [Using CUPS and LNS protocols \(p. 1031\)](#).

To find the Data Plane API endpoints for your AWS account and Region, use the [get-service-endpoint](#) CLI command shown here, or the [GetServiceEndpoint](#) REST API. For more information, see [AWS IoT Core for LoRaWAN Data Plane API Endpoints](#).

You can connect your LoRaWAN gateway on premises to communicate with AWS IoT Core for LoRaWAN endpoints. To establish this connection, first connect your on premises gateway to your AWS account in your VPC by using a VPN connection. You can then communicate with the data plane interface endpoints in the AWS IoT Core for LoRaWAN VPC that are powered by privatelink.

The following shows how to onboard these endpoints.

- [Create VPC interface endpoint and private hosted zone \(p. 1025\)](#)
- [Use VPN to connect LoRa gateways to your AWS account \(p. 1028\)](#)

Create VPC interface endpoint and private hosted zone

AWS IoT Core for LoRaWAN has two data plane endpoints, Configuration and Update Server (CUPS) endpoint and LoRaWAN Network Server (LNS) endpoint. The setup process to establish a privatelink connection to both endpoints is the same, so we can use the LNS endpoint for illustration purposes.

For your data plane endpoints, the LoRa gateways first connect to your AWS account in your Amazon VPC, which then connects to the VPC endpoint in the AWS IoT Core for LoRaWAN VPC.

When connecting to the endpoints, the DNS names can be resolved within one VPC but can't be resolved across multiple VPCs. To disable private DNS when creating the endpoint, disable the **Enable DNS name** setting. You can use private hosted zone to provide information about how you want Route 53 to respond to DNS queries for your VPCs. To share your VPC with an on-premises environment, you can use a Route 53 Resolver to facilitate hybrid DNS.

To complete this procedure, perform the following steps.

- [Create an Amazon VPC and subnet \(p. 1026\)](#)
- [Create an Amazon VPC interface endpoint \(p. 1026\)](#)

- [Configure private hosted zone \(p. 1026\)](#)
- [Configure Route 53 inbound resolver \(p. 1027\)](#)
- [Next steps \(p. 1028\)](#)

Create an Amazon VPC and subnet

You can reuse your Amazon VPC and subnet that you created when onboarding your control plane endpoint. For information, see [Create your Amazon VPC and subnet \(p. 1022\)](#).

Create an Amazon VPC interface endpoint

You can create a VPC endpoint for your VPC, which is similar to how you would create one for your control plane endpoint.

1. Navigate to the [VPC Endpoints](#) console and choose **Create Endpoint**.
2. In the **Create Endpoint** page, specify the following information.
 - Choose **AWS services** for **Service category**.
 - For **Service Name**, search by entering the keyword `lns`. In the list of `lns` services displayed, choose the LNS data plane API endpoint for your Region. The endpoint will be of the format `com.amazonaws.region.lorawan.lns`.

Note

If you're following this procedure for your CUPS endpoint, search for `cups`. The endpoint will be of the format `com.amazonaws.region.lorawan.cups`.

3. For **VPC and Subnets**, choose the VPC where you want to create the endpoint, and the Availability Zones (AZs) in which you want to create the endpoint network.

Note

The `iotwireless` service might not support all Availability Zones.

4. For **Enable DNS name**, make sure that **Enable for this endpoint** is not selected.

By not selecting this option, you can disable private DNS for the VPC endpoint and use private hosted zone instead.

5. For **Security group**, choose the security groups you want to associate with the endpoint network interfaces.
6. Optionally, you can add or remove tags. Tags are name-value pairs that you use to associate with your endpoint.

7. To create your VPC endpoint, choose **Create endpoint**.

Configure private hosted zone

After you create the privatelink endpoint, in the **Details** tab of your endpoint, you'll see a list of DNS names. You can use one of these DNS names to configure your private hosted zone. The DNS name will be of the format `vpce-xxxx.lns.lorawan.region.vpce.amazonaws.com`.

Create the private hosted zone

To create the private hosted zone:

1. Navigate to the [Route 53 Hosted zones](#) console and choose **Create hosted zone**.
2. In the **Create hosted zone** page, specify the following information.
 - For **Domain name**, enter the full service name for your LNS endpoint, `lns.lorawan.region.amazonaws.com`.

Note

If you're following this procedure for your CUPS endpoint, enter `cups.lorawan.region.amazonaws.com`.

- For **Type**, choose **Private hosted zone**.
 - Optionally, you can add or remove tags to associate with your hosted zone.
3. To create your private hosted zone, choose **Create hosted zone**.

For more information, see [Creating a private hosted zone](#).

After you have created a private hosted zone, you can create a record that tells the DNS how you want traffic to be routed to that domain.

Create a record

After you have created a private hosted zone, you can create a record that tells the DNS how you want traffic to be routed to that domain. To create a record:

1. In the list of hosted zones displayed, choose the private hosted zone that you created earlier and choose **Create record**.
2. Use the wizard method to create the record. If the console presents you the **Quick create** method, choose **Switch to wizard**.
3. Choose **Simple Routing** for **Routing policy** and then choose **Next**.
4. In the **Configure records** page, choose **Define simple record**.
5. In the **Define simple record** page:
 - For **Record name**, enter the alias of your AWS account number. You get this value when onboarding your gateway or by using the [GetServiceEndpoint](#) REST API.
 - For **Record type**, keep the value as **A – Routes traffic to an IPv4 address and some AWS resources**.
 - For **Value/Route traffic to**, choose **Alias to VPC endpoint**. Then choose your **Region** and then choose the endpoint that you created previously, as described in [Create an Amazon VPC interface endpoint \(p. 1026\)](#) from the list of endpoints displayed.
6. Choose **Define simple record** to create your record.

Configure Route 53 inbound resolver

To share a VPC endpoint to an on-premises environment, a Route 53 Resolver can be used to facilitate hybrid DNS. The inbound resolver will enable you to route traffic from the on-premises network to the data plane endpoints without going over the public internet. To return the private IP address values for your service, create the Route 53 Resolver in the same VPC as the VPC endpoint.

When you create the inbound resolver, you only have to specify your VPC and the subnets that you created previously in your Availability Zones (AZs). The Route 53 Resolver uses this information to automatically assigns an IP address to route traffic to each of the subnets.

To create the inbound resolver:

1. Navigate to the [Route 53 Inbound endpoints](#) console and choose **Create inbound endpoint**.

Note

Make sure that you're using the same AWS Region that you used when creating the endpoint and private hosted zone.

2. In the **Create inbound endpoint** page, specify the following information.
 - Enter a name for **Endpoint name** (for example, **VPC_A_Test**).

- For **VPC in the region**, choose the same VPC that you used when creating the VPC endpoint.
 - Configure the **Security group for this endpoint** to allow incoming traffic from the on premises network.
 - For IP address, choose **Use an IP address that is selected automatically**.
3. Choose **Submit** to create your inbound resolver.

For this example, let's assume that the IP addresses 10.100.0.145 and 10.100.192.10 were assigned for the inbound Route 53 Resolver for routing traffic.

Next steps

You've created the private hosted zone and an inbound resolver to route traffic for your DNS entries. You can now use either a Site-to-Site VPN or a Client VPN endpoint. For more information, see [Use VPN to connect LoRa gateways to your AWS account \(p. 1028\)](#).

Use VPN to connect LoRa gateways to your AWS account

To connect your gateways on premises to your AWS account, you can use either a Site-to-Site VPN connection or a Client VPN endpoint.

Before you can connect your on premises gateways, you must have created the VPC endpoint, and configured a private hosted zone and inbound resolver so that traffic from the gateways don't go over the public internet. For more information, see [Create VPC interface endpoint and private hosted zone \(p. 1025\)](#).

Site-to-Site VPN endpoint

If you don't have the gateway hardware or want to test the VPN connection using a different AWS account, you can use a Site-to-Site VPN connection. You can use Site-to-Site VPN to connect to the VPC endpoints from the same AWS account or another AWS account that you might be using in a different AWS Region.

Note

If you've the gateway hardware with you and want to set up a VPN connection, we recommend that you use Client VPN instead. For instructions, see [Client VPN endpoint \(p. 1029\)](#).

To set up a Site-to-Site VPN:

1. Create another VPC in the site from which you want to set up the connection. For VPC-A, you can reuse the VPC that you created previously. To create another VPC (for example, VPC-B), use a CIDR block that doesn't overlap with the CIDR block of the VPC you created previously.

For information about setting up the VPCs, follow the instructions described in [AWS setup Site-to-Site VPN connection](#).

Note

The Site-to-Site VPN VPN method described in the document uses OpenSWAN for the VPN connection, which supports only one VPN tunnel. If you use a different commercial software for the VPN, you might be able to set up two tunnels between the sites.

2. After you set up the VPN connection, update the /etc/resolv.conf file by adding the inbound resolver's IP address from your AWS account. You use this IP address for the nameserver. For information about how to obtain this IP address, see [Configure Route 53 inbound resolver \(p. 1027\)](#). For this example, we can use the IP address 10.100.0.145 that was assigned when you created the Route 53 Resolver.

```
options timeout:2 attempts:5
```

```
; generated by /usr/sbin/dhclient-script
search region.compute.internal
nameserver 10.100.0.145
```

3. We can now test whether the VPN connection uses the AWS PrivateLink endpoint instead of going over the public internet by using an nslookup command. The following shows an example of running the command.

```
nslookup account-specific-prefix.lns.lorawan.region.amazonaws.com
```

The following shows an example output of running the command, which shows a private IP address indicating that the connection has been established to the AWS PrivateLink LNS endpoint.

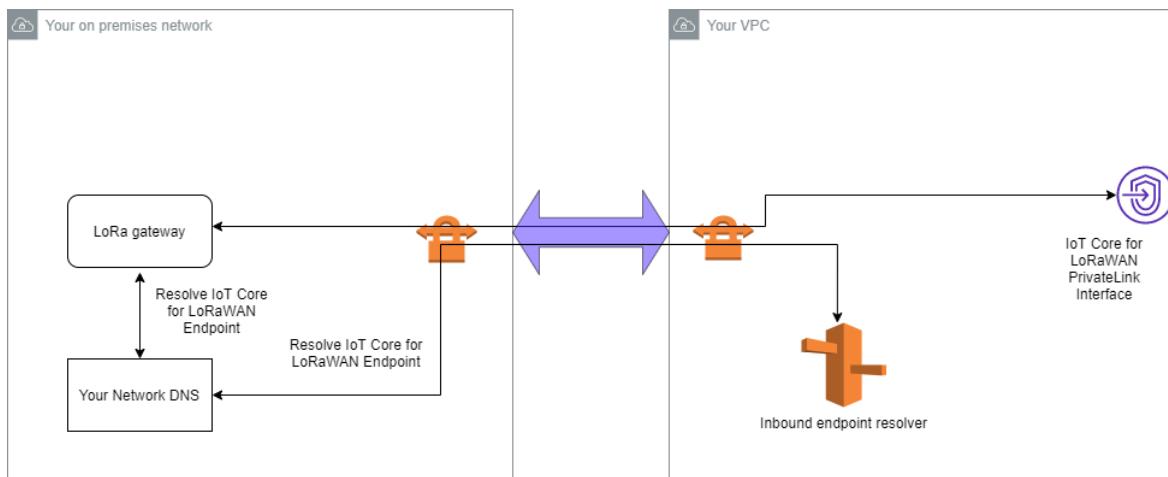
```
Server: 10.100.0.145
Address: 10.100.0.145

Non-authoritative answer:
Name: https://xxxxx.lns.lorawan.region.amazonaws.com
Address: 10.100.0.204
```

For information about using a Site-to-Site VPN connection, see [How Site-to-Site VPN works](#).

Client VPN endpoint

AWS Client VPN is a managed client-based VPN service that enables you to securely access AWS resources and resources in your on-premises network. The following shows the architecture for the client VPN service.



To establish a VPN connection to a Client VPN endpoint:

1. Create a Client VPN endpoint by following the instructions described in [Getting started with AWS Client VPN](#).
2. Log in to your on-premises network (for example, a Wi-Fi router) by using the access URL for that router (for example, 192.168.1.1), and find the root name and password.
3. Set up your LoRaWAN gateway by following the instructions in the gateway's documentation and then add your gateway to AWS IoT Core for LoRaWAN. For information about how to add your gateway, see [Onboard your gateways to AWS IoT Core for LoRaWAN \(p. 1004\)](#).
4. Check whether your gateway's firmware is up to date. If the firmware is out of date, you can follow the instructions provided in the on-premises network to update your gateway's firmware.

For more information, see [Update gateway firmware using CUPS service with AWS IoT Core for LoRaWAN \(p. 1033\)](#).

5. Check whether OpenVPN has been enabled. If it has been enabled, skip to the next step to configure the OpenVPN client inside the on-premises network. If it hasn't been enabled, follow the instructions in [Guide to install OpenVPN for OpenWrt](#).

Note

For this example, we use OpenVPN. You can use other VPN clients such as AWS VPN or AWS Direct Connect to set up your Client VPN connection.

6. Configure the OpenVPN client based on information from the client configuration and how you can use [OpenVPN client using LuCi](#).
7. SSH to your on-premises network and update the `/etc/resolv.conf` file by adding the IP address of the inbound resolver in your AWS account (10.100.0.145).
8. For the gateway traffic to use AWS PrivateLink to connect to the endpoint, replace the first DNS entry for your gateway to the inbound resolver's IP address.

For information about using a Site-to-Site VPN connection, see [Getting started with Client VPN](#).

Connect to LNS and CUPS VPC endpoints

The following shows how you can test your connection to the LNS and CUPS VPC endpoints.

Test CUPS endpoint

To test your AWS PrivateLink connection to the CUPS endpoint from your LoRa gateway, run the following command:

```
curl -k -v -X POST https://xxxx.cups.region.iotwireless.iot:443/update-info
    --cacert cups.trust --cert cups.crt --key cups.key --header "Content-Type:
application/json"
    --data '{
        "router": "xxxxxxxxxxxxxx",
        "cupsUri": "https://xxxx.cups.lorawan.region.amazonaws.com:443",
        "cupsCredCrc":1234, "tcCredCrc":552384314
    }'
    -output cups.out
```

Test LNS endpoint

To test your LNS endpoint, first provision a LoRaWAN device that will work with your wireless gateway. You can then add your device and perform the *join* procedure after which you can start sending uplink messages.

Managing gateways with AWS IoT Core for LoRaWAN

Gateways act as a bridge and carry LoRaWAN device data to and from a Network Server, usually over high-bandwidth networks like Wi-Fi, Ethernet, or Cellular. LoRaWAN gateways connect wireless devices to AWS IoT Core for LoRaWAN.

Following are some important considerations when using your gateways with AWS IoT Core for LoRaWAN. For information about how to add your gateway to AWS IoT Core for LoRaWAN, see [Onboard your gateways to AWS IoT Core for LoRaWAN \(p. 1004\)](#).

LoRa Basics Station software requirement

To connect to AWS IoT Core for LoRaWAN, your LoRaWAN gateway must have software called [LoRa Basics Station](#) running on it. LoRa Basics Station is an open source software that is maintained by Semtech Corporation and distributed by their [GitHub](#) repository. AWS IoT Core for LoRaWAN supports LoRa Basics Station version 2.0.4 and later.

Using qualified gateways from the AWS Partner Device Catalog

The [AWS Partner Device Catalog](#) contains gateways and developer kits that are qualified for use with AWS IoT Core for LoRaWAN. We recommend that you use these qualified gateways because you don't have to modify the embedding software for connecting the gateways to AWS IoT Core. These gateways already have a version of the BasicStation software compatible with AWS IoT Core for LoRaWAN.

Note

If you have a gateway that is not listed in the Partner Catalog as a qualified gateway with AWS IoT Core for LoRaWAN, you might still be able to use it if the gateway is running LoRa Basics Station software with version 2.0.4 and later. Make sure that you use **TLS Server and Client Authentication** for authenticating your LoRaWAN gateway.

Using CUPS and LNS protocols

LoRa Basics Station software contains two sub protocols for connecting gateways to network servers, LoRaWAN Network Server (LNS) and Configuration and Update Server (CUPS) protocols.

The LNS protocol establishes a data connection between a LoRa Basics Station compatible gateway and a network server. LoRa uplink and downlink messages are exchanged through this data connection over secure WebSockets.

The CUPS protocol enables credentials management, and remote configuration and firmware update of gateways. AWS IoT Core for LoRaWAN provides both LNS and CUPS endpoints for LoRaWAN data ingestion and remote gateway management respectively.

For more information, see [LNS protocol](#) and [CUPS protocol](#).

Configure your gateway's subbands and filtering capabilities

LoRaWAN gateways run a [LoRa Basics Station](#) software that enables the gateways to connect to AWS IoT Core for LoRaWAN. To connect to AWS IoT Core for LoRaWAN, your LoRa gateway first queries the CUPS server for the LNS endpoint, and then establishes a WebSockets data connection with that endpoint. After the connection is established, uplink and downlink frames can be exchanged through that connection.

Filtering of LoRa data frames received by gateway

After your LoRaWAN gateway establishes a connection to the endpoint, AWS IoT Core for LoRaWAN responds with a `router_config` message that specifies a set of parameters for the LoRa gateway's configuration, including filtering parameters `NetID` and `JoinEui`. For more information about `router_config` and how a connection is established with the LoRaWAN Network Server (LNS), see [LNS protocol](#).

{

```
"msgtype"      : "router_config"
"NetID"        : [ INT, .. ]
"JoinEui"      : [ [INT,INT], .. ] // ranges: beg,end inclusive
"region"       : STRING          // e.g. "EU863", "US902", ..
"hwspec"       : STRING
"freq_range"   : [ INT, INT ]     // min, max (hz)
"DRs"          : [ [INT,INT,INT], .. ] // sf,bw,dnonly
"sx1301_conf": [ SX1301CONF, .. ]
"nocca"        : BOOL
"nodec"        : BOOL
"nodwell"      : BOOL
}
```

The gateways carry LoRaWAN device data to and from LNS usually over high-bandwidth networks like Wi-Fi, Ethernet, or Cellular. The gateways usually pick up all messages and pass through the traffic that comes to it to AWS IoT Core for LoRaWAN. However, you can configure the gateways to filter some of the device data traffic, which helps conserve bandwidth usage and reduces the traffic flow between the gateway and LNS.

To configure your LoRa gateway to filter the data frames, you can use the parameters `NetID` and `JoinEui` in the `router_config` message. `NetID` is a list of NetID values that are accepted. Any LoRa data frame carrying a data frame other than those listed will be dropped. `JoinEui` is a list of pairs of integer values encoding ranges of JoinEUI values. Join request frames will be dropped by the gateway unless the field `JoinEui` in the message is within the range [BegEui,EndEui].

Frequency channels and subbands

For US915 and AU915 RF regions, wireless devices have choices of 64 125KHz and 8 500KHz uplink channels to access the LoRaWAN networks using the LoRa gateways. The uplink frequency channels are divided into 8 subbands, each with 8 125KHz channels and one 500KHz channel. For each regular gateway in AU915 region, one or more subbands will be supported.

Some wireless devices can't hop between subbands and use the frequency channels in only one subband when connected to AWS IoT Core for LoRaWAN. For the uplink packets from those devices to be transmitted, configure the LoRa gateways to use that particular subband. For gateways in other RF regions, such as EU868, this configuration is not required.

Configure your gateway to use filtering and subbands using the console

You can configure your gateway to use a particular subband and also enable the capability to filter the LoRa data frames. To specify these parameters using the console:

1. Navigate to the [AWS IoT Core for LoRaWAN Gateways](#) page of the AWS IoT console and choose **Add gateway**.
2. Specify the gateway details such as the **Gateway's Eui**, **Frequency band (RFRegion)** and an optional **Name** and **Description**, and choose whether to associate an AWS IoT thing to your gateway. For more information about how to add a gateway, see [Add a gateway using the console \(p. 1007\)](#).
3. In the **LoRaWAN configuration** section, you can specify the subbands and filtering information.
 - **SubBands**: To add a subband, choose **Add SubBand** and specify a list of integer values that indicate which subbands are supported by the gateway. The `SubBands` parameter can only be configured in the `RfRegion` `US915` and `AU915` and must have values in the range `[1, 8]` within one of these supported regions.
 - **NetIdFilters**: To filter uplink frames, choose **Add NetId** and specify a list of string values that the gateway uses. The `NetID` of the incoming uplink frame from the wireless device must match at least one of the listed values, otherwise the frame is dropped.

- **JoinEuiFilters:** Choose **Add JoinEui range** and specify a list of pairs of string values that a gateway uses to filter LoRa frames. The JoinEUI value specified as part of the join request from the wireless device must be within the range of at least one of the JoinEuiRange values, each listed as a pair of [BegEui, EndEui], otherwise the frame is dropped.
4. You can then continue to configure your gateway by following the instructions described in [Add a gateway using the console \(p. 1007\)](#).

After you've added a gateway, in the [AWS IoT Core for LoRaWAN Gateways](#) page of the AWS IoT console, if you select the gateway that you've added, you can see the SubBands and filters NetIdFilters and JoinEuiFilters in the **LoRaWAN specific details** section of the Gateway details page.

Configure your gateway to use filtering and subbands using the API

You can use the [CreateWirelessGateway](#) API that you use to create a gateway to configure the subbands you want to use and enable the filtering capability. Using the [CreateWirelessGateway](#) API, you can specify the subbands and filters as part of the gateway configuration information that you provide using the **LoRaWAN** field. The following shows the request token that includes this information.

```
POST /wireless-gateways HTTP/1.1
Content-type: application/json

{
  "Arn": "arn:aws:iotwireless:us-east-1:400232685877aa:WirelessGateway/
    a11e3d21-e44c-471c-afca-6716c228336a",
  "Description": "Using my first LoRaWAN gateway",
  "LoRaWAN": {
    "GatewayEui": "a1b2c3d4567890ab",
    "JoinEuiFilters": [
      ["0000000000000001", "00000000000000ff"],
      ["000000000000ff00", "000000000000ffff"]
    ],
    "NetIdFilters": ["000000", "000001"],
    "RfRegion": "US915",
    "SubBands": [2]
  },
  "Name": "myFirstLoRaWANGateway"
  "ThingArn": null,
  "ThingName": null
}
```

You can also use the [UpdateWirelessGateway](#) API to update the filters but not the subbands. If the **JoinEuiFilters** and **NetIdfilters** values are null, it means there is no update for the fields. If the values aren't null and empty lists are included, then the update is applied. To get the values of the fields that you specified, use the [GetWirelessGateway](#) API.

Update gateway firmware using CUPS service with AWS IoT Core for LoRaWAN

The [LoRa Basics Station](#) software that runs on your gateway provides credential management and firmware update interface using the Configuration and Update Server (CUPS) protocol. The CUPS protocol provides secure firmware update delivery with ECDSA signatures.

You'll have to frequently update your gateway's firmware. You can use the CUPS service with AWS IoT Core for LoRaWAN to provide firmware updates to the gateway where the updates can also be signed. To update the gateway's firmware, you can use the SDK or CLI but not the console.

The update process takes about 45 minutes to complete. It can take longer if you're setting up your gateway for the first time to connect to AWS IoT Core for LoRaWAN. Gateway manufacturers usually provide their own firmware update files and signatures so you can use that instead and proceed to [Upload the firmware file to an S3 bucket and add an IAM role \(p. 1037\)](#).

If you don't have the firmware update files, see [Generate the firmware update file and signature \(p. 1034\)](#) for an example that you can use to adapt to your application.

To perform your gateway's firmware update:

- [Generate the firmware update file and signature \(p. 1034\)](#)
- [Upload the firmware file to an S3 bucket and add an IAM role \(p. 1037\)](#)
- [Schedule and run the firmware update by using a task definition \(p. 1040\)](#)

Generate the firmware update file and signature

The steps in this procedure are optional and depend on the gateway you're using. Gateway manufacturers provide their own firmware update in the form of an update file or a script and Basics Station runs this script in the background. In this case, you'll most likely find the firmware update file in the release notes of the gateway you're using. You can then use that update file or script instead and proceed to [Upload the firmware file to an S3 bucket and add an IAM role \(p. 1037\)](#).

If you don't have this script, following shows the commands to run for generating the firmware update file. The updates can also be signed to ensure that the code was not altered or corrupted and devices run code published only by trusted authors.

In this procedure, you'll:

- [Generate the firmware update file \(p. 1034\)](#)
- [Generate signature for the firmware update \(p. 1036\)](#)
- [Review the next steps \(p. 1037\)](#)

Generate the firmware update file

The LoRa Basics Station software running on the gateway is capable of receiving firmware updates in the CUPS response. If you don't have a script provided by the manufacturer, refer to the following firmware update script that is written for the Raspberry Pi based RAKWireless Gateway. We have a base script and the new station binary, version file, and station.conf are attached to it.

Note

The script is specific to the RAKWireless Gateway, so you'll have to adapt it to your application depending on the gateway you're using.

Base script

Following shows a sample base script for the Raspberry Pi based RAKWireless Gateway. You can save the following commands in a file base.sh and then run the script in the terminal on the Raspberry Pi's web browser.

```
*#!/bin/bash*
execution_folder=/home/pi/Documents/basicstation/examples/aws_lorawan
station_path="$execution_folder/station"
version_path="$execution_folder/version.txt"
station_conf_path="$execution_folder/station.conf"

# Function to find the Basics Station binary at the end of this script
# and store it in the station path
function prepare_station()
```

```

{
    match=$(grep --text --line-number '^STATION:' $0 | cut -d ':' -f 1)
    payload_start=$((match + 1))
    match_end=$(grep --text --line-number '^END_STATION:' $0 | cut -d ':' -f 1)
    payload_end=$((match_end - 1))
    lines=$((payload_end-$payload_start+1))
    head -n $payload_end $0 | tail -n $lines > $station_path
}

# Function to find the version.txt at the end of this script
# and store it in the location for version.txt
function prepare_version()
{
    match=$(grep --text --line-number '^VERSION:' $0 | cut -d ':' -f 1)
    payload_start=$((match + 1))
    match_end=$(grep --text --line-number '^END_VERSION:' $0 | cut -d ':' -f 1)
    payload_end=$((match_end - 1))
    lines=$((payload_end-$payload_start+1))
    head -n $payload_end $0 | tail -n $lines > $version_path
}

# Function to find the version.txt at the end of this script
# and store it in the location for version.txt
function prepare_station_conf()
{
    match=$(grep --text --line-number '^CONF:' $0 | cut -d ':' -f 1)
    payload_start=$((match + 1))
    match_end=$(grep --text --line-number '^END_CONF:' $0 | cut -d ':' -f 1)
    payload_end=$((match_end - 1))
    lines=$((payload_end-$payload_start+1))
    head -n $payload_end $0 | tail -n $lines > $station_conf_path
}

# Stop the currently running Basics station so that it can be overwritten
# by the new one
killall station

# Store the different files
prepare_station
prepare_version
prepare_station_conf

# Provide execute permission for Basics station binary
chmod +x $station_path

# Remove update.bin so that it is not read again next time Basics station starts
rm -f /tmp/update.bin

# Exit so that rest of this script which has binaries attached does not get executed
exit 0

```

Add payload script

To the base script, we append the Basics Station binary, the version.txt that identifies the version to update to, and station.conf in a script called addpayload.sh. Then, run this script.

```

#!/bin/bash
*
base.sh > fwstation

# Add station
echo "STATION:" >> fwstation
cat $1 >> fwstation
echo "" >> fwstation

```

```
echo "END_STATION:" >> fwstation

# Add version.txt
echo "VERSION:" >> fwstation
cat $2 >> fwstation
echo "" >> fwstation
echo "END_VERSION:" >> fwstation

# Add station.conf
echo "CONF:" >> fwstation
cat $3 >> fwstation
echo "END_CONF:" >> fwstation

# executable
chmod +x fwstation
```

After you've run these scripts, you can run the following command in the terminal to generate the firmware update file, fwstation.

```
$ ./addpayload.sh station version.txt station.conf
```

Generate signature for the firmware update

The LoRa Basics Station software provides signed firmware updates with ECDSA signatures. To support signed updates, you'll need:

- A signature that must be generated by an ECDSA private key and less than 128 bytes.
- The private key that is used for the signature and must be stored in the gateway with file name of the format sig-%d.key. We recommend using the file name sig-0.key.
- A 32-bit CRC over the private key.

The signature and CRC will be passed to the AWS IoT Core for LoRaWAN APIs. To generate the previous files, you can use the following script gen.sh that is inspired by the [basicstation](#) example in the GitHub repository.

```
*#!/bin/bash

*function ecdsaKey() {
    # Key not password protected for simplicity
    openssl ecparam -name prime256v1 -genkey | openssl ec -out $1
}

# Generate ECDSA key
ecdsaKey sig-0.prime256v1.pem

# Generate public key
openssl ec -in sig-0.prime256v1.pem -pubout -out sig-0.prime256v1.pub

# Generate signature private key
openssl ec -in sig-0.prime256v1.pub -inform PEM -outform DER -pubin | tail -c 64 >
sig-0.key

# Generate signature
openssl dgst -sha512 -sign sig-0.prime256v1.pem $1 > sig-0.signature

# Convert signature to base64
openssl enc -base64 -in sig-0.signature -out sig-0.signature.base64

# Print the crc
crc_res=$(crc32 sig-0.key)printf "The crc for the private key=%d\n" $((16#$crc_res))
```

```
# Remove the generated files which won't be needed later
rm -rf sig-0.prime256v1.pem sig-0.signature sig-0.prime256v1.pub
```

The private key generated by the script should be saved into the gateway. The key file is in binary format.

```
./gen_sig.sh fwstation
read EC key
writing EC key
read EC key
writing EC key
read EC key
writing EC key
The crc for the private key=3434210794

$ cat sig-0.signature.base64
MEQCIDPY/p2ssgXIPNC0gZr+NzeTLpX+WfBo5tYWh5pQWN3AiBROen+XlIdMScv
AsfVfU/ZScJCalVNZh4esyS8mNIgA==

$ ls sig-0.key
sig-0.key

$ scp sig-0.key pi@192.168.1.11:/home/pi/Documents/basicstation/examples/iotwireless
```

Review the next steps

Now that you have generated the firmware and signature, go to the next topic to upload the firmware file, `fwstation`, to an Amazon S3 bucket. The bucket is a container that will store the firmware update file as an object. You can add an IAM role that will give the CUPS server permission to read the firmware update file in the S3 bucket.

Upload the firmware file to an S3 bucket and add an IAM role

You can use Amazon S3 to create a *bucket*, which is a container that can store your firmware update file. You can upload your file to the S3 bucket and add an IAM role that allows the CUPS server to read your update file from the bucket. For more information about Amazon S3, see [Getting started with Amazon S3](#).

The firmware update file that you want to upload depends on the gateway you're using. If you followed a procedure similar to the one described in [Generate the firmware update file and signature \(p. 1034\)](#), you'll upload the `fwstation` file generated by running the scripts.

This procedure takes about 20 minutes to complete.

To upload your firmware file:

- [Create an Amazon S3 bucket and upload the update file \(p. 1037\)](#)
- [Create an IAM role with permissions to read the S3 bucket \(p. 1038\)](#)
- [Review the next steps \(p. 1040\)](#)

Create an Amazon S3 bucket and upload the update file

You'll create an Amazon S3 bucket by using the AWS Management Console and then upload your firmware update file into the bucket.

Create an S3 bucket

To create an S3 bucket, open the [Amazon S3 console](#). Sign in if you haven't already and then perform the following steps:

1. Choose **Create bucket**.
2. Enter a unique and meaningful name for the **Bucket name**, (for example, `iotwirelessfwupdate`). For recommended naming convention for your bucket, see <https://docs.aws.amazon.com/AmazonS3/latest/userguide/bucketnamingrules.html>.
3. Make sure you selected the AWS Region selected as the one you used to create your LoRaWAN gateway and device, and the **Block all public access** setting is selected so that your bucket uses the default permissions.
4. Choose **Enable** for **Bucket versioning** which will help you keep multiple versions of the firmware update file in the same bucket.
5. Confirm **Server-side encryption** is set to **Disable** and choose **Create bucket**.

Upload your firmware update file

You can now see your bucket in the list of Buckets displayed in the AWS Management Console. Choose your bucket and complete the following steps to upload your file.

1. Choose your bucket and then choose **Upload**.
2. Choose **Add file** and then upload the firmware update file. If you followed the procedure described in [Generate the firmware update file and signature \(p. 1034\)](#), you'll upload the `fwstation` file, otherwise upload the file provided by your gateway manufacturer.
3. Make sure all settings are set to their default. Make sure that **Predefined ACLs** is set to **private** and choose **Upload** to upload your file.
4. Copy the S3 URI of the file you uploaded. Choose your bucket and you'll see the file you uploaded displayed in the list of **Objects**. Choose your file and then choose **Copy S3 URI**. The URI will be something like: `s3://iotwirelessfwupdate/fwstation` if you named your bucket similar to the example described previously (`fwstation`). You'll use the S3 URI when creating the IAM role.

Create an IAM role with permissions to read the S3 bucket

You'll now create an IAM role and policy that will give CUPS the permission to read your firmware update file from the S3 bucket.

Create an IAM policy for your role

To create an IAM policy for your AWS IoT Core for LoRaWAN destination role, open the [Policies hub of the IAM console](#) and then complete the following steps:

1. Choose **Create policy**, and choose the **JSON** tab.
2. Delete any content from the editor and paste this policy document. The policy provides permissions to access the `iotwireless` bucket and the firmware update file, `fwstation`, stored inside an object.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "VisualEditor0",  
            "Effect": "Allow",  
            "Action": [  
                "s3>ListBucketVersions",  
                "s3>ListBucket",  
                "s3GetObject"  
            ]  
        }  
    ]  
}
```

```
        ],
        "Resource": [
            "arn:aws:s3:::iotwirelessfwupdate/fwstation",
            "arn:aws:s3:::iotwirelessfwupdate"
        ]
    }
}
```

3. Choose **Review policy**, and in **Name**, enter a name for this policy (for example, `IoTWirelessFwUpdatePolicy`). You'll need this name to use in the next procedure.
4. Choose **Create policy**.

Create an IAM role with the attached policy

You'll now create an IAM role and attach the policy created previously for accessing the S3 bucket. Open the [Roles hub of the IAM console](#) and complete the following steps:

1. Choose **Create role**.
2. In **Select type of trusted entity**, choose **Another AWS account**.
3. In **Account ID**, enter your AWS account ID, and then choose **Next: Permissions**.
4. In the search box, enter the name of the IAM policy that you created in the previous procedure. Check the IAM policy (for example, `IoTWirelessFwUpdatePolicy`) you created earlier in the search results and choose it.
5. Choose **Next: Tags**, and then choose **Next: Review**.
6. In **Role name**, enter the name of this role (for example, `IoTWirelessFwUpdateRole`), and then choose **Create role**.

Edit trust relationship of the IAM role

In the confirmation message displayed after you ran the previous step, choose the name of the role you created to edit it. You'll edit the role to add the following trust relationship.

1. In the **Summary** section of the role you created, choose the **Trust relationships** tab, and then choose **Edit trust relationship**.
2. In **Policy Document**, change the **Principal** property to look like this example.

```
"Principal": {
    "Service": "iotwireless.amazonaws.com"
},
```

After you change the **Principal** property, the complete policy document should look like this example.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "iotwireless.amazonaws.com"
            },
            "Action": "sts:AssumeRole",
            "Condition": {}
        }
    ]
}
```

}

3. To save your changes and exit, choose **Update Trust Policy**.
4. Obtain the ARN for your role. Choose your IAM role and in the Summary section, you'll see a **Role ARN**, such as `arn:aws:iam::123456789012:role/IoTWirelessFwUpdateRole`. Copy this **Role ARN**.

Review the next steps

Now that you have created the S3 bucket and an IAM role that allows the CUPS server to read the S3 bucket, go to the next topic to schedule and run the firmware update. Keep the **S3 URI** and **Role ARN** that you copied previously so that you can enter them to create a task definition that will be run to perform the firmware update.

Schedule and run the firmware update by using a task definition

You can use a task definition to include details about the firmware update and define the update. AWS IoT Core for LoRaWAN provides a firmware update based on information from the following three fields associated with the gateway.

- **Station**

The version and build time of the Basics Station software. To identify this information, you can also generate it by using the Basics Station software that is being run by your gateway (for example, `2.0.5(rpi/std) 2021-03-09 03:45:09`).

- **PackageVersion**

The firmware version, specified by the file `version.txt` in the gateway. While this information might not be present in the gateway, we recommend it as a way to define your firmware version (for example, `1.0.0`).

- **Model**

The platform or model that is being used by the gateway (for example, Linux).

This procedure takes 20 minutes to complete.

To complete this procedure:

- [Get the current version running on your gateway \(p. 1040\)](#)
- [Create a wireless gateway task definition \(p. 1041\)](#)
- [Run the firmware update task and track progress \(p. 1042\)](#)

Get the current version running on your gateway

To determine your gateway's eligibility for a firmware update, the CUPS server checks all three fields, `Station`, `PackageVersion`, and `Model`, for a match when the gateway presents them during a CUPS request. When you use a task definition, these fields are stored as part of the `CurrentVersion` field.

You can use the AWS IoT Core for LoRaWAN API or AWS CLI to get the `CurrentVersion` for your gateway. Following commands show how to get this information using the CLI.

1. If you've already provisioned a gateway, you can get information about the gateway using the [`get-wireless-gateway`](#) command.

```
aws iotwireless get-wireless-gateway \
```

```
--identifier 5a11b0a85a11b0a8 \
--identifier-type GatewayEui
```

Following shows a sample output for the command.

```
{
    "Name": "Raspberry pi",
    "Id": "1352172b-0602-4b40-896f-54da9ed16b57",
    "Description": "Raspberry pi",
    "LoRaWAN": {
        "GatewayEui": "5a11b0a85a11b0a8",
        "RfRegion": "US915"
    },
    "Arn": "arn:aws:iotwireless:us-
east-1:231894231068:WirelessGateway/1352172b-0602-4b40-896f-54da9ed16b57"
}
```

- Using the wireless gateway ID reported by the `get-wireless-gateway` command, you can use the `get-wireless-gateway-firmware-information` command to get the `CurrentVersion`.

```
aws iotwireless get-wireless-gateway-firmware-information \
--id "3039b406-5cc9-4307-925b-9948c63da25b"
```

Following shows a sample output for the command, with information from all three fields displayed by the `CurrentVersion`.

```
{
    "LoRaWAN": {
        "CurrentVersion": {
            "PackageVersion": "1.0.0",
            "Model": "rpi",
            "Station": "2.0.5(rpi/std) 2021-03-09 03:45:09"
        }
    }
}
```

Create a wireless gateway task definition

When you create the task definition, we recommend that you specify automatic creation of tasks by using the `AutoCreateTasks` parameter. `AutoCreateTasks` applies to any gateway that has a match for all three parameters mentioned previously. If this parameter is disabled, the parameters have to be manually assigned to the gateway.

You can create the wireless gateway task definition by using the AWS IoT Core for LoRaWAN API or AWS CLI. Following commands show how to create the task definition using the CLI.

- Create a file, `input.json`, that'll contain the information to pass to the `CreateWirelessGatewayTaskDefinition` API. In the `input.json` file, provide the following information that you obtained earlier:

- UpdateDataSource**

Provide the link to your object containing the firmware update file that you uploaded to the S3 bucket. (for example, `s3://iotwirelessfwupdate/fwstation`.

- UpdateDataRole**

Provide the link to the Role ARN for the IAM role that you created, which provides permissions to read the S3 bucket. (for example, `arn:aws:iam::123456789012:role/IoTWirelessFwUpdateRole`.

- **SigKeyCRC and UpdateSignature**

This information might be provided by your gateway manufacturer, but if you followed the procedure described in [Generate the firmware update file and signature \(p. 1034\)](#), you'll find this information when generating the signature.

- **CurrentVersion**

Provide the `CurrentVersion` output that you obtained previously by running the `get-wireless-gateway-firmware-information` command.

```
cat input.json
```

Following shows the contents of the `input.json` file.

```
{  
    "AutoCreateTasks": true,  
    "Name": "FirmwareUpdate",  
    "Update":  
    {  
        "UpdateDataSource" : "s3://iotwirelessfwupdate/fwstation",  
        "UpdateDataRole" : "arn:aws:iam::123456789012:role/IoTWirelessFwUpdateRole",  
        "LoRaWAN" :  
        {  
            "SigKeyCrc": 3434210794,  
            "UpdateSignature": "MEQCIDPY/p2ssgXIPNC0gZr+NzeTLpX+WfBo5tYWbh5pQWN3AiBROen  
+XlIdMScvAsfVfU/ZScJCalhVNZh4esyS8mNIgA==",  
            "CurrentVersion" :  
            {  
                "PackageVersion": "1.0.0",  
                "Model": "rpi",  
                "Station": "2.0.5(rpi/std) 2021-03-09 03:45:09"  
            }  
        }  
    }  
}
```

2. Pass the `input.json` file to the [create-wireless-gateway-task-definition](#) command to create the task definition.

```
aws iotwireless create-wireless-gateway-task-definition \  
    --cli-input-json file://input.json
```

Following shows the output of the command.

```
{  
    "Id": "4ac46ff4-efc5-44fd-9def-e8517077bb12",  
    "Arn": "arn:aws:iotwireless:us-  
east-1:231894231068:WirelessGatewayTaskDefinition/4ac46ff4-efc5-44fd-9def-e8517077bb12"  
}
```

Run the firmware update task and track progress

The gateway is ready to receive the firmware update and, once powered on, it connects to the CUPS server. When the CUPS server finds a match in the version of the gateway, it schedules a firmware update.

A task is a task definition in process. As you specified automatic task creation by setting `AutoCreateTasks` to `True`, the firmware update task starts as soon as a matching gateway is found.

You can track the progress of the task by using the `GetWirelessGatewayTask` API. When you run the [get-wireless-gateway-task](#) command the first time, it will show the task status as `IN_PROGRESS`.

```
aws iotwireless get-wireless-gateway-task \
--id 1352172b-0602-4b40-896f-54da9ed16b57
```

Following shows the output of the command.

```
{
    "WirelessGatewayId": "1352172b-0602-4b40-896f-54da9ed16b57",
    "WirelessGatewayTaskDefinitionId": "ec11f9e7-b037-4fcc-aa60-a43b839f5de3",
    "LastUplinkReceivedAt": "2021-03-12T09:56:12.047Z",
    "TaskCreatedAt": "2021-03-12T09:56:12.047Z",
    "Status": "IN_PROGRESS"
}
```

When you run the command the next time, if the firmware update takes effect, it will show the updated fields, `Package`, `Version`, and `Model` and the task status changes to `COMPLETED`.

```
aws iotwireless get-wireless-gateway-task \
--id 1352172b-0602-4b40-896f-54da9ed16b57
```

Following shows the output of the command.

```
{
    "WirelessGatewayId": "1352172b-0602-4b40-896f-54da9ed16b57",
    "WirelessGatewayTaskDefinitionId": "ec11f9e7-b037-4fcc-aa60-a43b839f5de3",
    "LastUplinkReceivedAt": "2021-03-12T09:56:12.047Z",
    "TaskCreatedAt": "2021-03-12T09:56:12.047Z",
    "Status": "COMPLETED"
}
```

In this example, we showed you the firmware update using the Raspberry Pi based RAKWireless gateway. The firmware update script stops the running BasicStation to store the updated `Package`, `Version`, and `Model` fields so BasicStation will have to be restarted.

```
2021-03-12 09:56:13.108 [CUP:INFO] CUPS provided update.bin
2021-03-12 09:56:13.108 [CUP:INFO] CUPS provided signature len=70 keycrc=37316C36
2021-03-12 09:56:13.148 [CUP:INFO] ECDSA key#0 -> VERIFIED
2021-03-12 09:56:13.148 [CUP:INFO] Running update.bin as background process
2021-03-12 09:56:13.149 [SYS:VERB] /tmp/update.bin: Forked, waiting...
2021-03-12 09:56:13.151 [SYS:INFO] Process /tmp/update.bin (pid=6873) completed
2021-03-12 09:56:13.152 [CUP:INFO] Interaction with CUPS done - next regular check in 10s
```

If the firmware update fails, you see a status of `FIRST_RETRY` from the CUPS server, and the gateway sends the same request. If the CUPS server is unable to connect to the gateway after a `SECOND_RETRY`, it will show a status of `FAILED`.

After the previous task was `COMPLETED` or `FAILED`, delete the old task by using the [delete-wireless-gateway-task](#) command before starting a new one.

```
aws iotwireless delete-wireless-gateway-task \
--id 1352172b-0602-4b40-896f-54da9ed16b57
```

Managing devices with AWS IoT Core for LoRaWAN

LoRaWAN devices communicate with AWS IoT Core for LoRaWAN through LoRaWAN gateways. Adding devices to AWS IoT Core for LoRaWAN lets AWS IoT process the messages received from the devices for use by AWS IoT and other services.

Following are some important considerations when using your devices with AWS IoT Core for LoRaWAN. For information about how to add your device to AWS IoT Core for LoRaWAN, see [Onboard your devices to AWS IoT Core for LoRaWAN \(p. 1010\)](#).

Device considerations

When selecting a device that you want to use for communicating with AWS IoT Core for LoRaWAN, consider the following.

- Available sensors
- Battery capacity
- Energy consumption
- Cost
- Antenna type and transmission range

Using devices with gateways qualified for AWS IoT Core for LoRaWAN

The devices that you use can be paired with wireless gateways that are qualified for use with AWS IoT Core for LoRaWAN. You can find these gateways and developer kits in the [AWS Partner Device Catalog](#). We also recommend that you consider proximity of these devices to your gateways. For more information, see [Using qualified gateways from the AWS Partner Device Catalog \(p. 1031\)](#).

LoRaWAN version

AWS IoT Core for LoRaWAN supports all devices that comply to 1.0.x or 1.1 LoRaWAN specifications standardized by LoRa Alliance.

Activation modes

Before your LoRaWAN device can send uplink data, you must complete a process called *activation* or *join* procedure. To activate your device, you can either use OTAA (Over the air activation) or ABP (Activation by personalization). We recommend that you use OTAA to activate your device because new session keys are generated for each activation, which makes it more secure.

Your wireless device specification is based on the LoRaWAN version and activation mode, which determines the root keys and session keys generated for each activation. For more information, see [Add your wireless device specification to AWS IoT Core for LoRaWAN using the console \(p. 1011\)](#).

Device classes

LoRaWAN devices can send uplink messages at any time. Listening to downlink messages consumes battery capacity and reduces battery duration. The LoRaWAN protocol specifies three classes of LoRaWAN devices.

- Class A devices sleep most of the time and listen for downlink messages only for a short period of time. These devices are mostly battery-powered sensors with a battery lifetime of up to 10 years.
- Class B devices can receive messages in scheduled downlink slots. These devices are mostly battery-powered actuators.
- Class C devices never sleep and continuously listen to incoming messages and so there isn't much delay in receiving the messages. These devices are mostly mains-powered actuators.

For more information about these wireless device considerations, refer to the resources mentioned in [Learn more about LoRaWAN \(p. 1000\)](#).

View format of uplink messages sent from LoRaWAN devices

After you've connected your LoRaWAN device to AWS IoT Core for LoRaWAN, you can observe the format of the uplink message that you'll receive from your wireless device.

Before you can observe the uplink messages

You must have onboarded your wireless device and connected your device to AWS IoT so that it can transmit and receive data. For information about onboarding your device to AWS IoT Core for LoRaWAN, see [Onboard your devices to AWS IoT Core for LoRaWAN \(p. 1010\)](#).

What do the uplink messages contain?

LoRaWAN devices connect to AWS IoT Core for LoRaWAN by using LoRaWAN gateways. The uplink message that you receive from the device will contain the following information.

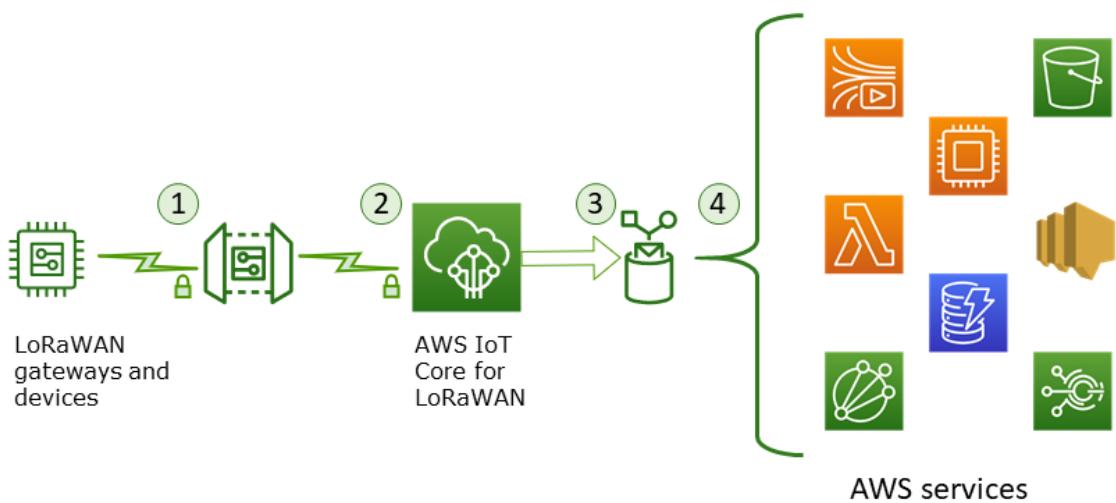
- Payload data that corresponds to the encrypted payload message that is sent from the wireless device.
- Wireless metadata that includes:
 - Device information such as DevEui, the data rate, and the frequency channel in which the device is operating.
 - Optional additional parameters and the gateway information for gateways that are connected to the device. The gateway parameters include the gateway's EUI, the SNR, and RSSI.

By using the wireless metadata, you can obtain useful information about the wireless device and the data that is transmitted between your device and AWS IoT. For example, you can use the `AckedMessageId` parameter to check whether the last confirmed downlink message has been received by the device. Optionally, if you choose to include the gateway information, you can identify whether you want to switch to a stronger gateway channel that's closer to your device.

How to observe the uplink messages?

After you've onboarded your device, you can use the [MQTT test client](#) on the **Test** page of the AWS IoT console to subscribe to the topic that you specified when creating your destination. You'll start to see messages after your device is connected and starts sending payload data.

This diagram identifies the key elements in a LoRaWAN system connected to AWS IoT Core for LoRaWAN, which shows the primary data plane and how data flows through the system.



When the wireless device starts sending uplink data, AWS IoT Core for LoRaWAN wraps the wireless metadata information with the payload and then sends it to your AWS applications.

Uplink message example

The following example shows the format of the uplink message received from your device.

```
{  
    "WirelessDeviceId": "5b58245e-146c-4c30-9703-0ca942e3ff35",  
    "PayloadData": "Cc48AAAAAAAAAA=",  
    "WirelessMetadata":  
    {  
        "LoRaWAN":  
        {  
            "ADR": false,  
            "Bandwidth": 125,  
            "ClassB": false,  
            "CodeRate": "4/5",  
            "DataRate": "0",  
            "DevAddr": "00b96cd4",  
            "DevEui": "58a0cb000202c99",  
            "FOptLen": 2,  
            "FCnt": 1,  
            "Fport": 136,  
            "Frequency": "868100000",  
            "Gateways": [  
                {  
                    "GatewayEui": "80029cfffe5cf1cc",  
                    "Snr": -29,  
                    "Rssi": 9.75  
                }  
            ],  
            "MIC": "7255cb07",  
            "MType": "UnconfirmedDataUp",  
            "Major": "LoRaWANR1",  
            "Modulation": "LORA",  
            "PolarizationInversion": false,  
            "SpreadingFactor": 12,  
            "Timestamp": "2021-05-03T03:24:29Z"  
        }  
    }  
}
```

}

For information about the different parameters in the uplink metadata, see [SendDataToWirelessDevice](#). Note that the `SendDataToWirelessDevice` API is used for data that is sent from AWS IoT Core for LoRaWAN to your wireless device. You can use the API parameters as a reference to learn about the different fields in your uplink message.

Exclude gateway metadata from uplink metadata

If you want to exclude the gateway metadata information from your uplink metadata, disable the `AddGwMetadata` parameter when you create the service profile. For information about disabling this parameter, see [Add service profiles \(p. 1014\)](#).

In this case, you won't see the `Gateways` section in the uplink metadata, as illustrated in the following example.

```
{  
    "WirelessDeviceId": "0d9a439b-e77a-4573-a791-49d5c0f4db95",  
    "PayloadData": "AAAAAAA//8=",  
    "WirelessMetadata": {  
        "LoRaWAN": {  
            "ClassB": false,  
            "CodeRate": "4/5",  
            "DataRate": "1",  
            "DevAddr": "01920f27",  
            "DevEui": "fffffff10000163b0",  
            "FCnt": 1,  
            "FPort": 5,  
            "Timestamp": "2021-04-29T05:19:43.646Z"  
        }  
    }  
}
```

Create multicast groups to send a downlink payload to multiple devices

To send a downlink payload to multiple devices, create a multicast group. Using multicast, a source can send data to a single multicast address, which is then distributed to an entire group of recipient devices.

Devices in a multicast group share the same multicast address, session keys, and frame counter. By using the same session keys, devices in a multicast group can decrypt the message when a downlink transmission is initiated. A multicast group only supports downlink. It doesn't confirm whether the downlink payload has been received by the devices.

With AWS IoT Core for LoRaWAN's multicast groups, you can:

- Filter your list of devices by using the device profile, RFRegion, or device class, and then add these devices to a multicast group.
- Schedule and send one or more downlink payload messages to devices in a multicast group, within a 48-hour distribution window.
- Have devices temporarily switch to Class B or class C mode at the start of your multicast session for receiving the downlink message.
- Monitor your multicast group setup and the state of its devices, and also troubleshoot any issues.
- Use Firmware Updates-Over-The-Air (FUOTA) to securely deploy firmware updates to devices in a multicast group.

AWS IoT Core for LoRaWAN's support for FUOTA and multicast groups is based on the [LoRa Alliance's](#) following specifications:

- LoRaWAN Remote Multicast Setup Specification, TS005-2.0.0
- LoRaWAN Fragmented Data Block Transportation Specification, TS004-2.0.0
- LoRaWAN Application Layer Clock Synchronization Specification, TS003-2.0.0

Note

AWS IoT Core for LoRaWAN automatically performs the clock synchronization for the device according to the LoRa Alliance specification. Using function `AppTimeReq`, it replies the server-side time to the devices that request it using `ClockSync` signaling.

The following shows how to create your multicast group and schedule a downlink message.

Topics

- [Prepare devices for multicast and FUOTA configuration \(p. 1048\)](#)
- [Create multicast groups and add devices to the group \(p. 1050\)](#)
- [Monitor and troubleshoot status of your multicast group and devices in the group \(p. 1054\)](#)
- [Schedule a downlink message to send to devices in your multicast group \(p. 1055\)](#)

Prepare devices for multicast and FUOTA configuration

When you add your wireless device to AWS IoT Core for LoRaWAN, you can prepare your wireless device for multicast setup and FUOTA configuration by using the console or the CLI. If you're performing this configuration for the first time, we recommend that you use the console. To manage your multicast group and add or remove a number of devices from your group, we recommend using the CLI to manage a large number of resources.

GenAppKey and FPorts

When you add your wireless device, before you can add your devices to multicast groups or perform FUOTA updates, configure the following parameters. Before you configure these parameters, make sure that your devices support FUOTA and multicast and your wireless device specification is either OTAA v1.1 or OTAAv1.0.x.

- **GenAppKey:** For devices that support the LoRaWAN version 1.0.x and to use multicast groups, the GenAppKey is the device-specific root key from which the session keys for your multicast group are derived.

Note

For LoRaWAN devices that use the wireless specification OTAA v1.1, the `AppKey` is used for the same purpose as the `GenAppKey`.

To set up the parameters to initiate the data transfer, AWS IoT Core for LoRaWAN distributes session keys with the end devices. For more information about LoRaWAN versions, see [LoRaWAN version \(p. 1044\)](#).

Note

AWS IoT Core for LoRaWAN stores the `GenAppKey` information that you provide in an encrypted format.

- **FPorts:** According to the LoRaWAN specifications for FUOTA and multicast groups, AWS IoT Core for LoRaWAN assigns the default values for the following fields of the `FPorts` parameter. If you have already assigned any of the following `FPort` values, then you can choose a different value that is available, from 1 to 223.
 - **Multicast:** 200

This **FPort** value is used for multicast groups.

- FUOTA: 201

This **FPort** value is used for FUOTA.

- ClockSync: 202

This **FPort** value is used for the clock synchronization.

Device profiles for multicast and FUOTA

At the start of a multicast session, a class B or class C distribution window is used to send the downlink message to the devices in your group. The devices that you add for multicast and FUOTA must support class B or class C modes of operation. Depending on the device class that your device supports, choose a device profile for your device that has either or both class B or class C modes enabled.

For information about device profiles, see [Add profiles to AWS IoT Core for LoRaWAN \(p. 1013\)](#).

Prepare devices for multicast and FUOTA by using the console

To specify the FPorts and GenAppKey parameters for multicast setup and FUOTA by using the console:

1. Navigate to the [Devices hub of the AWS IoT console](#) and choose **Add wireless device**.
2. Choose the **Wireless device specification**. Your device must use OTAA for device activation. When you choose OTAA v1.0.x or OTAA v1.1, a **FUOTA configuration-Optional** section appears.
3. Enter the EUI (Extended Unique Identifier) parameters for your wireless device.
4. Expand the **FUOTA configuration-Optional** section and then choose **This device supports firmware updates over the air (FUOTA)**. You can now enter the **FPort** values for multicast, FUOTA, and clock sync. If you chose OTAA v1.0.x for the wireless device specification, enter the **GenAppKey**.
5. Add your device to AWS IoT Core for LoRaWAN by choosing your profiles and a destination for routing messages. For the device profile linked to the device, make sure you select one or both **Supports Class B** or **Supports Class C** modes.

Note

To specify the FUOTA configuration parameters, you must use the [Devices hub of the AWS IoT console](#). These parameters don't appear if you onboard your devices by using the [Intro](#) page of the AWS IoT console.

For more information about the wireless device specification and onboarding your device, see [Add your wireless device to AWS IoT Core for LoRaWAN \(p. 1011\)](#).

Note

You can specify these parameters only when you create the wireless device. You can't change or specify parameters when you update an existing device.

Prepare devices for multicast and FUOTA by using the API operation

To use multicast groups or to perform FUOTA updates, configure these parameters by using the `createWirelessDevice` API operation or the `create-wireless-device` CLI command. In addition to specifying the application key and FPorts parameters, make sure that the device profile that's linked to the device supports one or both class B or class C modes.

You can provide an `input.json` file as input to the `create-wireless-device` command.

```
aws iotwireless create-wireless-device \
```

```
--cli-input-json file://input.json
```

where:

Contents of input.json

```
{  
    "Description": "My LoRaWAN wireless device"  
    "DestinationName": "IoTWirelessDestination"  
    "LoRaWAN": {  
        "DeviceProfileId": "ab0c23d3-b001-45ef-6a01-2bc3de4f5333",  
        "ServiceProfileId": "fe98dc76-cd12-001e-2d34-5550432da100",  
        "FPorts": {  
            "ClockSync": 202,  
            "Fuota": 201,  
            "Multicast": 200  
        },  
        "OtaaV1_0_x": {  
            "AppKey": "3f4ca100e2fc675ea123f4eb12c4a012",  
            "AppEui": "b4c231a359bc2e3d",  
            "GenAppKey": "01c3f004a2d6efffe32c4eda14bcd2b4"  
        },  
        "DevEui": "ac12efc654d23fc2"  
    },  
    "Name": "SampleIoTWirelessThing"  
    "Type": LoRaWAN  
}
```

For information about the CLI commands that you can use, see [AWS CLI reference](#).

Note

After you specify the values of these parameters, you can't update them by using the `UpdateWirelessDevice` API operation. Instead, you can create a new device with the values for the parameters `GenAppKey` and `FPorts`.

To get information about the values specified for these parameters, you can use the `GetWirelessDevice` API operation or the `get-wireless-device` CLI command.

Next steps

After you've configured the parameters, you can create multicast groups and FUOTA tasks to send downlink payload or update the firmware of your LoRaWAN devices.

- For information about creating multicast groups, see [Create multicast groups and add devices to the group \(p. 1050\)](#).
- For information about creating FUOTA tasks, see [Create FUOTA task and provide firmware image \(p. 1060\)](#).

Create multicast groups and add devices to the group

You can create multicast groups by using the console or the CLI. If you're creating your multicast group for the first time, we recommend that you use the console to add your multicast group. When you want to manage your multicast group and add or remove devices from your group, you can use the CLI.

After exchanging signaling with the end devices you added, AWS IoT Core for LoRaWAN establishes the shared keys with the end devices and sets up the parameters for the data transfer.

Prerequisites

Before you can create multicast groups and add devices to the group:

- Prepare your devices for multicast and FUOTA setup by specifying the FUOTA configuration parameters `GenAppKey` and `FPorts`. For more information, see [Prepare devices for multicast and FUOTA configuration \(p. 1048\)](#).
- Check whether the devices support class B or class C modes of operation. Depending on the device class that your device supports, choose a device profile that has one or both **Supports Class B** or **Supports Class C** modes enabled. For information about device profiles, see [Add profiles to AWS IoT Core for LoRaWAN \(p. 1013\)](#).

At the start of the multicast session, a class B or class C distribution window is used to send downlink messages to the devices in your group.

Create multicast groups by using the console

To create multicast groups by using the console, go to the [Multicast groups](#) page of the AWS IoT console and choose **Create multicast group**.

1. Create a multicast group

To create your multicast group, specify the multicast properties and tags for your group.

1. Specify multicast properties

To specify multicast properties, enter the following information for your multicast group.

- **Name:** Enter a unique name for your multicast group. The name must contain only letters, numbers, hyphens, and underscores. It can't contain spaces.
- **Description:** You can provide an optional description for your multicast group. The description length can be up to 2,048 characters.

2. Tags for multicast group

You can optionally provide any key-value pairs as **Tags** for your multicast group. To continue creating your multicast group, choose **Next**.

2. Add devices to a multicast group

You can add individual devices or a group of devices to your multicast group. To add devices:

1. Specify RFRegion

Specify the **RFRegion** or frequency band for your multicast group. The **RFRegion** for your multicast group must match the **RFRegion** of devices that you add to the multicast group. For more information about the **RFRegion**, see [Consider selection of LoRa frequency bands for your gateways and device connection \(p. 1005\)](#).

2. Select a multicast device class

Choose whether you want devices in the multicast group to switch to a class B or class C mode at the start of the multicast session. A class B session can receive downlink messages at regular downlink slots and a class C session can receive downlink messages at anytime.

3. Choose the devices you want to add to the group

Choose whether you want to add devices individually or in bulk to the multicast group.

- To add devices individually, enter the wireless device ID of each device that you want to add to your group.
- To add devices in bulk, you can filter the devices you want to add by device profile or tags. For device profile, you can add devices with a profile that supports class B, class C, or both device classes.

4. To create your multicast group, choose **Create**.

The multicast group details and the devices you added appear in the group. For information about the status of the multicast group and your devices and for troubleshooting any issues, see [Monitor and troubleshoot status of your multicast group and devices in the group \(p. 1054\)](#).

After creating a multicast group, you can choose **Action** to edit, delete, or add devices to the multicast group. After you've added the devices, you can schedule a session for the downlink payload to be sent to the devices in your group.

Create multicast groups by using the API

To create multicast groups and add devices to the group by using the API:

1. Create a multicast group

To create your multicast group, use the [CreateMulticastGroup](#) API operation or the [create-multicast-group](#) CLI command. You can provide an `input.json` file as input to the `create-multicast-group` command.

```
aws iotwireless create-multicast-group \
--cli-input-json file://input.json
```

where:

Contents of `input.json`

```
{
  "Description": "Multicast group to send downlink payload and perform FUOTA updates.",
  "LoRaWAN": {
    "DLClass": "ClassB",
    "RfRegion": "US915"
  },
  "Name": "MC_group_FUOTA"
}
```

After you create your multicast group, you can use the following API operations or CLI commands to update, delete, or get information about your multicast groups.

- [UpdateMulticastGroup](#) or [update-multicast-group](#)
- [GetMulticastGroup](#) or [get-multicast-group](#)
- [ListMulticastGroups](#) or [list-multicast-groups](#)
- [DeleteMulticastGroup](#) or [delete-multicast-group](#)

2. Add devices to a multicast group

You can add devices to your multicast group individually or in bulk.

- To add devices in bulk to your multicast group, use the [StartBulkAssociateWirelessDeviceWithMulticastGroup](#) API operation or the [start-bulk-associate-wireless-device-with-multicast-group](#) CLI command. To filter the devices you want to associate in bulk to your multicast group, provide a query string. The following shows how you can add a group of devices that has a device profile with the specified ID linked to it.

```
aws iotwireless start-bulk-associate-wireless-device-with-multicast-group \
--id "12abd34e-5f67-89c2-9293-593b1bd862e0" \
```

```
--cli-input-json file://input.json
```

where:

Contents of input.json

```
{  
    "QueryString": "DeviceProfileName: MyWirelessDevice AND DeviceProfileId:  
d6d8ef8e-7045-496d-b3f4-ebcaa1d564bf",  
    "Tags": [  
        {  
            "Key": "Multicast",  
            "Value": "ClassB"  
        }  
    ]  
}
```

Here, `multicast-groups/d6d8ef8e-7045-496d-b3f4-ebcaa1d564bf/bulk` is the URL that's used to associate devices with the group.

- To add devices individually to your multicast group, use the [AssociateWirelessDeviceWithMulticastGroup](#) API operation or the [associate-wireless-device-with-multicast-group](#) CLI. Provide the wireless device ID for each device you want to add to your group.

```
aws iotwireless associate-wireless-device-with-multicast-group \  
--id "12abd34e-5f67-89c2-9293-593b1bd862e0" \  
--wireless-device-id "ab0c23d3-b001-45ef-6a01-2bc3de4f5333"
```

After you create your multicast group, you can use the following API operations or CLI commands to get information about your multicast group or to disassociate devices.

- [DisassociateWirelessDeviceFromMulticastGroup](#) or [disassociate-wireless-device-from-multicast-group](#)
- [StartBulkDisassociateWirelessDeviceFromMulticastGroup](#) or [start-bulk-disassociate-wireless-device-from-multicast-group](#)
- [ListWirelessDevices](#) or [list-wireless-devices](#)

Note

The [ListWirelessDevices](#) API operation can be used to list wireless devices in general, and wireless devices that are associated with a multicast group or a FUOTA task.

- To list wireless devices associated with a multicast group, use the [ListWirelessDevices](#) API operation with `MulticastGroupID` as the filter.
- To list wireless devices associated with a FUOTA task, use the [ListWirelessDevices](#) API operation with `FuotaTaskID` as the filter.

Next steps

After you've created a multicast group and added devices, you can continue adding devices and monitor the status of the multicast group and your devices. If your devices have been added successfully to the group, you can configure and schedule a downlink message to be sent to the devices. Before you can send a downlink message, your devices' status must be **Multicast setup ready**. After you schedule a downlink message, the status changes to **Session attempting**. For more information, see [Schedule a downlink message to send to devices in your multicast group \(p. 1055\)](#).

If you want to update the firmware of the devices in the multicast group, you can perform Firmware Updates Over-The-Air (FUOTA) with AWS IoT Core for LoRaWAN. For more information, see [Firmware Updates Over-The-Air \(FUOTA\) for AWS IoT Core for LoRaWAN devices \(p. 1057\)](#).

If your devices weren't added or if you see an error in the multicast group or device statuses, you can hover over the error to get more information and resolve it. If you still see an error, for information about how to troubleshoot and resolve the issue, see [Monitor and troubleshoot status of your multicast group and devices in the group \(p. 1054\)](#).

Monitor and troubleshoot status of your multicast group and devices in the group

After you've added devices and created your multicast group, open the AWS Management Console. Navigate to the [Multicast groups](#) page of the AWS IoT console and choose the multicast group you created to view its details. You'll see information about the multicast group, the number of devices that have been added, and device status details. You can use the status information to track progress of your multicast session and troubleshoot any errors.

Multicast group status

Your multicast group can have one of the following status messages displayed in the AWS Management Console.

- **Pending**

This status indicates that you've created a multicast group but it doesn't yet have a multicast session. You'll see this status message displayed when your group has been created. During this time, you can update your multicast group, and associate or disassociate devices with your group. After the status changes from **Pending**, additional devices can't be added to the group.

- **Session attempting**

After your devices have been added successfully to the multicast group, when your group has a scheduled multicast session, you'll see this status message displayed. During this time, you can't update or add devices to your multicast group. If you cancel your multicast session, the group status changes to **Pending**.

- **In session**

When it's the earliest session time for your multicast session, you'll see this status message displayed. A multicast group also continues to be in this state when it's associated with a FUOTA task that has an ongoing firmware update session.

If you don't have an associated FUOTA task in session, and if the multicast session is canceled because the session time exceeded the time out or you canceled your multicast session, the group status changes to **Pending**.

- **Delete waiting**

If you delete your multicast group, its group status changes to **Delete waiting**. Deletions are permanent and can't be undone. This action can take time and the group status will be **Delete_Waiting** until the multicast group has been deleted. After your multicast group enters this state, it can't transition to one of the other states.

Status of devices in multicast group

The devices in your multicast group can have one of the following status messages displayed in the AWS Management Console. You can hover over each status message to get more information about what it indicates.

- **Package attempting**

After your devices have been associated with the multicast group, the device status is **Package attempting**. This status indicates that AWS IoT Core for LoRaWAN has not yet confirmed whether the device supports multicast setup and operation.

- **Package unsupported**

After your devices have been associated with the multicast group, AWS IoT Core for LoRaWAN checks whether your device's firmware is capable of multicast setup and operation. If your device doesn't have the supported multicast package, its status is **Package unsupported**. To resolve the error, check whether your device's firmware is capable of multicast setup and operation.

- **Multicast setup attempting**

If the devices associated with your multicast group are capable of multicast setup and operation, the status is **Multicast setup attempting**. This status indicates that the device hasn't completed the multicast setup yet.

- **Multicast setup ready**

Your device has completed the multicast setup and has been added to the multicast group. This status indicates that the devices are ready for a multicast session and a downlink message can be sent to those devices. The status also indicates when you can use FUOTA to update the firmware of the devices in the group.

- **Session attempting**

A multicast session has been scheduled for the devices in your multicast group. At the start of a multicast group session, the device status is **Session attempting**, and requests are sent for whether a class B or class C distribution window can be initiated for the session. If the time it takes to set up the multicast session exceeds the timeout or if you cancel the multicast session, the status changes to **Multicast setup done**.

- **In session**

This status indicates that a class B or class C distribution window has been initiated and your device has an ongoing multicast session. During this time, downlink messages can be sent from AWS IoT Core for LoRaWAN to devices in the multicast group. If you update your session time, it overrides the current session and the status changes to **Session attempting**. When the session time ends or if you cancel the multicast session, the status changes to **Multicast setup ready**.

Next steps

Now that you've learned the different statuses of your multicast group and the devices in your group, and how you can troubleshoot any issues such as when a device is not capable of multicast setup, you can schedule a downlink message to be sent to the devices and your multicast group will be in **In session**. For information about scheduling a downlink message, see [Schedule a downlink message to send to devices in your multicast group \(p. 1055\)](#).

Schedule a downlink message to send to devices in your multicast group

After you've successfully added devices to your multicast group, you can start a multicast session and configure a downlink message to be sent to those devices. The downlink message must be scheduled within 48 hours and the start time for the multicast must be at least 30 minutes later than the current time.

Note

Devices in a multicast group can't acknowledge when a downlink message has been received.

Prerequisites

Before you can send a downlink message, you must have created a multicast group and successfully added devices to the group for which you want to send a downlink message. You can't add more devices after a start time has been scheduled for your multicast session. For more information, see [Create multicast groups and add devices to the group \(p. 1050\)](#).

If any of the devices weren't added successfully, the multicast group and device status will contain information to help you resolve the errors. If the errors still persist, for information about troubleshooting these errors, see [Monitor and troubleshoot status of your multicast group and devices in the group \(p. 1054\)](#).

Schedule a downlink message by using the console

To send a downlink message by using the console, go to the [Multicast groups](#) page of the AWS IoT console and choose the multicast group you created. In the multicast group details page, choose **Schedule downlink message** and then choose **Schedule downlink session**.

1. Schedule downlink message window

You can set up a time window for a downlink message to be sent to the devices in your multicast group. The downlink message must be scheduled within 48 hours.

To schedule your multicast session, specify the following parameters:

- **Start date and Start time:** The start date and time must be at least 30 minutes after and 48 hours before the current time.

Note

The time that you specify is in UTC so consider checking the time difference with your time zone when scheduling the downlink window.

- **Session timeout:** The time after which you want the multicast session to timeout if no downlink message has been received. The minimum timeout allowed is 60 seconds. The maximum timeout value is 2 days for class B multicast groups and 18 hours for class C multicast groups.

2. Configure your downlink message

To configure your downlink message, specify the following parameters:

- **Data rate:** Choose a data rate for your downlink message. The data rate depends on RFRegion and payload size. The default data rate is 8 for the US915 region and 0 for the EU868 region.
- **Frequency:** Choose a frequency for sending your downlink message. To avoid messaging conflicts, choose an available frequency depending on the RFRegion.
- **FPort:** Choose an available frequency port for sending the downlink message to your devices.
- **Payload:** Specify the maximum size of your payload depending on the data rate. Using the default data rate, you can have a maximum payload size of 33 bytes in the US915 RfRegion and 51 bytes in the EU868 RfRegion. Using larger data rates, you can transfer up to a maximum payload size of 242 bytes.

To schedule your downlink message, choose **Schedule**.

Schedule a downlink message by using the API

To schedule a downlink message by using the API, use the [StartMulticastGroupSession](#) API operation or the [start-multicast-group-session](#) CLI command.

You can use the following API operations or CLI commands to get information about a multicast group and to delete a multicast group.

- [GetMulticastGroupSession](#) or `get-multicast-group-session`
- [DeleteMulticastGroupSession](#) or `delete-multicast-group-session`

To send data to a multicast group after the session has been started, use the [SendDataToMulticastGroup](#) API operation or the `send-data-to-multicast-group` CLI command.

Next steps

After you've configured a downlink message to be sent to the devices, the message is sent at the start of the session. The devices in a multicast group can't confirm whether the message has been received.

Configure additional downlink messages

You can also configure additional downlink messages to be sent to the devices in your multicast group:

- To configure additional downlink messages from the console:
 1. Go to the [Multicast groups](#) page of the AWS IoT console and choose the multicast group you created.
 2. In the multicast group details page, choose **Schedule downlink message** and then choose **Configure additional downlink message**.
 3. Specify the parameters **Data rate**, **Frequency**, **FPort**, and **Payload**, similar to how you configured these parameters for your first downlink message.
- To configure additional downlink messages using the API or CLI, call the [SendDataToMulticastGroup](#) API operation or the `send-data-to-multicast-group` CLI command for each additional downlink message.

Update session schedule

You can also update the session schedule to use a new start date and time for your multicast session. The new session schedule will override the previously scheduled session.

Note

Update your multicast session only when required. These updates can cause a group of devices to wake up for a long duration and drain the battery.

- To update the session schedule from the console:
 1. Go to the [Multicast groups](#) page of the AWS IoT console and choose the multicast group you created.
 2. In the multicast group details page, choose **Schedule downlink message** and then choose **Update session schedule**.
 3. Specify the parameters **State date**, **Start time**, and **Session timeout**, similar to how you specified these parameters for your first downlink message.
- To update the session schedule from the API or CLI, use the [StartMulticastGroupSession](#) API operation or the `start-multicast-group-session` CLI command.

Firmware Updates Over-The-Air (FUOTA) for AWS IoT Core for LoRaWAN devices

Use Firmware Updates Over-The-Air (FUOTA) to deploy firmware updates to AWS IoT Core for LoRaWAN devices.

Using FUOTA, you can send firmware updates to individual devices or to a group of devices. You can also send firmware updates to multiple devices by creating a multicast group. First add your devices to the

multicast group, and then send your firmware update image to all those devices. We recommend that you digitally sign the firmware images so that devices receiving the images can verify that they're coming from the right source.

With AWS IoT Core for LoRaWAN's FUOTA updates, you can:

- Deploy new firmware images or delta images to a single device or a group of devices.
- Verify the authenticity and integrity of new firmware after it's deployed to devices.
- Monitor the progress of a deployment and debug issues in case of a failed deployment.

AWS IoT Core for LoRaWAN's support for FUOTA and multicast groups is based on the [LoRa Alliance's](#) following specifications:

- LoRaWAN Remote Multicast Setup Specification, TS005-2.0.0
- LoRaWAN Fragmented Data Block Transportation Specification, TS004-2.0.0
- LoRaWAN Application Layer Clock Synchronization Specification, TS003-2.0.0

Note

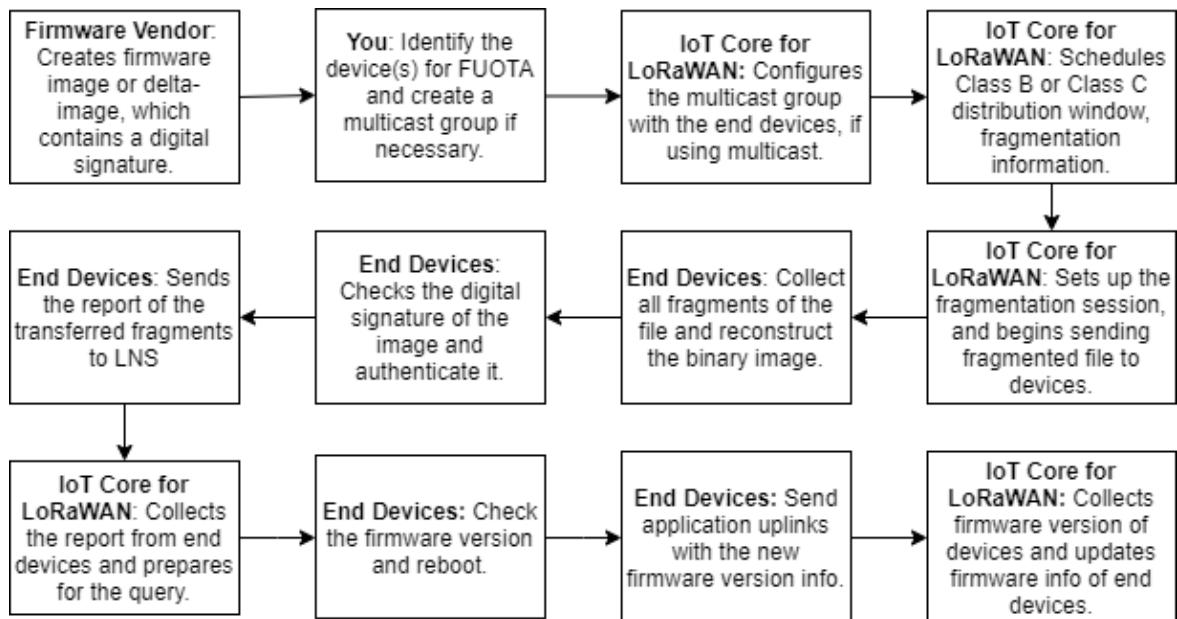
AWS IoT Core for LoRaWAN automatically performs the clock synchronization according to the LoRa Alliance specification. It uses the function `AppTimeReq` to reply the server-side time to the devices that request it using `ClockSync` signaling.

The following shows how to perform FUOTA updates.

- [FUOTA process overview \(p. 1058\)](#)
- [Create FUOTA task and provide firmware image \(p. 1060\)](#)
- [Add devices and multicast groups to a FUOTA task and schedule a FUOTA session \(p. 1062\)](#)
- [Monitor and troubleshoot the status of your FUOTA task and devices added to the task \(p. 1065\)](#)

FUOTA process overview

The following diagram shows how AWS IoT Core for LoRaWAN performs the FUOTA process for your end devices. If you're adding individual devices to your FUOTA session, you can skip the steps for creating and configuring your multicast group. You can add your devices directly to a FUOTA session, and AWS IoT Core for LoRaWAN will then start the firmware update process.



To perform FUOTA updates for your devices, first create your digitally signed firmware image and configure the devices and multicast groups that you want to add to your FUOTA task. After you start a FUOTA session, your end devices collect all fragments, reconstruct the image from the fragments, report the status to AWS IoT Core for LoRaWAN, and then apply the new firmware image.

The following illustrates the different steps in the FUOTA process:

1. Create a firmware image or delta image with a digital signature

For AWS IoT Core for LoRaWAN to perform FUOTA updates for your LoRaWAN devices, we recommend that you digitally sign the firmware image or the delta image when sending firmware updates over the air. The devices that receive the images can then verify that it's coming from the right source.

Your firmware image must not be larger than 1 megabyte in size. The larger your firmware size, the longer it can take for your update process to complete. For faster data transfer or if your new image is larger than 1 Megabyte, use a delta image, which is the part of your new image that's the delta between your new firmware image and the previous image.

Note

AWS IoT Core for LoRaWAN doesn't provide the digital signature generation tool and the firmware version management system. You can use any third-party tool to generate the digital signature for your firmware image. We recommend that you use a digital signature tool such as the one embedded in the [ARM Mbed GitHub repository](#), which also includes tools for generating the delta image and for devices to use that image.

2. Identify and configure the devices for FUOTA

After you identify the devices for FUOTA, send firmware updates to individual or multiple devices.

- To send your firmware updates to multiple devices, create a multicast group, and configure the multicast group with end devices. For more information, see [Create multicast groups to send a downlink payload to multiple devices \(p. 1047\)](#).
- To send firmware updates to individual devices, add those devices to your FUOTA session and then perform the firmware update.

3. Schedule a distribution window and set up fragmentation session

If you created a multicast group, you can specify the class B or class C distribution window to determine when the devices can receive the fragments from AWS IoT Core for LoRaWAN. Your devices might be operating in class A before they switch to class B or class C mode. You must also specify the start time of the session.

Class B or class C devices wake up at the specified distribution window and start receiving the downlink packets. Devices operating in class C mode can consume more power than class B devices. For more information, see [Device classes \(p. 1044\)](#).

4. End devices report status to AWS IoT Core for LoRaWAN and update firmware image

After you set up a fragmentation session, your end devices and AWS IoT Core for LoRaWAN perform the following steps to update the firmware of your devices.

1. Because LoRaWAN devices have a low data rate, to start the FUOTA process, AWS IoT Core for LoRaWAN sets up a fragmentation session to fragment the firmware image. Then it sends these fragments to the end devices.
2. After AWS IoT Core for LoRaWAN sends the image fragments, your LoRaWAN end devices perform the following tasks.
 - a. Collect the fragments and then reconstruct the binary image from these fragments.
 - b. Check the digital signature of the reconstructed image to authenticate the image and verify that it's coming from the right source.
 - c. Compare the firmware version from AWS IoT Core for LoRaWAN to the current version.
 - d. Report the status of the fragmented images that were transferred to AWS IoT Core for LoRaWAN, and then apply the new firmware image.

Note

In some cases, the end devices report the status of the fragmented images that were transferred to AWS IoT Core for LoRaWAN before checking the digital signature of the firmware image.

Now that you've learned the FUOTA process, you can create your FUOTA task and add devices to the task for updating their firmware. For more information, see [Create FUOTA task and provide firmware image \(p. 1060\)](#).

Create FUOTA task and provide firmware image

To update the firmware of your LoRaWAN devices, first create a FUOTA task and provide the digitally signed firmware image you want to use for the update. You can then add your devices and multicast groups to the task and schedule a FUOTA session. When the session starts, AWS IoT Core for LoRaWAN sets up a fragmentation session and your end devices collect the fragments, reconstruct the image, and apply the new firmware. For information about the FUOTA process, see [FUOTA process overview \(p. 1058\)](#).

The following shows how you can create a FUOTA task and upload the firmware image or delta image that you'll store in an S3 bucket.

Prerequisites

Before you can perform FUOTA updates, the firmware image must be digitally signed so that your end devices can verify the authenticity of the image when applying the image. You can use any third-party tool to generate the digital signature for your firmware image. We recommend that you use a digital signature tool such as the one embedded in the [ARM Mbed GitHub repository](#), which also includes tools for generating the delta image and for devices to use that image.

Create FUOTA task and upload firmware image by using the console

To create a FUOTA task and upload your firmware image by using the console, go to the [FUOTA tasks](#) tab of the console and then choose **Create FUOTA task**.

1. Create FUOTA task

To create your FUOTA task, specify the task properties and tags.

1. Specify FUOTA task properties

To specify FUOTA task properties, enter the following information for your FUOTA task.

- **Name:** Enter a unique name for your FUOTA task. The name must contain only letters, numbers, hyphens, and underscores. It can't contain spaces.
- **Description:** You can provide an optional description for your multicast group. The description field can be up to 2,048 characters.
- **RFRegion:** Set the frequency band for your FUOTA task. The frequency band must match the one you used to provision your wireless devices or multicast groups.

2. Tags for FUOTA task

You can optionally provide any key-value pairs as **Tags** for your FUOTA task. To continue creating your task, choose **Next**.

2. Upload firmware image

Choose the firmware image file that you want to use to update the firmware of the devices you add to the FUOTA task. The firmware image file is stored in an S3 bucket. You can provide AWS IoT Core for LoRaWAN the permissions to access the firmware image on your behalf. We recommend that you digitally sign the firmware images so that its authenticity is verified when the firmware update is performed.

1. Choose firmware image file

You can either upload a new firmware image file to an S3 bucket or choose an existing image that has already been uploaded to an S3 bucket.

Note

The firmware image file must not be larger than 1 megabyte in size. The larger your firmware size, the longer it can take for your update process to complete.

- To use an existing image, choose **Select an existing firmware image**, choose **Browse S3**, and then choose the firmware image file you want to use.

AWS IoT Core for LoRaWAN populates the S3 URL, which is the path to your firmware image file in the S3 bucket. The format of the path is `s3://bucket_name/file_name`. To view the file in the [Amazon Simple Storage Service](#) console, choose **View**.

- To upload a new firmware image.
 - a. Choose **Upload a new firmware image** and upload your firmware image. The image file must not be larger than 1 megabyte.
 - b. To create an S3 bucket and enter a **Bucket name** for storing your firmware image file, choose **Create S3 bucket**.

2. Permissions to access the bucket

You can either create a new service role or choose an existing role to allow AWS IoT Core for LoRaWAN to access the firmware image file in the S3 bucket on your behalf. Choose **Next**.

To create a new role, you can enter a role name or leave it blank for a random name to be generated automatically. To view the policy permissions that grant access to the S3 bucket, choose **View policy permissions**.

For more information about using an S3 bucket to store your image and granting AWS IoT Core for LoRaWAN permissions to access it, see [Upload the firmware file to an S3 bucket and add an IAM role \(p. 1037\)](#).

3. Review and create

To create your FUOTA task, review the FUOTA task and configuration details that you specified and then choose **Create task**.

Create FUOTA task and upload firmware image by using the API

To create a FUOTA task and specify your firmware image file by using the API, use the `CreateFuotaTask` API operation or the `create-fuota-task` CLI command. You can provide an `input.json` file as input to the `create-fuota-task` command. When you use the API or CLI, the firmware image file that you provide as input must be already uploaded to an S3 bucket. You also specify the IAM role that gives AWS IoT Core for LoRaWAN access to the firmware image in the S3 bucket.

```
aws iotwireless create-fuota-task \
--cli-input-json file://input.json
```

where:

Contents of `input.json`

```
{
  "Description": "FUOTA task to update firmware of devices in multicast group.",
  "FirmwareUpdateImage": "S3:/firmware_bucket/firmware_image",
  "FirmwareUpdateRole": "arn:aws:iam::123456789012:role/service-role/ACF1zBEI",
  "LoRaWAN": {
    "RfRegion": "US915"
  },
  "Name": "FUOTA_Task_MC"
}
```

After you create your FUOTA task, you can use the following API operations or CLI commands to update, delete, or get information about your FUOTA task.

- [UpdateFuotaTask](#) or `update-fuota-task`
- [GetFuotaTask](#) or `get-fuota-task`
- [ListFuotaTasks](#) or `list-fuota-tasks`
- [DeleteFuotaTask](#) or `delete-fuota-task`

Next steps

Now that you've created a FUOTA task and provided the firmware image, you can add devices to the task for updating their firmware. You can either add individual devices or multicast groups to the task. For more information, see [Add devices and multicast groups to a FUOTA task and schedule a FUOTA session \(p. 1062\)](#).

Add devices and multicast groups to a FUOTA task and schedule a FUOTA session

After you've created a FUOTA task, you can add devices to your task for which you want to update the firmware. After your devices have been added successfully to the FUOTA task, you can schedule a FUOTA session to update the device firmware.

- If you have only a small number of devices, you can add those devices directly to your FUOTA task.
- If you have a large number of devices that you want to update firmware for, you can add these devices to your multicast groups, and then add the multicast groups to your FUOTA task. For information about creating and using multicast groups, see [Create multicast groups to send a downlink payload to multiple devices \(p. 1047\)](#).

Note

You can add either individual devices or multicast groups to the FUOTA task. You can't add both devices and multicast groups to the task.

After you've added your devices or multicast groups, you can start a firmware update session. AWS IoT Core for LoRaWAN collects the firmware image, fragments the images, and then stores the fragments in an encrypted format. Your end devices collect the fragments and apply the new firmware image. The time that it takes for the firmware update depends on the image size and how the images were fragmented. After the firmware update is complete, the encrypted fragments of the firmware image stored by AWS IoT Core for LoRaWAN is deleted. You can still find the firmware image in the S3 bucket.

Prerequisites

Before you can add devices or multicast groups to your FUOTA task, do the following.

- You must have already created the FUOTA task and provided your firmware image. For more information, see [Create FUOTA task and provide firmware image \(p. 1060\)](#).
- Provision the wireless devices that you want to update the device firmware for. For more information about onboarding your device, see [Onboard your devices to AWS IoT Core for LoRaWAN \(p. 1010\)](#).
- To update the firmware of multiple devices, you can add them to a multicast group. For more information, see [Create multicast groups to send a downlink payload to multiple devices \(p. 1047\)](#).
- When you onboard the devices to AWS IoT Core for LoRaWAN, specify the FUOTA configuration parameter **FPorts**. If you're using a LoRaWAN v1.0.x device, you must also specify the **GenAppKey**. For more information about the FUOTA configuration parameters, see [Prepare devices for multicast and FUOTA configuration \(p. 1048\)](#).

Add devices to a FUOTA task and schedule a FUOTA session by using the console

To add devices or multicast groups and schedule a FUOTA session by using the console, go to the [FUOTA tasks](#) tab of the console. Then, choose the FUOTA task that you want to add devices to and perform the firmware update.

Add devices and multicast groups

1. You can add either individual devices or multicast groups to your FUOTA task. However, you can't add both individual devices and multicast groups to the same FUOTA task. To add devices using the console, do the following.
 1. In the **FUOTA task details**, choose **Add device**.
 2. Choose the frequency band or **RFRegion** for the devices you add to the task. This value must match the **RFRegion** that you chose for the FUOTA task.
 3. Choose whether you want to add individual devices or multicast groups to the task.
 - To add individual devices, choose **Add individual devices** and enter the device ID of each device that you want to add to your FUOTA task.
 - To add multicast groups, choose **Add multicast groups** and add your multicast groups to the task. You can filter the multicast groups you want to add to the task by using the device profile or tags. When you filter by device profile, you can choose multicast groups that with devices that have a profile with **Supports Class B** or **Supports Class C** enabled.

2. Schedule FUOTA session

After your devices or multicast groups have been added successfully, you can schedule a FUOTA session. To schedule a session, do the following.

1. Choose the FUOTA task for which you want to update the device firmware and then choose **Schedule FUOTA session**.
2. Specify a **Start date** and **Start time** for your FUOTA session. Make sure that the start time is 30 minutes or later from the current time.

Add devices to a FUOTA task and schedule a FUOTA session by using the API

You can use the AWS IoT Wireless API or the CLI to add your wireless devices or multicast groups to your FUOTA task. You can then schedule a FUOTA session.

1. Add devices and multicast groups

You can associate either wireless devices or multicast groups with your FUOTA task.

- To associate individual devices to your FUOTA task, use the [AssociateWirelessDeviceWithFuotaTask](#) API operation or the `associate-wireless-device-with-fuota-task` CLI command, and provide the `WirelessDeviceID` as input.

```
aws iotwireless associate-wireless-device-with-fuota-task \
    --id "01a23cde-5678-4a5b-ab1d-33456808ecb2"
    --wireless-device-id "ab0c23d3-b001-45ef-6a01-2bc3de4f5333"
```

- To associate multicast groups to your FUOTA task, use the [AssociateMulticastGroupWithFuotaTask](#) API operation or the `associate-multicast-group-with-fuota-task` CLI command, and provide the `MulticastGroupID` as input.

```
aws iotwireless associate-multicast-group-with-FUOTA-task \
    --id 01a23cde-5678-4a5b-ab1d-33456808ecb2"
    --multicast-group-id
```

After you've associated your wireless devices or multicast group to your FUOTA task, use the following API operations or CLI commands to list your devices or multicast groups or to disassociate them from your task.

- [DisassociateWirelessDeviceFromFuotaTask](#) or `disassociate-wireless-device-from-fuota-task`
- [DisassociateMulticastGroupFromFuotaTask](#) or `disassociate-multicast-group-from-fuota-task`
- [ListWirelessDevices](#) or `list-wireless-devices`
- [ListMulticastGroups](#) or `list-multicast-groups-by-fuota-task`

Note

The API:

- [ListWirelessDevices](#) can list wireless devices in general, and devices associated with a multicast group, when `MulticastGroupID` is used as the filter. The API lists wireless devices associated with a FUOTA task when `FuotaTaskID` is used as the filter.
- [ListMulticastGroups](#) can list multicast groups in general and multicast groups associated with a FUOTA task when `FuotaTaskID` is used as the filter.

2. Schedule FUOTA session

After your devices or multicast groups have been successfully added to the FUOTA task, you can start a FUOTA session to update the device firmware. The start time must be 30 minutes or later from the current time. To schedule a FUOTA session by using the API or CLI, use the [StartFuotaTask](#) API operation or the `start-fuota-task` CLI command.

After you've started a FUOTA session, You can no longer add devices or multicast groups to the task. You can get information about the status of your FUOTA session by using the [GetFuotaTask](#) API operation or the `get-fuota-task` CLI command.

Monitor and troubleshoot the status of your FUOTA task and devices added to the task

After you have provisioned the wireless devices and created any multicast groups that you might want to use, you can start a FUOTA session by performing the following steps.

FUOTA task status

Your FUOTA task can have one of the following status messages displayed in the AWS Management Console.

- **Pending**

This status indicates that you've created a FUOTA task, but it doesn't yet have a firmware update session. You'll see this status message displayed when your task has been created. During this time, you can update your FUOTA task, and associate or disassociate devices or multicast groups with your task. After the status changes from **Pending**, additional devices can't be added to the task.

- **FUOTA session waiting**

After your devices have been added successfully to the FUOTA task, when your task has a scheduled firmware update session, you'll see this status message displayed. During this time, you can't update or add devices to your FUOTA session. If you cancel your FUOTA session, the group status changes to **Pending**.

- **In FUOTA session**

When your FUOTA session begins, you'll see this status message displayed. The fragmentation session starts and your end devices collect the fragments, reconstruct the firmware image, compare the new firmware version with the original version, and apply the new image.

- **FUOTA done**

After your end devices report to AWS IoT Core for LoRaWAN that the new firmware image has been applied, or when the session times out, the FUOTA session is marked as done, and you'll see this status displayed.

You'll also see this status displayed in any of the following cases so be sure to check whether the firmware update was applied correctly to the devices.

- When the FUOTA task status was **FUOTA session waiting**, and there's an S3 bucket error, such as the link to the image file in the S3 bucket is incorrect or AWS IoT Core for LoRaWAN doesn't have sufficient permissions to access the file in the bucket.
- When the FUOTA task status was **FUOTA session waiting**, and there's a request to start a FUOTA session, but a response isn't received from the devices or multicast groups in your FUOTA task.
- When the FUOTA task status was **In FUOTA session**, and the devices or multicast groups haven't sent any fragments for a certain time period, which results in the session to timeout.

- **Delete waiting**

If you delete your FUOTA task that's in any of the other states, you'll see this status displayed. A deletion action is permanent and can't be undone. This action can take time and the task status will be **Delete waiting** until the FUOTA task has been deleted. After your FUOTA task enters this state, it can't transition to one of the other states.

Status of devices in a FUOTA task

The devices in your FUOTA task can have one of the following status messages displayed in the AWS Management Console. You can hover over each status message to get more information about what it indicates.

- **Initial**

When it's the start time of your FUOTA session, AWS IoT Core for LoRaWAN checks whether your device has the supported package for the firmware update. If your device has the supported package, the FUOTA session for the device starts. The firmware image is fragmented and the fragments are sent to your device. When you see this status displayed, it indicates that the FUOTA session for the device hasn't started yet.

- **Package unsupported**

If the device doesn't have the supported FUOTA package, you'll see this status displayed. If the firmware update package isn't supported, the FUOTA session for your device can't start. To resolve this error, check whether your device's firmware can receive firmware updates using FUOTA.

- **Fragmentation algorithm unsupported**

At the start of your FUOTA session, AWS IoT Core for LoRaWAN sets up a fragmentation session for your device. If you see this status displayed, it means that the type of fragmentation algorithm used can't be applied for your device's firmware update. The error occurs because your device doesn't have the supported FUOTA package. To resolve this error, check whether your device's firmware can receive firmware updates using FUOTA.

- **Not enough memory**

After AWS IoT Core for LoRaWAN sends the image fragments, your end devices collect the image fragments and reconstruct the binary image from these fragments. This status is displayed when your device doesn't have enough memory to assemble the incoming fragments of the firmware image, which can result in your firmware update session ending prematurely. To resolve the error, check whether your device's hardware can receive this update. If your device can't receive this update, use a delta image to update the firmware.

- **Fragmentation index unsupported**

The fragmentation index identifies one of the four simultaneously possible fragmentation sessions. If your device doesn't support the indicated fragmentation index value, this status is displayed. To resolve this error, do one or more of the following.

- Start a new FUOTA task for the device.
- If the error persists, switch from unicast to multicast mode.
- If the error still isn't resolved, check your device firmware.

- **Memory error**

This status indicates that your device has experienced a memory error when receiving the incoming fragments from AWS IoT Core for LoRaWAN. If this error occurs, your device might not be capable of receiving this update. To resolve the error, check whether your device's hardware can receive this update. If needed, use a delta image to update the device firmware.

- **Wrong descriptor**

Your device doesn't support the indicated descriptor. The descriptor is a field that describes the file that will be transported during the fragmentation session. If you see this error, contact [AWS Support Center](#).

- **Session count replay**

This status indicates that your device has previously used this session count. To resolve the error, start a new FUOTA task for the device.

- **Missing fragments**

As your device collects the image fragments from AWS IoT Core for LoRaWAN, it reconstructs the new firmware image from the independent, coded fragments. If your device hasn't received all the fragments, the new image can't be reconstructed, and you'll see this status. To resolve the error, start a new FUOTA task for the device.

- **MIC error**

When your device reconstructs the new firmware image from the collected fragments, it performs a MIC (Message Integrity Check) to verify the authenticity of your image and whether it's coming from the right source. If your device detects a mismatch in the MIC after reassembling the fragments, this status is displayed. To resolve the error, start a new FUOTA task for the device.

- **Successful**

The FUOTA session for your device was successful.

Note

While this status message indicates that the devices have reconstructed the image from the fragments and verified it, the device firmware might not have been updated when the device reports the status to AWS IoT Core for LoRaWAN. Check whether your device firmware has been updated.

Next steps

You've learned about the different statuses of the FUOTA task and its devices and how you can troubleshoot any issues. For more information about each of these statuses, see the [LoRaWAN Fragmented Data Block Transportation Specification, TS004-1.0.0](#).

Monitoring your wireless resource fleet in real time using network analyzer

Network analyzer uses a default WebSocket connection to receive real-time trace message logs for your wireless connectivity resources. By using network analyzer, you can add the resources you want to monitor, activate a trace messaging session, and start receiving trace messages in real time.

To monitor your resources, you can also use Amazon CloudWatch. To use CloudWatch, you set up an IAM role to configure logging and then wait for the log entries to be displayed in the console. Network analyzer significantly reduces the time that it takes to set up a connection and start receiving trace messages, providing you with just-in-time log information for your fleet of resources. For information about monitoring by using CloudWatch, see [Monitoring and logging for AWS IoT Core for LoRaWAN using Amazon CloudWatch \(p. 1078\)](#).

By reducing your setup time and using the information from the trace messages, you can monitor your resources more effectively, get meaningful insights, and troubleshoot errors. You can monitor both LoRaWAN devices and LoRaWAN gateways. For example, you can quickly identify a join error when

onboarding one of your LoRaWAN devices. To debug the error, use the information in the provided trace message log.

How to use network analyzer

To monitor your resource fleet and start receiving trace messages, perform the following steps

1. Add resources and specify configuration settings for your configuration

Add resources to your network analyzer configuration by entering the wireless gateway and wireless device identifiers. You can also edit the configuration settings, log levels and wireless device frame information, for your network analyzer configuration.

2. Stream network analyzer trace messages with WebSockets

You can generate a presigned request URL using the credentials for your IAM role to stream network analyzer trace messages by using the WebSocket protocol.

3. Activate your trace messaging session and monitor trace messages

To start receiving trace messages, activate your trace messaging session. To avoid incurring additional costs, you can either deactivate or close your network analyzer trace messaging session.

The following shows how to add resources and activate your trace messaging session.

Topics

- [Add resources and update the network analyzer configuration \(p. 1068\)](#)
- [Stream network analyzer trace messages with WebSockets \(p. 1071\)](#)
- [View and monitor network analyzer trace message logs in real time \(p. 1075\)](#)

Add resources and update the network analyzer configuration

Before you can activate trace messaging, you must add resources to your configuration. You can use only a single, default network analyzer configuration. AWS IoT Core for LoRaWAN assigns the name, **NetworkAnalyzerConfig_Default**, to this configuration, and this field can't be edited. This configuration is automatically added to your AWS account when you use network analyzer from the console.

You can add the resources that you want to monitor to this default configuration. Resources can be either or both LoRaWAN devices and LoRaWAN gateways. To add each individual resource to the configuration, use the wireless gateway and wireless device identifiers.

Configuration settings

To configure settings, first add resources to your default configuration and activate trace messaging. After you've received the trace message logs, you can also customize the following parameters to update your default configuration and filter the log stream.

- **Frame info**

This setting is the frame info of your wireless device resources for trace messages. The frame info is enabled by default, and can be used to debug the communication between your network server and the end devices.

- **Log levels**

You can view Info or Error logs, or you can turn off logging.

- **Info**

Logs with a log level of **Info** are more verbose and contain log streams that are both informative and contain errors. The informative logs can be used to view changes to the state of a device or gateway.

Note

Collecting more verbose log streams can incur additional costs. For more information about pricing, see [AWS IoT Core pricing](#).

- **Error**

Logs with a log level of **Error** are less verbose and display only error information. You can use these logs when an application has an error, such as a device connection error. By using the information from the log stream, you can identify and troubleshoot errors for resources in your fleet.

Prerequisites

Before you can add resources, you must have onboarded the gateways and devices that you want to monitor to AWS IoT Core for LoRaWAN. For more information, see [Connecting gateways and devices to AWS IoT Core for LoRaWAN \(p. 1002\)](#).

Add resources and update the network analyzer configuration by using the console

You can add resources and customize the optional parameters by using the AWS IoT console or the AWS IoT Wireless API. In addition to resources, you can also edit your configuration settings and save the updated configuration.

To add resources to your configuration (console)

1. Open the [Network Analyzer hub of the AWS IoT console](#) and choose the network analyzer configuration, **NetworkAnalyzerConfig_Default**.
2. Choose **Add resources**.
3. Add the resources you want to monitor by using the wireless gateway and wireless device identifiers. You can add up to 250 wireless gateways or wireless devices. To add your resource:
 - a. Use the **View gateways** or **View devices** tab to see the list of gateways and devices that you've added to your AWS account.
 - b. Copy the **WirelessDeviceID** or the **WirelessGatewayID** of the device or gateway that you want to monitor and enter the identifier value for the corresponding resource.
 - c. To continue adding resources, choose **Add gateway** or **Add device** and add your wireless gateway or device. If you added a resource that you no longer want to monitor, choose **Remove resource**.
4. After you've added all the resources, choose **Add**.

You'll see the number of gateways and devices that you added in the [Network Analyzer hub page](#). You can still continue adding gateways and devices until you activate the trace messaging session. After the session has been activated, to add resources, you'll have to deactivate the session.

To edit the network analyzer configuration (console)

You can also edit the network analyzer configuration and choose whether to disable frame info and the log level for your trace message logs.

1. Open the [Network Analyzer hub of the AWS IoT console](#) and choose the network analyzer configuration, **NetworkAnalyzerConfig_Default**.

2. Choose **Edit**.
3. Choose whether to disable frame info and use **Select log levels** to choose the log levels that you want to use for your trace message logs. Choose **Save**.

You'll see the configuration settings that you specified in the details page of your network analyzer configuration.

Add resources and update the network analyzer configuration by using the API

You can use the [AWS IoT Wireless API operations](#) or the [AWS IoT Wireless CLI commands](#) to add resources and update the configuration settings for your network analyzer configuration.

- To add resources and update your network analyzer configuration, use the [UpdateNetworkAnalyzerConfiguration API](#) or the [update-network-analyzer-configuration CLI](#).
 - **Add resources**

For the wireless devices you want to add, use `WirelessDevicesToAdd` to enter the `WirelessDeviceID` for the devices as an array of strings. For the wireless gateways you want to add, use `WirelessGatewaysToAdd` to enter the `WirelessGatewayID` for the gateways as an array of strings.

- **Edit configuration**

To edit your network analyzer configuration, use the `TraceContent` parameter to specify whether `WirelessDeviceFrameInfo` should be `ENABLED` or `DISABLED`, and whether the `LogLevel` parameter should be `INFO`, `ERROR`, or `DISABLED`.

```
{  
    "TraceContent": {  
        "LogLevel": "string",  
        "WirelessDeviceFrameInfo": "string"  
    },  
    "WirelessDevicesToAdd": [ "string" ],  
    "WirelessDevicesToRemove": [ "string" ],  
    "WirelessGatewaysToAdd": [ "string" ],  
    "WirelessGatewaysToRemove": [ "string" ]  
}
```

- To get information about the configuration and the resources that you've added, use the [GetNetworkAnalyzerConfiguration API operation](#) or the [get-network-analyzer-configuration command](#). Provide the name of the network analyzer configuration, `NetworkAnalyzerConfig_Default`, as input.

Next steps

Now that you've added resources and specified any optional configuration settings for your configuration, you can use the WebSocket protocol to establish a connection with AWS IoT Core for LoRaWAN for using network analyzer. You can then activate trace messaging and start receiving trace messages for your resources. For more information, see [Stream network analyzer trace messages with WebSockets \(p. 1071\)](#).

Stream network analyzer trace messages with WebSockets

When you use the WebSocket protocol, you can stream network analyzer trace messages in real time. When you send a request, the service responds with a JSON structure. After you activate trace messaging, you can use the message logs to get information about your resources and troubleshoot errors. For more information, see [WebSocket protocol](#).

The following shows how to stream network analyzer trace messages with WebSockets.

Topics

- [Generate a presigned request with the WebSocket library \(p. 1071\)](#)
- [WebSocket messages and status codes \(p. 1074\)](#)

Generate a presigned request with the WebSocket library

The following shows how you to generate a presigned request so that you can use the WebSocket library to send requests to the service.,

Add a policy for WebSocket requests to your IAM role

To use the WebSocket protocol to call network analyzer, attach the following policy to the AWS Identity and Access Management (IAM) role that makes this request.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": "iotwireless:StartNetworkAnalyzerStream",  
            "Resource": "*"  
        }  
    ]  
}
```

Create a presigned URL

Construct a URL for your WebSocket request that contains the information needed to set up communication between your application and the network analyzer. To verify the identity of the request, WebSocket streaming uses the Amazon Signature Version 4 process for signing requests. For more information about Signature Version 4, see [Signing AWS API Requests](#) in the *Amazon Web Services General Reference*.

To call network analyzer, use the `StartNetworkAnalyzerStream` request URL. The request will be signed using the credentials for the IAM role mentioned previously. The URL has the following format with line breaks added for readability.

```
GET wss://api.iotwireless.<region>.amazonaws.com/start-network-analyzer-stream?X-Amz-  
Algorithm=AWS4-HMAC-SHA256  
&X-Amz-Credential=Signature Version 4 credential scope  
&X-Amz-Date=date  
&X-Amz-Expires=time in seconds until expiration  
&X-Amz-Security-Token=security-token  
&X-Amz-Signature=Signature Version 4 signature  
&X-Amz-SignedHeaders=host
```

Use the following values for the Signature Version 4 parameters:

- **X-Amz-Algorithm** – The algorithm you're using in the signing process. The only valid value is AWS4-HMAC-SHA256.
- **X-Amz-Credential** – A string separated by slashes ("/") that is formed by concatenating your access-key ID and your credential scope components. Credential scope includes the date in YYYYMMDD format, the AWS Region, the service name, and a termination string (aws4_request).
- **X-Amz-Date** – The date and time that the signature was created. Generate the date and time by following the instructions in [Handling Dates in Signature Version 4](#) in the *Amazon Web Services General Reference*.
- **X-Amz-Expires** – The length of time in seconds until the credentials expire. The maximum value is 300 seconds (5 minutes).
- **X-Amz-Security-Token** – (optional) A Signature Version 4 token for temporary credentials. If you specify this parameter, include it in the canonical request. For more information, see [Requesting Temporary Security Credentials](#) in the *AWS Identity and Access Management User Guide*.
- **X-Amz-Signature** – The Signature Version 4 signature that you generated for the request.
- **X-Amz-SignedHeaders** – The headers that are signed when creating the signature for the request. The only valid value is host.

Construct the request URL and create Signature Version 4 signature

To construct the URL for the request and create the Signature Version 4 signature, use the following steps. The examples are in pseudocode.

Task 1: Create a canonical request

Create a string that includes information from your request in a standardized format. This ensures that when AWS receives the request, it can calculate the same signature that you calculate in [Task 3: Calculate the signature \(p. 1074\)](#). For more information, see [Create a Canonical Request for Signature Version 4](#) in the *Amazon Web Services General Reference*.

1. Define variables for the request in your application.

```
# HTTP verb
method = "GET"
# Service name
service = "iotwireless"
# AWS Region
region = "AWS Region"
# Service streaming endpoint
endpoint = "wss://api.iotwireless.region.amazonaws.com"
# Host
host = "api.iotwireless.<b>region</b>.amazonaws.com"
# Date and time of request
amz-date = YYYYMMDD'T'HHMMSS'Z'
# Date without time for credential scope
datestamp = YYYYMMDD
```

2. Create a canonical URI (uniform resource identifier). The canonical URI is the part of the URI between the domain and the query string.

```
canonical_uri = "/start-network-analyzer-stream"
```

3. Create the canonical headers and signed headers. Note the trailing \n in the canonical headers.

- Append the lowercase header name followed by a colon.
- Append a comma-separated list of values for that header. Don't sort the values in headers that have multiple values.

- Append a new line (\n).

```
canonical_headers = "host:" + host + "\n"
signed_headers = "host"
```

4. Match the algorithm to the hashing algorithm. You must use SHA-256.

```
algorithm = "AWS4-HMAC-SHA256"
```

5. Create the credential scope, which scopes the derived key to the date, Region, and service to which the request is made.

```
credential_scope = datestamp + "/" + region + "/" + service + "/" + "aws4_request"
```

6. Create the canonical query string. Query string values must be URI-encoded and sorted by name.

- Sort the parameter names by character code point in ascending order. Parameters with duplicate names should be sorted by value. For example, a parameter name that begins with the uppercase letter F precedes a parameter name that begins with a lowercase letter b.
- Do not URI-encode any of the unreserved characters that [RFC 3986](#) defines: A-Z, a-z, 0-9, hyphen (-), underscore (_), period (.), and tilde (~).
- Percent-encode all other characters with %XY, where X and Y are hexadecimal characters (0-9 and uppercase A-F). For example, the space character must be encoded as %20 (not using '+', as some encoding schemes do) and extended UTF-8 characters must be in the form %XY%ZA%BC.
- Double-encode any equals (=) characters in parameter values.

```
canonical_querystring = "X-Amz-Algorithm=" + algorithm
canonical_querystring += "&X-Amz-Credential=" + URI-encode(access_key + "/" +
    credential_scope)
canonical_querystring += "&X-Amz-Date=" + amz_date
canonical_querystring += "&X-Amz-Expires=300"
canonical_querystring += "&X-Amz-Security-Token=" + token
canonical_querystring += "&X-Amz-SignedHeaders=" + signed_headers
canonical_querystring += "&language-code=en-US&media-encoding=pcm&sample-rate=16000"
```

7. Create a hash of the payload. For a GET request, the payload is an empty string.

```
payload_hash = HashSHA256("").Encode("utf-8")).HexDigest()
```

8. Combine all of the elements to create the canonical request.

```
canonical_request = method + '\n'
+ canonical_uri + '\n'
+ canonical_querystring + '\n'
+ canonical_headers + '\n'
+ signed_headers + '\n'
+ payload_hash
```

Task 2: Create the string to sign

The string to sign contains meta information about your request. You use the string to sign in the next step when you calculate the request signature. For more information, see [Create a String to Sign for Signature Version 4](#) in the *Amazon Web Services General Reference*.

```
string_to_sign=algorithm + "\n"
+ amz_date + "\n"
+ credential_scope + "\n"
+ HashSHA256(canonical_request.Encode("utf-8")).HexDigest()
```

Task 3: Calculate the signature

You derive a signing key from your AWS secret access key. For a greater degree of protection, the derived key is specific to the date, service, and AWS Region. You use the derived key to sign the request. For more information, see [Calculate the Signature for AWS Signature Version 4](#) in the *Amazon Web Services General Reference*.

The code assumes that you have implemented the `GetSignatureKey` function to derive a signing key. For more information and example functions, see [Examples of How to Derive a Signing Key for Signature Version 4](#) in the *Amazon Web Services General Reference*.

The function `HMAC(key, data)` represents an HMAC-SHA256 function that returns the results in binary format.

```
#Create the signing key
signing_key = GetSignatureKey(secret_key, datetamp, region, service)

# Sign the string_to_sign using the signing key
signature = HMAC.new(signing_key, (string_to_sign).Encode("utf-8"), Sha256()).HexDigest
```

Task 4: Add signing information to request and create request URL

After you calculate the signature, add it to the query string. For more information, see [Add the Signature to the Request](#) in the *Amazon Web Services General Reference*.

```
#Add the authentication information to the query string
canonical_querystring += "&X-Amz-Signature=" + signature

# Sign the string_to_sign using the signing key
request_url = endpoint + canonical_uri + "?" + canonical_querystring
```

Next steps

You can now use the request URL with your WebSocket library to make the request to the service and observe the messages. For more information, see [WebSocket messages and status codes \(p. 1074\)](#).

WebSocket messages and status codes

After you've created a presigned request, you can use the request URL with your WebSocket library, or a library that's suited to your programming language, to make requests to the service. For more information about how you can generate this presigned request, see [Generate a presigned request with the WebSocket library \(p. 1071\)](#).

WebSocket messages

The WebSocket protocol can be used to establish a bi-directional connection. Messages can be transmitted from client to server and from server to client. However, network analyzer supports only messages that are sent from server to client. Any message received from the client is unexpected and the server will automatically close the WebSocket connection if a message is received from client.

When the request is received and a trace messaging session has started, the server responds with a JSON structure, which is the payload. For more information about the payload and how you can activate trace messaging from the AWS Management Console, see [View and monitor network analyzer trace message logs in real time \(p. 1075\)](#).

WebSocket status codes

The following shows the WebSocket status codes for the communication from the server to client. The WebSocket status codes follow the [RFC Standard of Normal closure of connections](#).

The following shows the supported status codes:

- **1000**

This status code indicates a normal closure, which means that the WebSocket connection has been established and the request has been fulfilled. This status can be observed when a session is idle, causing the connection to time out.

- **1002**

This status code indicates that the endpoint is terminating the connection because of a protocol error.

- **1003**

This status code indicates an error status where the endpoint terminated the connection because it received data in a format that it can't accept. The endpoint supports only text data and might display this status code if it receives a binary message or a message from the client that's using an unsupported format.

- **1008**

This status code indicates an error status where the endpoint terminated the connection because it received a message that violates its policy. This status is generic and is displayed when the other status codes, such as 1003 or 1009, aren't applicable. You'll also see this status displayed if there's a need to hide the policy, or when there's an authorization failure, such as an expired signature.

- **1011**

This status code indicates an error status where the server is terminating the connection because it encountered an unexpected condition or internal error that prevented it from fulfilling the request.

Next steps

Now that you've learned how to generate a presigned request and how you can observe messages from the server by using the WebSocket connection, you can activate trace messaging and start receiving message logs for your wireless gateway and wireless device resources. For more information, see [View and monitor network analyzer trace message logs in real time \(p. 1075\)](#).

View and monitor network analyzer trace message logs in real time

If you've added resources to your network analyzer configuration, you can activate trace messaging to start receiving trace messages for your resources. You can use either the AWS Management Console, the AWS IoT Wireless API, or the AWS CLI.

Prerequisites

Before you can activate trace messaging by using network analyzer, you must have:

- Added the resources that you want to monitor to your default network analyzer configuration. For more information, see [Add resources and update the network analyzer configuration \(p. 1068\)](#).
- Generated a presigned request by using the `StartNetworkAnalyzerStream` request URL. The request will be signed using the credentials for the AWS Identity and Access Management role that makes this request. For more information, see [Create a presigned URL \(p. 1071\)](#)

Activate trace messaging by using the console

To activate trace messaging

1. Open the [Network Analyzer hub of the AWS IoT console](#) and choose your network analyzer configuration, **NetworkAnalyzerConfig_Default**.
2. In the details page of your network analyzer configuration, choose **Activate trace messaging** and then choose **Activate**.

You'll start receiving trace messages where the newest trace message appears first in the console.

Note

After the messaging session starts, receiving trace messages can incur additional costs until you deactivate the session or leave the trace session. For more information about pricing, see [AWS IoT Core pricing](#).

View and monitor trace messages

After you activate trace messaging, the WebSocket connection is established and trace messages start appearing in real time, newest first. You can customize the preferences to specify the number of trace messages to be displayed in each page and to display only the relevant fields for each message. For example, you can customize the trace message log to show only logs for wireless gateway resources that have **Log level** set to **ERROR**, so that you can quickly identify and debug errors with your gateways. The trace messages contain the following information.

- **Message Number:** A unique number that shows the last message received first.
- **Resource ID:** The wireless gateway or wireless device ID of the resource.
- **Timestamp:** The time when the message was received.
- **Message ID:** An identifier that AWS IoT Core for LoRaWAN assigns to each received message.
- **FPort:** The frequency port for communicating with the device by using the WebSocket connection.
- **DevEui:** The extended unique identifier (EUI) for your wireless device.
- **Resource:** Whether the monitored resource is a wireless device or a wireless gateway.
- **Event:** The event for a log message for a wireless device, which can be **Join**, **Rejoin**, **Uplink_Data**, **Downlink_Data**, or **Registration**.
- **Log level:** Information about INFO or ERROR log streams for your device.

Network analyzer JSON log message

You can also choose one trace message at a time to view the JSON payload for that message. Depending on the message that you select in the trace message logs, you'll see information in the JSON payload that indicates contains 2 parts: **CustomerLog** and **LoRaFrame**.

CustomerLog

The **CustomerLog** part of the JSON displays the type and identifier of the resource that received the message, the log level, and the message content. The following example shows a **CustomerLog** log

message. You can use the `message` field in the JSON to get more information about the error and how it can be resolved.

LoRaFrame

The **LoRaFrame** part of the JSON has a **Message ID** and contains information about the physical payload for the device and the wireless metadata.

The following shows the structure of the trace message.

```
export type TraceMessage = {
  ResourceId: string;
  Timestamp: string;
  LoRaFrame:
  {
    MessageId: string;
    PhysicalPayload: any;
    WirelessMetadata:
    {
      fPort: number;
      dataRate: number;
      devEui: string;
      frequency: number,
      timestamp: string;
    },
  }
  CustomerLog:
  {
    resource: string;
    wirelessDeviceId: string;
    wirelessDeviceType: string;
    event: string;
    logLevel: string;
    messageId: string;
    message: string;
  },
};
```

Review and next steps

In this section, you've viewed trace messages and learned how you can use the information to debug errors. After you've viewed all messages, you can:

- **Deactivate trace messaging**

To avoid incurring any additional costs, you can deactivate the trace messaging session. Deactivating the session disconnects your WebSocket connection so you won't receive any additional trace messages. You can still continue viewing the existing messages in the console.

- **Edit frame info for your configuration**

You can edit the network analyzer configuration and choose whether to deactivate frame info and choose the log levels for your messages. Before you update your configuration, consider deactivating your trace messaging session. To make these edits, open the [Network Analyzer details page in the AWS IoT console](#) and choose **Edit**. You can then update your configuration with the new configuration settings and activate trace messaging to see the updated messages.

- **Add resources to your configuration**

You can also add more resources to your network analyzer configuration and monitor them in real time. You can add up to a combined total of 250 wireless gateway and wireless device resources. To add resources, on the [Network Analyzer details page of the AWS IoT console](#), choose the **Resources**

tab and choose **Add resources**. You can then update your configuration with the new resources and activate trace messaging to see the updated messages for the additional resources.

For more information about updating your network analyzer configuration by editing the configuration settings and adding resources, see [Add resources and update the network analyzer configuration \(p. 1068\)](#).

Monitoring and logging for AWS IoT Core for LoRaWAN using Amazon CloudWatch

You can monitor your AWS IoT Core for LoRaWAN resources and applications that run in real time by using Amazon CloudWatch. This section also contains information about how you can monitor the status of any Sidewalk devices that you've onboarded using Amazon CloudWatch. For information about onboarding Amazon Sidewalk devices to AWS IoT Core, see [Getting started with Amazon Sidewalk Integration for AWS IoT Core \(p. 1099\)](#).

Use CloudWatch to collect and track metrics, which are variables that you can measure for your resources and applications. For more information about the benefits of using monitoring, see [Monitoring AWS IoT \(p. 400\)](#).

If you want to obtain more real-time log information from your devices, use the network analyzer. For more information, see [Monitoring your wireless resource fleet in real time using network analyzer \(p. 1067\)](#).

How to monitor your AWS IoT Core for LoRaWAN resources

To log and monitor your wireless resources, perform the following steps.

1. Create a logging role to log your AWS IoT Core for LoRaWAN resources, as described in [Create logging role and policy for AWS IoT Core for LoRaWAN \(p. 1079\)](#).
2. Log messages in the CloudWatch Logs console have a default log level of `ERROR`, which is less verbose and contains only error information. If you want to view more verbose messages, we recommend that you use the CLI to configure logging first, as described in [Configure logging for AWS IoT Core for LoRaWAN resources \(p. 1081\)](#).
3. Next, you can monitor your resources by viewing the log entries in the CloudWatch Logs console. For more information, see [View CloudWatch AWS IoT Core for LoRaWAN log entries \(p. 1089\)](#).
4. You can create filter expressions by using **Logs groups** but we recommend that you first create simple filters and view log entries in the log groups, and then go to CloudWatch Insights to create queries to filter the log entries depending on the resource or event you're monitoring. For more information, see [Use CloudWatch Insights to filter logs for AWS IoT Core for LoRaWAN \(p. 1095\)](#).

The following topics show how to configure logging for AWS IoT Core for LoRaWAN and to collect metrics from CloudWatch. In addition to LoRaWAN devices, you can use these topics to configure logging for any Sidewalk devices that you've added to your account and monitor them. For information about how to add these devices, see [Amazon Sidewalk Integration for AWS IoT Core \(p. 1099\)](#).

Topics

- [Configure Logging for AWS IoT Core for LoRaWAN \(p. 1079\)](#)
- [Monitor AWS IoT Core for LoRaWAN using CloudWatch Logs \(p. 1089\)](#)

Configure Logging for AWS IoT Core for LoRaWAN

Before you can monitor and log AWS IoT activity, first enable logging for AWS IoT Core for LoRaWAN resources by using either the CLI or API.

When considering how to configure your AWS IoT Core for LoRaWAN logging, the default logging configuration determines how AWS IoT activity will be logged unless you specify otherwise. Starting out, you might want to obtain detailed logs with a default log level of `INFO`.

After reviewing the initial logs, you can change the default log level to `ERROR`, which is less verbose, and set a more verbose, resource-specific log level on resources that might need more attention. Log levels can be changed whenever you want.

The following topics show how to configure logging for AWS IoT Core for LoRaWAN resources.

Topics

- [Create logging role and policy for AWS IoT Core for LoRaWAN \(p. 1079\)](#)
- [Configure logging for AWS IoT Core for LoRaWAN resources \(p. 1081\)](#)

Create logging role and policy for AWS IoT Core for LoRaWAN

The following shows how to create a logging role for only AWS IoT Core for LoRaWAN resources. If you want to also create a logging role for AWS IoT Core, see [Create a logging role \(p. 401\)](#).

Create a logging role for AWS IoT Core for LoRaWAN

Before you can enable logging, you must create an IAM role and a policy that gives AWS permission to monitor AWS IoT Core for LoRaWAN activity on your behalf.

Create IAM role for logging

To create a logging role for AWS IoT Core for LoRaWAN, open the [Roles hub of the IAM console](#) and choose **Create role**.

1. Under **Select type of trusted entity**, choose **Another AWS account**.
2. In **Account ID**, enter your AWS account ID, and then choose **Next: Permissions**.
3. In the search box, enter `AWSIoTWirelessLogging`.
4. Select the box next to the policy named `AWSIoTWirelessLogging`, and then choose **Next: Tags**.
5. Choose **Next: Review**.
6. In **Role name**, enter `IoTWirelessLogsRole`, and then choose **Create role**.

Edit trust relationship of the IAM role

In the confirmation message displayed after you ran the previous step, choose the name of the role you created, `IoTWirelessLogsRole`. Next, you'll edit the role to add the following trust relationship.

1. In the **Summary** section of the role `IoTWirelessLogsRole`, choose the **Trust relationships** tab, and then choose **Edit trust relationship**.
2. In **Policy Document**, change the **Principal** property to look like this example.

```
"Principal": {  
    "Service": "iotwireless.amazonaws.com"  
},
```

After you change the `Principal` property, the complete policy document should look like this example.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Principal": {  
                "Service": "iotwireless.amazonaws.com"  
            },  
            "Action": "sts:AssumeRole",  
            "Condition": {}  
        }  
    ]  
}
```

3. To save your changes and exit, choose **Update Trust Policy**.

Logging policy for AWS IoT Core for LoRaWAN

The following policy document provides the role policy and trust policy that allows AWS IoT Core for LoRaWAN to submit log entries to CloudWatch on your behalf.

Note

This AWS managed policy document was automatically created for you when you created the logging role, **IoTWirelessLogsRole**.

Role policy

The following shows the role policy document.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "logs>CreateLogGroup",  
                "logs>CreateLogStream",  
                "logs>DescribeLogGroups",  
                "logs>DescribeLogStreams",  
                "logs>PutLogEvents"  
            ],  
            "Resource": "arn:aws:logs:*:log-group:/aws/iotwireless*"  
        }  
    ]  
}
```

Trust policy to log only AWS IoT Core for LoRaWAN activity

The following shows the trust policy for logging only AWS IoT Core for LoRaWAN activity.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "",  
            "Effect": "Allow",  
            "Principal": {  
                "Service": "iotwireless.amazonaws.com"  
            },  
            "Action": "logs>PutLogEvents",  
            "Resource": "arn:aws:logs:*:log-group:/aws/iotwireless*"  
        }  
    ]  
}
```

```
        "Service": [
            "iotwireless.amazonaws.com"
        ],
        "Action": "sts:AssumeRole"
    ]
}
```

If you created the IAM role to also log AWS IoT Core activity, then the policy documents allow you to log both activities. For information about creating a logging role for AWS IoT Core, see [Create a logging role \(p. 401\)](#).

Next steps

You've learned how to create a logging role to log your AWS IoT Core for LoRaWAN resources. By default, logs have a log level of `ERROR`, so if you want to see only error information, go to [View CloudWatch AWS IoT Core for LoRaWAN log entries \(p. 1089\)](#) to monitor your wireless resources by viewing the log entries.

If you want more information in the log entries, you can configure the default log level for your resources or for different event types, such as setting the log level to `INFO`. For information about configuring logging for your resources, see [Configure logging for AWS IoT Core for LoRaWAN resources \(p. 1081\)](#).

Configure logging for AWS IoT Core for LoRaWAN resources

To configure logging for AWS IoT Core for LoRaWAN resources, you can use either the API or the CLI. When starting to monitor AWS IoT Core for LoRaWAN resources, you can use the default configuration. To do this, you can skip this topic and proceed to [Monitor AWS IoT Core for LoRaWAN using CloudWatch Logs \(p. 1089\)](#) to monitor your logs.

After you start monitoring the logs, you can use the CLI to change the log levels to a more verbose option, such as providing `INFO` and `ERROR` information and enabling logging for more resources.

AWS IoT Core for LoRaWAN resources and log levels

Before you use the API or CLI, use the following table to learn about the different log levels and the resources that you can configure logging for. The table shows parameters that you see in the CloudWatch logs when you monitor the resources. How you configure the logging for your resources will determine the logs you see in the console.

For information about what a sample CloudWatch logs looks like and how you can use these parameters to log useful information about the AWS IoT Core for LoRaWAN resources, see [View CloudWatch AWS IoT Core for LoRaWAN log entries \(p. 1089\)](#).

Log levels and resources

Name	Possible values	Description
<code>logLevel</code>	<code>INFO</code> , <code>ERROR</code> , or <code>DISABLED</code>	<ul style="list-style-type: none"><code>ERROR</code>: Displays any error that causes an operation to fail. Logs include only <code>ERROR</code> information.<code>INFO</code>: Provides high-level information about the flow of things. Logs include <code>INFO</code> and <code>ERROR</code> information.<code>DISABLED</code>: Disables all logging.

Name	Possible values	Description
resource	WirelessGateway or WirelessDevice	The type of the resource, which can be WirelessGateway or WirelessDevice.
wirelessGatewayType	LoRaWAN	The type of the wireless gateway, when resource is WirelessGateway, which is always LoRaWAN.
wirelessDeviceType	LoRaWAN or Sidewalk	The type of the wireless device, when resource is WirelessDevice, which can be LoRaWAN or Sidewalk.
wirelessGatewayId	-	The identifier of the wireless gateway, when resource is WirelessGateway.
wirelessDeviceId	-	The identifier of the wireless device, when resource is WirelessDevice.
event	Join, Rejoin, Registration, Uplink_data, Downlink_data, CUPS_Request, and Certificate	The type of event being logged, which depends on whether the resource that you're logging is a wireless device or a wireless gateway. For more information, see View CloudWatch AWS IoT Core for LoRaWAN log entries (p. 1089) .

AWS IoT Wireless logging API

You can use the following API actions to configure logging of resources. The table also shows a sample IAM policy that you must create for using the API actions. The following section describes how you can use the APIs to configure log levels of your resources.

Logging API actions

API name	Description	Sample IAM policy
GetLogLevelsByResourceTypes	Returns current default log levels, or log levels by resource types, which can include log options for wireless devices or wireless gateways.	<pre>{ "Version": "2012-10-17", "Statement": [{ "Effect": "Allow", "Action": ["iotwireless:GetLogLevelsByResourceTypes"], "Resource": ["*"] }] }</pre>
GetResourceLogLevel	Returns the log-level override for a given resource identifier and resource	<pre>{ }</pre>

API name	Description	Sample IAM policy
	<p>type. The resource can be a wireless device or a wireless gateway.</p>	<pre> "Version": "2012-10-17", "Statement": [{ "Effect": "Allow", "Action": ["iotwireless:GetResourceLogLevel"], "Resource": ["arn:aws:iotwireless:us-east-1:123456789012:WirelessDevice/012bc5ab12-cd3a-d00e-1f0e20c1204a",] }] } </pre>
PutResourceLogLevel	<p>Sets the log-level override for a given resource identifier and resource type. The resource can be a wireless gateway or a wireless device.</p> <p>Note This API has a limit of 200 log-level overrides per account.</p>	<pre> { "Version": "2012-10-17", "Statement": [{ "Effect": "Allow", "Action": ["iotwireless:PutResourceLogLevel"], "Resource": ["arn:aws:iotwireless:us-east-1:123456789012:WirelessDevice/012bc5ab12-cd3a-d00e-1f0e20c1204a",] }] } </pre>

API name	Description	Sample IAM policy
ResetAllResourceLogLevels	<p>Removes the log-level overrides for all resources, which includes both wireless gateways and wireless devices.</p> <p>Note This API doesn't affect the log levels that are set using the UpdateLogLevelByResourceType API.</p>	<pre>{ "Version": "2012-10-17", "Statement": [{ "Effect": "Allow", "Action": ["iotwireless:ResetAllResourceLogLevels"], "Resource": ["arn:aws:iotwireless:us-east-1:123456789012:WirelessDevice/*", "arn:aws:iotwireless:us-east-1:123456789012:WirelessGateway/*"] }] }</pre>
ResetResourceLogLevel	<p>Removes the log-level override for a given resource identifier and resource type. The resource can be a wireless gateway or a wireless device.</p>	<pre>{ "Version": "2012-10-17", "Statement": [{ "Effect": "Allow", "Action": ["iotwireless:ResetResourceLogLevel"], "Resource": ["arn:aws:iotwireless:us-east-1:123456789012:WirelessDevice/012bc5ab12-cd3a-d00e-1f0e20c1204a", ...] }] }</pre>

API name	Description	Sample IAM policy
UpdateLogLevelByResourceType	<p>Sets default log level, or log levels by resource types. You can use this API for log options for wireless devices or wireless gateways, and control the log messages that'll be displayed in CloudWatch.</p> <p>Note Events are optional and the event type is tied to the resource type. For more information, see Events and resource types (p. 1090).</p>	<pre>{ "Version": "2012-10-17", "Statement": [{ "Effect": "Allow", "Action": ["iotwireless:UpdateLogLevelByResourceType"], "Resource": ["*"] }] }</pre>

Configure log levels of resources using the CLI

This section describes how to configure log levels for AWS IoT Core for LoRaWAN resources by using the API or AWS CLI.

Before you use the CLI:

- Make sure you created the IAM policy for the API for which you want to run the CLI command, as described previously.
- You need the Amazon Resource Name (ARN) of the role you want to use. If you need to create a role to use for logging, see [Create logging role and policy for AWS IoT Core for LoRaWAN \(p. 1079\)](#).

Why use the AWS CLI

By default, if you create the IAM role, `IoTWirelessLogsRole`, as described in [Create logging role and policy for AWS IoT Core for LoRaWAN \(p. 1079\)](#), you'll see CloudWatch logs in the AWS Management Console that have a default log level of `ERROR`. To change the default log level for all your resources or for specific resources, use the AWS IoT Wireless logging API or CLI.

How to use the AWS CLI

The API actions can be categorized into the following types depending on whether you want to configure log levels for all resources or for specific resources:

- API actions `GetLogLevelByResourceTypes` and `UpdateLogLevelByResourceTypes` can retrieve and update the log levels for all resources in your account that are of a specific type, such as a wireless gateway, or a LoRaWAN or Sidewalk device.
- API actions `GetResourceLogLevel`, `PutResourceLogLevel`, and `ResetResourceLogLevel` can retrieve, update, and reset log levels of individual resources that you specify using a resource identifier.
- API action `ResetAllResourceLogLevels` resets the log-level override to `null` for all resources for which you specified a log-level override using the `PutResourceLogLevel` API.

To use the CLI to configure resource-specific logging for AWS IoT

Note

You can also perform this procedure with the API by using the methods in the AWS API that correspond to the CLI commands shown here.

1. By default, all resources have log level set to `ERROR`. To set the default log levels, or log levels by resource types for all resources in your account, use the [update-log-levels-by-resource-types](#) command. The following example shows how you can create a JSON file, `Input.json`, and provide it as an input to the CLI command. You can use this command to selectively disable logging or override the default log level for specific types of resources and events.

```
{  
    "DefaultLogLevel": "INFO",  
    "WirelessDeviceLogOptions":  
    [  
        {  
            "Type": "Sidewalk",  
            "LogLevel": "INFO",  
            "Events":  
            [  
                {  
                    "Event": "Registration",  
                    "LogLevel": "DISABLED"  
                }  
            ]  
        },  
        {  
            "Type": "LoRaWAN",  
            "LogLevel": "INFO",  
            "Events":  
            [  
                {  
                    "Event": "Join",  
                    "LogLevel": "DISABLED"  
                },  
                {  
                    "Event": "Rejoin",  
                    "LogLevel": "ERROR"  
                }  
            ]  
        }  
    ]  
    "WirelessGatewayLogOptions":  
    [  
        {  
            "Type": "LoRaWAN",  
            "LogLevel": "INFO",  
            "Events":  
            [  
                {  
                    "Event": "CUPS_Request",  
                    "LogLevel": "DISABLED"  
                },  
                {  
                    "Event": "Certificate",  
                    "LogLevel": "ERROR"  
                }  
            ]  
        }  
    ]  
}
```

where:

WirelessDeviceLogOptions

The list of log options for a wireless device. Each log option includes the wireless device type (Sidewalk or LoRaWAN), and a list of wireless device event log options. Each wireless device event log option can optionally include the event type and its log level.

WirelessGatewayLogOptions

The list of log options for a wireless gateway. Each log option includes the wireless gateway type (LoRaWAN), and a list of wireless gateway event log options. Each wireless gateway event log option can optionally include the event type and its log level.

DefaultLogLevel

The log level to use for all your resources. Valid values are: `ERROR`, `INFO`, and `DISABLED`. The default value is `INFO`.

LogLevel

The log level you want to use for individual resource types and events. These log levels override the default log level, such as the log level `INFO` for the LoRaWAN gateway, and log levels `DISABLED` and `ERROR` for the two event types.

Run the following command to provide the `Input.json` file as input to the command. This command doesn't produce any output.

```
aws iotwireless update-log-levels-by-resource-types \
--cli-input-json Input.json
```

If you want to remove the log options for both wireless devices and wireless gateways, run the following command.

```
{
  "DefaultLogLevel": "DISABLED",
  "WirelessDeviceLogOptions": [],
  "WirelessGatewayLogOptions": []
}
```

2. The `update-log-levels-by-resource-types` command doesn't return any output. Use the [get-log-levels-by-resource-types](#) command to retrieve resource-specific logging information. The command returns the default log level, and the wireless device and wireless gateway log options.

Note

The `get-log-levels-by-resource-types` command can't directly retrieve the log levels in the CloudWatch console. You can use the `get-log-levels-by-resource-types` command to get the latest log-level information that you've specified for your resources using the `update-log-levels-by-resource-types` command.

```
aws iotwireless get-log-levels-by-resource-types
```

When you run the following command, it returns the latest logging information you specified with `update-log-levels-by-resource-types`. For example, if you remove the wireless device log options, then running the `get-log-levels-by-resource-types` will return this value as `null`.

```
{
  "DefaultLogLevel": "INFO",
```

```
"WirelessDeviceLogOptions": null,  
"WirelessGatewayLogOptions":  
[  
  {  
    "Type": "LoRaWAN",  
    "LogLevel": "INFO",  
    "Events":  
      [  
        {  
          "Event": "CUPS_Request",  
          "LogLevel": "DISABLED"  
        },  
        {  
          "Event": "Certificate",  
          "LogLevel": "ERROR"  
        }  
      ]  
  }  
]
```

3. To control log levels for individual wireless gateways or wireless device resources, use the following CLI commands:

- [put-resource-log-level](#)
- [get-resource-log-level](#)
- [reset-resource-log-level](#)

For an example for when to use these CLIs, say that you have a large number of wireless devices or gateways in your account that are being logged. If you want to troubleshoot errors for only some of your wireless devices, you can disable logging for all wireless devices by setting the DefaultLogLevel to DISABLED, and use the **put-resource-log-level** to set the LogLevel to ERROR for only those devices in your account.

```
aws iotwireless put-resource-log-level \  
  --resource-identifier  
  --resource-type WirelessDevice  
  --log-level ERROR
```

In this example, the command sets the log level to ERROR only for the specified wireless device resource and the logs for all other resources are disabled. This command doesn't produce any output. To retrieve this information and verify that the log levels were set, use the **get-resource-log-level** command.

4. In the previous step, after you've debugged the issue and resolved the error, you can run the **reset-resource-log-level** command to reset the log level for that resource to null. If you used the **put-resource-log-level** command to set the log-level override for more than one wireless device or gateway resource, such as for troubleshooting errors for multiple devices, you can reset the log-level overrides back to null for all those resources using the **reset-all-resource-log-levels** command.

```
aws iotwireless reset-all-resource-log-levels
```

This command doesn't produce any output. To retrieve the logging information for the resources, run the **get-resource-log-level** command.

Next Steps

You've learned how to create the logging role and use the AWS IoT Wireless API to configure logging for your AWS IoT Core for LoRaWAN resources. Next, to learn about monitoring your log entries, go to [Monitor AWS IoT Core for LoRaWAN using CloudWatch Logs \(p. 1089\)](#).

Monitor AWS IoT Core for LoRaWAN using CloudWatch Logs

AWS IoT Core for LoRaWAN has more than 50 CloudWatch log entries that are enabled by default. Each log entry describes the event type, log level, and the resource type. For more information, see [AWS IoT Core for LoRaWAN resources and log levels \(p. 1081\)](#).

How to monitor your AWS IoT Core for LoRaWAN resources

When logging is enabled for AWS IoT Core for LoRaWAN, AWS IoT Core for LoRaWAN sends progress events about each message as it passes from your devices through AWS IoT and back. By default, AWS IoT Core for LoRaWAN log entries have a default log level of error. When you enable logging as described in [Create logging role and policy for AWS IoT Core for LoRaWAN \(p. 1079\)](#), you'll see messages in the CloudWatch console that have a default log level of `ERROR`. By using this log level, the messages will show only error information for all wireless devices and gateway resources that you're using.

If you want the logs to show additional information, such as those that have a log level of `INFO`, or disable logs for some of your devices and show log messages for only some of your devices, you can use the AWS IoT Core for LoRaWAN logging API. For more information, see [Configure log levels of resources using the CLI \(p. 1085\)](#).

You can also create filter expressions to display only the required messages.

Before you can view AWS IoT Core for LoRaWAN logs in the console

To make the `/aws/iotwireless` log group appear in the CloudWatch console, you must have done the following.

- Enabled logging in AWS IoT Core for LoRaWAN. For more information about how to enable logging in AWS IoT Core for LoRaWAN, see [Configure Logging for AWS IoT Core for LoRaWAN \(p. 1079\)](#).
- Written some log entries by performing AWS IoT Core for LoRaWAN operations.

To create and use filter expressions more effectively, we recommend that you try using CloudWatch Insights as described in the following topics. We also recommend that you follow the topics in the order they're presented here. This will help you use CloudWatch **Log groups** first to learn about the different types of resources, its event types, and log levels that you can use to view log entries in the console. You can then learn how to create filter expressions by using CloudWatch Insights to get more helpful information from your resources.

Topics

- [View CloudWatch AWS IoT Core for LoRaWAN log entries \(p. 1089\)](#)
- [Use CloudWatch Insights to filter logs for AWS IoT Core for LoRaWAN \(p. 1095\)](#)

View CloudWatch AWS IoT Core for LoRaWAN log entries

After you've configured logging for AWS IoT Core for LoRaWAN as described in [Create logging role and policy for AWS IoT Core for LoRaWAN \(p. 1079\)](#) and written some log entries, you can view the log entries in the CloudWatch console by performing the following steps.

Viewing AWS IoT logs in the CloudWatch Log groups console

In the [CloudWatch console](#), CloudWatch logs appear in a log group named `/aws/iotwireless`. For more information about CloudWatch Logs, see [CloudWatch Logs](#).

To view your AWS IoT logs in the CloudWatch console

Navigate to the [CloudWatch console](#) and choose **Log groups** in the navigation pane.

1. In the **Filter** text box, enter `/aws/iotwireless`, and then choose the `/aws/iotwireless` Log group.
2. To see a complete list of the AWS IoT Core for LoRaWAN logs generated for your account, choose **Search all**. To look at an individual log stream, choose the expand icon.
3. To filter the log streams, you can also enter a query in the **Filter events** text box. Here are some queries to try:

- `{ $.logLevel = "ERROR" }`

Use this filter to find all logs that have a log level of `ERROR` and you can expand the individual error streams to read the error messages, which will help you resolve them.

- `{ $.resource = "WirelessGateway" }`

Find all logs for the `WirelessGateway` resource regardless of the log level.

- `{ $.event = "CUPS_Request" && $.logLevel = "ERROR" }`

Find all logs that have an event type of `CUPS_Request` and a log level of `ERROR`.

Events and resource types

The following table shows the different types of events for which you'll see log entries. The event types also depend on whether the resource type is a wireless device or a wireless gateway. You can use the default log level for the resources and event types or override the default log level by specifying a log level for each of them.

Event types based on resources used

Resource	Resource type	Event type	
Wireless gateway	LoRaWAN	<ul style="list-style-type: none">• CUPS_Request• Certificate	
Wireless device	LoRaWAN	<ul style="list-style-type: none">• Join• Rejoin• Uplink_Data• Downlink_Data	
Wireless device	Sidewalk	<ul style="list-style-type: none">• Registration• Uplink_Data• Downlink_Data	

The following topic contains more information about these event types and the log entries for wireless gateways and wireless devices.

Topics

- [Log entries for wireless gateways and wireless device resources \(p. 1091\)](#)

Log entries for wireless gateways and wireless device resources

After you've enabled logging, you can view log entries for your wireless gateways and wireless devices. The following section describes the various kinds of log entries based on your resource and event types.

Wireless gateway log entries

This section shows some of the sample log entries for your wireless gateway resources that you'll see in the [CloudWatch console](#). These log messages can have event type as `CUPS_Request` or `Certificate`, and can be configured to display a log level of `INFO`, `ERROR`, or `DISABLED` at the resource level or the event level. If you want to see only error information, set the log level to `ERROR`. The message in the `ERROR` log entry will contain information about why it failed.

The log entries for your wireless gateway resource can be classified based on the following event types:

- **CUPS_Request**

The LoRa Basics Station running on your gateway periodically sends a request to the Configuration and Update Server (CUPS) for updates. For this event type, if you set log level to `INFO` when configuring the CLI for your wireless gateway resource, then in the logs:

- If the event is successful, you'll see log messages that have a `logLevel` of `INFO`. The messages will include details about the CUPS response sent to your gateway and the gateway details. Following shows an example of this log entry. For more information about the `logLevel` and other fields in the log entry, see [AWS IoT Core for LoRaWAN resources and log levels \(p. 1081\)](#).

```
{  
  "timestamp": "2021-05-13T16:56:08.853Z",  
  "resource": "WirelessGateway",  
  "wirelessGatewayId": "5da85cc8-3361-4c79-8be3-3360fb87abda",  
  "wirelessGatewayType": "LoRaWAN",  
  "gatewayEui": "feffff0000000e2",  
  "event": "CUPS_Request",  
  "logLevel": "INFO",  
  "message": "Sending CUPS response of total length 3213 to GatewayEui:  
feffff0000000e2 with TC Credentials,"  
}
```

- If there is an error, you'll see log entries that have a `logLevel` of `ERROR`, and the messages will include details about the error. Examples of when an error can occur for the `CUPS_Request` event include: missing CUPS CRC, mismatch in the gateway's TC Uri with AWS IoT Core for LoRaWAN, missing `IoTWirelessGatewayCertManagerRole`, or not able to obtain wireless gateway record. Following example shows a missing CRC log entry. To resolve the error, check your gateway setup to verify that you've entered the correct CUPS CRC.

```
{  
  "timestamp": "2021-05-13T16:56:08.853Z",  
  "resource": "WirelessGateway",  
  "wirelessGatewayId": "5da85cc8-3361-4c79-8be3-3360fb87abda",  
  "wirelessGatewayType": "LoRaWAN",  
  "gatewayEui": "feffff0000000e2",  
  "event": "CUPS_Request",  
  "logLevel": "ERROR",  
  "message": "The CUPS CRC is missing from the request. Check your gateway setup and  
enter the CUPS CRC,"  
}
```

- **Certificate**

These log entries will help you check whether your wireless gateway presented the correct certificate for authenticating connection to AWS IoT. For this event type, if you set log level to `INFO` when configuring the CLI for your wireless gateway resource, then in the logs:

- If the event is successful, you'll see log messages that have a `logLevel` of `INFO`. The messages will include details about the Certificate ID and the Wireless gateway identifier. Following shows an example of this log entry. For more information about the `logLevel` and other fields in the log entry, see [AWS IoT Core for LoRaWAN resources and log levels \(p. 1081\)](#).

```
{  
    "resource": "WirelessGateway",  
    "wirelessGatewayId": "5da85cc8-3361-4c79-8be3-3360fb87abda",  
    "wirelessGatewayType": "LoRaWAN",  
    "event": "Certificate",  
    "logLevel": "INFO",  
    "message": "Gateway connection authenticated.  
    (CertificateId: b5942a7aee973eda24314e416889227a5e0aa5ed87e6eb89239a83f515dea17c,  
    WirelessGatewayId: 5da85cc8-3361-4c79-8be3-3360fb87abda)"  
}
```

- If there is an error, you'll see log entries that have a `logLevel` of `ERROR`, and the messages will include details about the error. Examples of when an error can occur for the Certificate event include an invalid Certificate ID, wireless gateway identifier, or a mismatch between the wireless gateway identifier and the Certificate ID. Following example shows an `ERROR` due to invalid wireless gateway identifier. To resolve the error, check the gateway identifiers.

```
{  
    "resource": "WirelessGateway",  
    "wirelessGatewayId": "5da85cc8-3361-4c79-8be3-3360fb87abda",  
    "wirelessGatewayType": "LoRaWAN",  
    "event": "Certificate",  
    "logLevel": "INFO",  
    "message": "The gateway connection couldn't be authenticated because a provisioned  
    gateway associated with the certificate couldn't be found.  
    (CertificateId: 729828e264810f6fc7134daf68056e8fd848afc32bfe8082beeb44116d709d9e)"  
}
```

Wireless device log entries

This section shows some of the sample log entries for your wireless device resources that you'll see in the [CloudWatch console](#). The event type for these log messages depend on whether you're using a LoRaWAN or a Sidewalk device. Each wireless device resource or event type can be configured to display a log level of `INFO`, `ERROR`, or `DISABLED`.

Note

Your request must not contain both LoRaWAN and Sidewalk wireless metadata at the same time. To avoid an `ERROR` log entry for this scenario, specify either LoRaWAN or Sidewalk wireless data.

LoRaWAN device log entries

The log entries for your LoRaWAN wireless device can be classified based on the following event types:

- **Join and Rejoin**

When you add a LoRaWAN device and connect it to AWS IoT Core for LoRaWAN, before your device can send uplink data, you must complete a process called activation or join procedure. For more information, see [Add your wireless device to AWS IoT Core for LoRaWAN \(p. 1011\)](#).

For this event type, if you set log level to `INFO` when configuring the CLI for your wireless gateway resource, then in the logs:

- If the event is successful, you'll see log messages that have a `logLevel` of `INFO`. The messages will include details about the status of your join or rejoin request. Following shows an example of this

log entry. For more information about the `logLevel` and other fields in the log entry, see [AWS IoT Core for LoRaWAN resources and log levels \(p. 1081\)](#).

```
{  
    "timestamp": "2021-05-13T16:56:08.853Z",  
    "resource": "WirelessDevice",  
    "wirelessDeviceType": "LoRaWAN",  
    "wirelessDeviceId": "5da85cc8-3361-4c79-8be3-3360fb87abda",  
    "devEui": "feffff00000000e2",  
    "event": "Rejoin",  
    "logLevel": "INFO",  
    "message": "Rejoin succeeded"  
}
```

- If there is an error, you'll see log entries that have a `logLevel` of `ERROR`, and the messages will include details about the error. Examples of when an error can occur for the `Join` and `Rejoin` events include invalid LoRaWAN region setting, or invalid Message Integrity Code (MIC) check. Following example shows a join error due to MIC check. To resolve the error, check whether you've entered the correct root keys.

```
{  
    "timestamp": "2020-11-24T01:46:50.883481989Z",  
    "resource": "WirelessDevice",  
    "wirelessDeviceType": "LoRaWAN",  
    "wirelessDeviceId": "cb4c087c-1be5-4990-8654-ccf543ee9fff",  
    "devEui": "58a0cb000020255c",  
    "event": "Join",  
    "logLevel": "ERROR",  
    "message": "invalid MIC. It's most likely caused by wrong root keys."  
}
```

- **Uplink_Data and Downlink_Data**

The event type `Uplink_Data` is used for messages that are generated by AWS IoT Core for LoRaWAN when the payload is sent from the wireless device to AWS IoT. The event type `Downlink_Data` is used for messages that are related to downlink messages that are sent from AWS IoT to the wireless device.

Note

Events `Uplink_Data` and `Downlink_Data` apply to both LoRaWAN and Sidewalk devices.

For this event type, if you set log level to `INFO` when configuring the CLI for your wireless devices, then in the logs, you'll see:

- If the event is successful, you'll see log messages that have a `logLevel` of `INFO`. The messages will include details about the status of the uplink or downlink message that was sent and the wireless device identifier. Following shows an example of this log entry for a Sidewalk device. For more information about the `logLevel` and other fields in the log entry, see [AWS IoT Core for LoRaWAN resources and log levels \(p. 1081\)](#).

```
{  
    "resource": "WirelessDevice",  
    "wirelessDeviceId": "5371db88-d63d-481a-868a-e54b6431845d",  
    "wirelessDeviceType": "Sidewalk",  
    "event": "Downlink_Data",  
    "logLevel": "INFO",  
    "messageId": "8da04fa8-037d-4ae9-bf67-35c4bb33da71",  
    "message": "Message delivery succeeded. MessageId: 8da04fa8-037d-4ae9-  
bf67-35c4bb33da71. AWS IoT Core: {\\"message\\":\\"OK\\",\\"traceId\\":\\"038b5b05-a340-  
d18a-150d-d5a578233b09\\"}"  
}
```

- If there is an error, you'll see log entries that have a `logLevel` of `ERROR`, and the messages will include details about the error, which will help you resolve it. Examples of when an error can occur for the Registration event include: authentication issues, invalid or too many requests, unable to encrypt or decrypt the payload, or unable to find the wireless device using the specified ID. Following example shows a permission error encountered while processing a message.

```
{  
    "resource": "WirelessDevice",  
    "wirelessDeviceId": "cb4c087c-1be5-4990-8654-ccf543ee9fff",  
    "wirelessDeviceType": "LoRaWAN",  
    "event": "Uplink_Data",  
    "logLevel": "ERROR",  
    "message": "Cannot assume role MessageId: ef38877f-3454-4c99-96ed-5088c1cd8dee.  
Access denied: User: arn:aws:sts::005196538709:assumed-role/  
DataRoutingServiceRole/6368b35fd48c445c9a14781b5d5890ed is not authorized to perform:  
sts:AssumeRole on resource: arn:aws:iam::400232685877:role/ExecuteRules_Role\tstatus  
code: 403, request id: 471c3e35-f8f3-4e94-b734-c862f63f4edb"  
}
```

Sidewalk device log entries

The log entries for your Sidewalk device can be classified based on the following event types:

- **Registration**

These log entries will help you monitor the status of any Sidewalk devices that you're registering with AWS IoT Core for LoRaWAN. For this event type, if you set log level to `INFO` when configuring the CLI for your wireless device resource, then in the logs, you'll see log messages that have a `logLevel` of `INFO` and `ERROR`. The messages will include details about the registration progress from start to completion. `ERROR` log messages will contain information about how to troubleshoot issues with registering your device.

Following shows an example for a log message with log level of `INFO`. For more information about the `logLevel` and other fields in the log entry, see [AWS IoT Core for LoRaWAN resources and log levels \(p. 1081\)](#).

```
{  
    "resource": "WirelessDevice",  
    "wirelessDeviceId": "8d0b2775-e19b-4b2a-a351-cb8a2734a504",  
    "wirelessDeviceType": "Sidewalk",  
    "event": "Registration",  
    "logLevel": "INFO",  
    "message": "Successfully completed device registration. Amazon SidewalkId =  
2000000002"  
}
```

- **Uplink_Data and Downlink_Data**

The event types `Uplink_Data` and `Downlink_Data` for Sidewalk devices are similar to the corresponding event types for LoRaWAN devices. For more information, refer to the **Uplink_Data and Downlink_Data** section described previously for LoRaWAN device log entries.

Next steps

You've learned how to view log entries for your resources and the different log entries that you can view in the CloudWatch console after enabling logging for AWS IoT Core for LoRaWAN. While you can create filter streams using **Log groups**, we recommend that you use CloudWatch Insights to create and use

filter streams. For more information, see [Use CloudWatch Insights to filter logs for AWS IoT Core for LoRaWAN \(p. 1095\)](#).

Use CloudWatch Insights to filter logs for AWS IoT Core for LoRaWAN

While you can use CloudWatch Logs to create filter expressions, we recommend that you use CloudWatch insights to more effectively create and use filter expressions depending on your application.

We recommend that you first use CloudWatch **Log groups** to learn about the different types of resources, its event types, and log levels that you can use to view log entries in the console. You can then use the examples of some filter expressions on this page as a reference to create your own filters for your AWS IoT Core for LoRaWAN resources.

Viewing AWS IoT logs in the CloudWatch Logs insights console

In the [CloudWatch console](#), CloudWatch logs appear in a log group named `/aws/iotwireless`. For more information about CloudWatch Logs, see [CloudWatch Logs](#).

To view your AWS IoT logs in the CloudWatch console

Navigate to the [CloudWatch console](#) and choose **Logs Insights** in the navigation pane.

1. In the **Filter** text box, enter `/aws/iotwireless`, and then choose the `/aws/iotwireless` Logs Insights.
2. To see a complete list of log groups, choose **Select log group(s)**. To look at log groups for AWS IoT Core for LoRaWAN, choose `/aws/iotwireless`.

You can now start entering queries to filter the log groups. The following sections contain some useful queries that'll help you gain insights about your resource metrics.

Create useful queries to filter and gain insights for AWS IoT Core for LoRaWAN

You can use filter expressions to show additional helpful log information with CloudWatch Insights. Following shows some sample queries:

Show only logs for specific resource types

You can create a query that'll help you show logs for only specific resource types, such as a LoRaWAN gateway or a Sidewalk device. For example, to filter logs to show only messages for Sidewalk devices, you can enter the following query and choose **Run query**. To save this query, choose **Save**.

```
fields @message
| filter @message like /Sidewalk/
```

After the query runs, you'll see the results in the **Logs** tab, which shows the timestamps for logs related to Sidewalk devices in your account. You'll also see a bar graph, which will show the time at which the events occurred, if there were such events that occurred previously related to your Sidewalk device. Following shows an example if you expand one of the results in the **Logs** tab. Alternatively, if you want to troubleshoot errors related to Sidewalk devices, you can add another filter that sets the log level to **ERROR** and show only error information.

Field	Value
@ingestionTime	1623894967640
@log	954314929104:/aws/iotwireless
@logStream	WirelessDevice-
Downlink_Data-715adccfb34170214ec2f6667ddfa13cb5af2c3ddfc52fbe0e554a2e780bed	

```

@message      {
    "resource": "WirelessDevice",
    "wirelessDeviceId": "3b058d05-4e84-4e1a-b026-4932bddf978d",
    "wirelessDeviceType": "Sidewalk",
    "devEui": "feffff000000011a",
    "event": "Downlink_Data",
    "logLevel": "INFO",
    "messageId": "7e752a10-28f5-45a5-923f-6fa7133fedda",
    "message": "Successfully sent downlink message. Amazon SidewalkId = 2000000006, Sequence number = 0"
}
@timestamp      1623894967640
devEui          feffff000000011a
event           Downlink_Data
logLevel        INFO
message         Successfully sent downlink message. Amazon SidewalkId = 2000000006,
Sequence number = 0
messageId       7e752a10-28f5-45a5-923f-6fa7133fedda
resource        WirelessDevice
wirelessDeviceId 3b058d05-4e84-4e1a-b026-4932bddf978d
wirelessDeviceType Sidewalk

```

Show specific messages or events

You can create a query that'll help you show specific messages and observe when the events occurred. For example, if you want to see when your downlink message was sent from your LoRaWAN wireless device, you can enter the following query and choose **Run query**. To save this query, choose **Save**.

```
filter @message like /Downlink message sent/
```

After the query runs, you'll see the results in the **Logs** tab, which shows the timestamps when the downlink message was successfully sent to your wireless device. You'll also see a bar graph, which will show the time at which a downlink message was sent, if there were downlink messages were previously sent to your wireless device. Following shows an example if you expand one of the results in the **Logs** tab. Alternatively, if a downlink message wasn't sent, you can modify the query to display only results for when the message wasn't sent so that you can debug the issue.

Field	Value
@ingestionTime	1623884043676
@log	954314929104:/aws/iotwireless
@logStream	WirelessDevice-
Downlink_Data-42d0e6d09ba4d7015f4e9756fc616d401cd85fe3ac19854d9fdb866153c872	
@message	<pre> { "timestamp": "2021-06-16T22:54:00.770493863Z", "resource": "WirelessDevice", "wirelessDeviceId": "3b058d05-4e84-4e1a-b026-4932bddf978d", "wirelessDeviceType": "LoRaWAN", "devEui": "feffff000000011a", "event": "Downlink_Data", "logLevel": "INFO", "messageId": "7e752a10-28f5-45a5-923f-6fa7133fedda", "message": "Downlink message sent. MessageId: 7e752a10-28f5-45a5-923f-6fa7133fedda" } </pre>
@timestamp	1623884040858
devEui	feffff000000011a
event	Downlink_Data
logLevel	INFO
message	Downlink message sent. MessageId: 7e752a10-28f5-45a5-923f-6fa7133fedda
messageId	7e752a10-28f5-45a5-923f-6fa7133fedda
resource	WirelessDevice
timestamp	2021-06-16T22:54:00.770493863Z

```
wirelessDeviceId      3b058d05-4e84-4e1a-b026-4932bddf978d
wirelessDeviceType    LoRaWAN
```

Next steps

You've learned how to use CloudWatch Insights to gain more helpful information by creating queries to filter log messages. You can combine some of the filters described previously and design your own filters depending on the resource you're monitoring. For more information about using CloudWatch Insights, see [Analyzing log data with CloudWatch Insights](#).

After you've created queries with CloudWatch Insights, if you've saved them, you can load and run the saved queries as needed. Alternatively, if you click the **History** button in the CloudWatch **Logs Insights** console, you can view the previously run queries and rerun them as needed, or further modify them by creating additional queries.

Data security with AWS IoT Core for LoRaWAN

Two methods secure the data from your AWS IoT Core for LoRaWAN devices:

- **The security that wireless devices use to communicate with the gateways.**

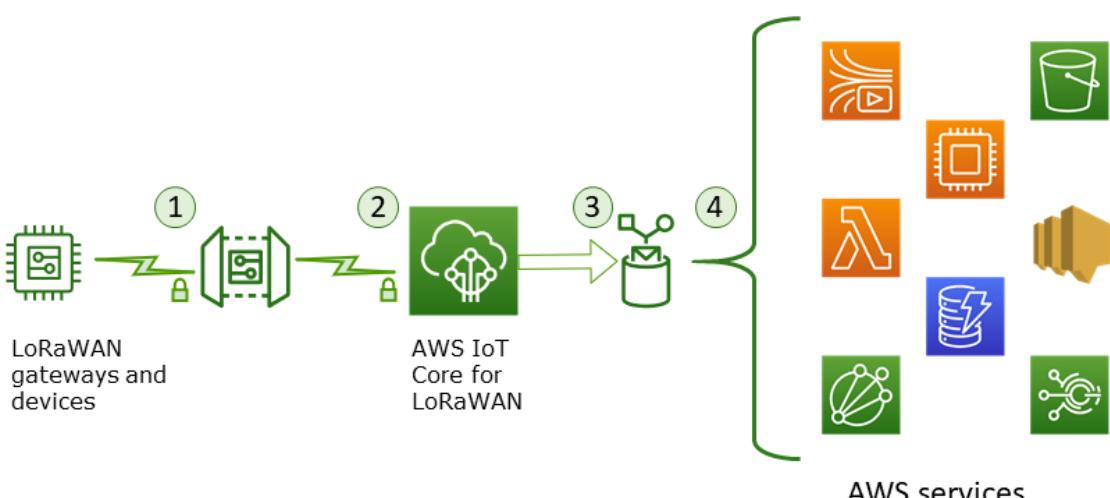
The LoRaWAN devices follow the security practices described in [LoRaWAN™ SECURITY: A White Paper Prepared for the LoRa Alliance™ by Gemalto, Actility, and Semtech](#) to communicate with the gateways.

- **The security that AWS IoT Core uses to connect gateways to AWS IoT Core for LoRaWAN and send the data to other AWS services.**

AWS IoT Core security is described in [Data protection in AWS IoT Core \(p. 361\)](#).

How data is secured throughout the system

This diagram identifies the key elements in a LoRaWAN system connected to AWS IoT Core for LoRaWAN to identify how data is secured throughout.



1. The LoRaWAN wireless device encrypts its binary messages using AES128 CTR mode before it transmits them.

2. Gateway connections to AWS IoT Core for LoRaWAN are secured by TLS as described in [Transport security in AWS IoT \(p. 362\)](#). AWS IoT Core for LoRaWAN decrypts the binary message and encodes the decrypted binary message payload as a base64 string.
3. The resulting base64-encoded message is sent as the message payload to the AWS IoT rule described in the destination assigned to the device. Data within AWS is encrypted using AWS-owned keys.
4. The AWS IoT rule directs the message data to the services described in the rule's configuration. Data within AWS is encrypted using AWS-owned keys.

LoRaWAN device and gateway transport security

LoRaWAN devices and AWS IoT Core for LoRaWAN store pre-shared root keys. Session keys are derived by both LoRaWAN devices and AWS IoT Core for LoRaWAN following the protocols. The symmetric session keys are used for encryption and decryption in a standard AES-128 CTR mode. A 4-byte message integrity code (MIC) is also used to check the data integrity following a standard AES-128 CMAC algorithm. The session keys can be updated by using the Join/Rejoin process.

The security practice for LoRa gateways is described in the LoRaWAN specifications. LoRa gateways connect to AWS IoT Core for LoRaWAN through a web socket using a [Basics Station](#). AWS IoT Core for LoRaWAN supports only [Basics Station](#) version 2.0.4 and later.

Before the web socket connection is established, AWS IoT Core for LoRaWAN uses the [TLS Server and Client Authentication mode \(p. 362\)](#) to authenticate the gateway. AWS IoT Core for LoRaWAN also maintains a Configuration and Update Server (CUPS) that configures and updates the certificates and keys used for TLS authentication.

Amazon Sidewalk Integration for AWS IoT Core

[Amazon Sidewalk](#) is a shared network that helps devices like Amazon Echo, Ring security cams, outdoor lights, motion sensors, and tile trackers work better at home and beyond the front door. When enabled, this network can support other Sidewalk devices in your community, and open the door to innovations such as locating items connected to Amazon Sidewalk. Amazon Sidewalk helps your devices get connected and stay connected. For example, if your device loses its Wi-Fi connection, Sidewalk can simplify reconnecting to your router. For more information, see [Amazon Sidewalk Quick Start Guide](#).

The following sections show how to onboard your Sidewalk devices with AWS IoT and use event notifications to notify you of events such as when your Sidewalk device is registered. For information about using Amazon CloudWatch to monitor your Sidewalk devices, see [Monitoring and logging for AWS IoT Core for LoRaWAN using Amazon CloudWatch](#) (p. 1078).

How to onboard your Sidewalk device

You can onboard your Sidewalk devices to AWS IoT by using the console or the AWS IoT Wireless API. After your Amazon Sidewalk devices are authenticated, their messages are sent to AWS IoT Core. You can then start developing your business applications on the AWS Cloud, which uses the data from your Amazon Sidewalk devices.

Using the console

To onboard your Sidewalk devices by using the AWS Management Console, first register your device in the [Sidewalk Developer Service \(SDS\) console](#) account and then associate your Amazon ID with your AWS account. To see the Sidewalk devices you've added and manage them, sign in to the AWS Management Console and navigate to the [Devices](#) page on the AWS IoT console.

Using the API or CLI

You can onboard both Sidewalk and LoRaWAN devices by using the [AWS IoT Wireless](#) API. The AWS IoT Wireless API that AWS IoT Core is built on is supported by the AWS SDK. For more information, see [AWS SDKs and Toolkits](#).

You can use the AWS CLI to run commands for onboarding and managing your Sidewalk devices. For more information, see [AWS IoT Wireless CLI reference](#).

Getting started with Amazon Sidewalk Integration for AWS IoT Core

With Amazon Sidewalk Integration for AWS IoT Core, you can add your Sidewalk device fleet to the AWS Cloud. Use the following steps to get started.

1. Review the Sidewalk SDK and documentation

Learn more about Amazon Sidewalk and how your devices can use it.

- a. Review the Amazon Sidewalk Quick Start Guide.
 - b. Download an SDK for Amazon Sidewalk.
 - c. Open the [Sidewalk Developer Service \(SDS\) console](#).
2. **Register your prototype device**
In the SDS console, register your prototype device with Amazon Sidewalk.
 3. **Associate your Sidewalk Amazon ID with your AWS account**
In the [AWS IoT console](#), associate your Sidewalk Amazon ID with your AWS account.
Your Amazon Sidewalk devices appear in the **Sidewalk** tab of the **Devices** hub of the [AWS IoT console](#).
 4. **Complete the Amazon Sidewalk device configuration in the AWS IoT console**
Create the destinations and rules your Sidewalk device needs to route and format the data for AWS services.

The following topics show how you can add Sidewalk devices and connect them to AWS IoT. Before adding your devices, make sure that your AWS account has the required IAM permissions to perform the following procedures.

Topics

- [Add your Sidewalk account credentials \(p. 1100\)](#)
- [Add a destination for your Sidewalk device \(p. 1101\)](#)
- [Create rules to process Sidewalk device messages \(p. 1103\)](#)
- [Connect your Sidewalk device and view uplink metadata format \(p. 1104\)](#)

Add your Sidewalk account credentials

You can connect Sidewalk devices to AWS IoT by using the AWS Management Console or the AWS IoT Wireless API. To onboard your device, we'll create a wireless connectivity profile for your Sidewalk device and then add a destination and AWS IoT rule for the profile and Sidewalk endpoints.

Before adding your device, you must add your Sidewalk account credentials. You can add your credentials by using the AWS Management Console or the AWS IoT Wireless API.

Add your Sidewalk account credentials by using the console

To add your Sidewalk account credentials from the console:

1. Navigate to the [Profiles](#) page of the AWS IoT console and choose the **Sidewalk** tab.

Note

Make sure that you're using the `us-east-1` Region. This tab doesn't appear in the console if you're using a different Region.

2. Enter the Sidewalk Amazon ID. You get this ID from the [Sidewalk Developer Service \(SDS\) console](#) when designing your Sidewalk product. For more information, see [Design your Sidewalk product](#).
3. Upload the **AppServerPrivateKey**, which is the server key that your vendor provided. The **AppServerPrivateKey** is the ED25519 private key (`app-server-ed25519-private.txt` file), which is a 64-digit hexadecimal value. You generated this key by using the Sidewalk certificate generation tool when you designed your Sidewalk product. For more information, see [Design your Sidewalk product](#).
4. To add your Sidewalk credentials, choose **Add credential**.

Add your Sidewalk account credentials by using the API

You can use the AWS IoT Wireless API to add your Sidewalk account credentials. The following list describes the API actions.

AWS IoT Wireless API actions for Sidewalk account

- [AssociateAwsAccountWithPartnerAccount](#)
- [DisassociateAwsAccountFromPartnerAccount](#)
- [GetPartnerAccount](#)
- [ListPartnerAccounts](#)
- [UpdatePartnerAccount](#)

For the complete list of the actions and data types available to create and manage AWS IoT Core for LoRaWAN resources, see the [AWS IoT Wireless API reference](#).

How to use the AWS CLI to add an account

You can use the AWS CLI to associate a Sidewalk account to your AWS account by using the [associate-aws-account-with-partner-account](#) command, as illustrated by the following example.

Note

You can also perform this procedure with the API by using the methods in the AWS API that correspond to the CLI commands shown here.

```
aws iotwireless associate-aws-account-with-partner-account \
  --sidewalk
  AmazonId="12345678901234",AppServerPrivateKey="a123b45c6d78e9f012a34cd5e6a7890b12c3d45e6f78a1b234c56d7
```

Next steps

Now that you've added the credentials and set up a wireless connectivity profile, you can add a destination for your Sidewalk device. You'll see the credentials that you added in the [Profiles](#) page of the AWS IoT console, on the **Sidewalk** tab. You'll define a role name and a rule name for the destination that can route messages sent from your devices. For more information, see [Add a destination for your Sidewalk device \(p. 1101\)](#).

Add a destination for your Sidewalk device

Before you can add an AWS IoT Core for LoRaWAN destination and create a rule for routing the messages sent from your Sidewalk device, you must create a wireless connectivity profile. To create the profile, first register your Sidewalk device, and then add the credentials to your AWS account. For more information, see [Add your Sidewalk account credentials \(p. 1100\)](#).

Creating a Sidewalk destination is similar to how you create a destination for your LoRaWAN devices. The following shows how you can create a destination by using the AWS Management Console or the API.

Add a destination by using the console

You can add your Sidewalk destination from the [Destinations](#) page of the AWS IoT console.

Specify the following fields when creating an AWS IoT Core for LoRaWAN destination, and then choose **Add destination**.

- **Destination details**

Enter a **Destination name** and an optional description for your destination. For the **Destination name**, enter **SidewalkDestination**. You can optionally enter a description, such as **This is a destination for Sidewalk devices**.

- **Rule name**

The AWS IoT rule that is configured to process the device's data. Your destination needs a rule to process the messages it receives. Enter a rule name (say **SidewalkRule**) and then choose **Copy** to copy the rule name that you'll enter when creating the AWS IoT rule. You can either choose **Create rule** to create the rule now or navigate to the **Rules** Hub of the AWS IoT console and create a rule with the name you copied.

For more information about AWS IoT rules for destinations, see [Create rules to process LoRaWAN device messages \(p. 1018\)](#).

- **Role name**

The IAM role that gives the device's data permission to access the rule named in **Rule name**. To create the IAM role, follow the steps described in [Create an IAM role for your destinations \(p. 1016\)](#). When creating the role:

- For **Select type of trusted entity**, choose **AWS service**, and then choose **IoT** as the service.
- Enter **SidewalkRole** for the **Role name**.
- Use the same policy document as described in [Create an IAM role for your destinations \(p. 1016\)](#).

For more information about IAM roles, see [Using IAM roles](#).

Add a destination by using the API

The following lists describe the API actions that perform the tasks associated with adding, updating, or deleting a destination.

AWS IoT Wireless API actions for service profiles

- [CreateDestination](#)
- [GetDestination](#)
- [ListDestinations](#)
- [UpdateDestination](#)
- [DeleteDestination](#)

For the complete list of actions and data types available to create and manage AWS IoT Core for LoRaWAN resources, see the [AWS IoT Wireless API reference](#).

How to use the AWS CLI to add a destination

You can use the AWS CLI to add a destination by using the `create-destination` command. The following example creates a destination.

```
aws iotwireless create-destination \
  --name SidewalkDestination \
  --expression-type RuleName \
  --expression SidewalkRule \
  --role-arn arn:aws:iam::123456789012:role/SidewalkRole
```

Running this command creates a destination with the specified destination name, rule name, and role name. For information about rule and role names for destinations, see [Create rules to process LoRaWAN device messages \(p. 1018\)](#) and [Create an IAM role for your destinations \(p. 1016\)](#).

For information about the CLIs that you can use, see [AWS CLI reference](#).

Next steps

Now that you've added the destination, you can create the destination rule for your Sidewalk device that will route messages to other services. For more information, see [Create rules to process Sidewalk device messages \(p. 1103\)](#).

Create rules to process Sidewalk device messages

AWS IoT rules can receive the messages from Sidewalk devices and route them to other services. [AWS IoT Core for LoRaWAN destinations \(p. 1015\)](#) associate a Sidewalk device with the rule that processes the device's message data to send to other services.

You can use an existing rule for your destination. In this section, we'll create the rule, `SidewalkRule`, that you specified when creating the Sidewalk destination, as described in [Add a destination for your Sidewalk device \(p. 1101\)](#). When creating the rule, we'll create an AWS Lambda action to republish the message to an AWS IoT topic.

Create a Sidewalk destination rule

Navigate to the [Rules Hub](#) of the AWS IoT console and perform the following steps.

1. Choose **Create a rule** to create a new rule for the destination.
2. Enter the name **SidewalkRule** for the **Name** and specify an optional **Description** for the rule (for example, `Sidewalk rule for lambda action to republish a topic`).
3. Change the default query statement to **SELECT *** so that any actions associated with the rule will be performed. Keep the SQL version to `2016-03-23`.
4. Under **Set one or more actions**, choose **Add action**.
5. For the rule action, choose **Send a message to a Lambda function** and then choose **Configure action**.
6. You can choose an existing Lambda function or create a new one. In this example, we'll create a Lambda function. Choose **Create a new Lambda function**.

Create your function using AWS Lambda

Choosing **Create a new Lambda function** opens the [Functions](#) page of the Lambda console. Perform the following steps.

1. To create your function, choose **Author from scratch**.
2. For **Function name**, enter a name (for example, `Sidewalk_Handler`), choose `Python 3.8` as **Runtime**, and then choose **Create function**.
3. Choose the `lambda.py` function in the **Code source** section of the console.
4. In the function body, delete any code inside the function body, and add a print statement for your Lambda function. You can also use `base64` to decode the `PayloadData` to receive the application data that your device sends to AWS IoT. The following shows an example Lambda function.

```
import json
import base64

def lambda_handler(event, context):

    message = json.dumps(event)
    print (message)
```

```
payload_data = base64.b64decode(event["PayloadData"])
print(payload_data)
print(int(payload_data,16))
```

5. To deploy your function code, choose **Deploy**.
6. Go back to the [Rules](#) Hub of the console and refresh the page. Choose the Lambda function that you created and choose **Add action**.

Republish a message to an AWS IoT topic

You can add a second action to republish a message to an AWS IoT topic from the [Rules](#) Hub of the console.

1. Choose **Add action**.
2. Choose **Republish a message to an AWS IoT topic** and choose **Configure action**.
3. Enter **project/sensor/observed** for the **Topic** and make sure the **Quality of Service** is set to **0 - The message is delivered zero or more times**.
4. Choose **Create Role**. Enter **SidewalkRepublishRole** for the role name and choose **Create Role**.
5. Choose **Add action**.

Both actions appear in the [Rules](#) Hub of the AWS IoT console.

6. Choose **Create rule**.

The rule appears on the [Rules](#) page that shows the list of rules.

Next steps

Now that you've created the destination rule for your Sidewalk device, you can connect your device and observe messages on the topic that you subscribed to. For more information, see [Connect your Sidewalk device and view uplink metadata format \(p. 1104\)](#).

Connect your Sidewalk device and view uplink metadata format

After you've added your Sidewalk credentials and added the destination, you can provision your Sidewalk endpoints and connect your device.

Connect your Sidewalk device

You can provision your device as a Sidewalk endpoint by generating the device certificates and application server certificates from the [Sidewalk Developer Service \(SDS\) console](#). For more information, see [Provision and Configure your Sidewalk Endpoints](#).

After you connect your device, you'll see your Sidewalk device in the [Devices](#) page of the AWS IoT console, on the **Sidewalk** tab. When your device is connected and starts sending data, you'll see the date and time for the **Last uplink received at** field.

View format of uplink messages

After you've connected your device, you can subscribe to the topic (for example, `project/sensor/observed`) that you specified when creating the Sidewalk destination rule, and observe uplink messages from the device. To subscribe to the topic, go to the [MQTT test client](#) on the [Test](#) page of the AWS IoT console, enter the topic name (for example, `project/sensor/observed`), and then choose **Subscribe**.

The following example shows the format of the uplink messages that are sent from Sidewalk devices to AWS IoT. The `WirelessMetadata` contains metadata about the message request.

```
{  
    "WirelessDeviceId": "8dcc5978df94950b57a58116c6f52e6",  
    "PayloadData": "AAAAAAA//8=",  
    "TransmitMode": "0",  
    "WirelessMetadata":  
        "WirelessMetadata(sidewalk=Sidewalk(seq=1983,  
            messageType=null, cmdExStatus=null,  
            nackExStatus=null))  
}
```

The following table shows a definition of the different parameters in the uplink metadata. For information about parameters `WirelessDeviceID`, `PayloadData`, and `messageType`, see [SidewalkDevice](#) and [SidewalkSendDataToDevice](#).

Uplink metadata parameters

Parameter	Description	Type	Required
<code>TransmitMode</code>	The transmission mode for data that is sent from the wireless device. Can be 0 for unconfirmed mode, 1 for confirmed, and 2 for unused.	Integer	Yes
<code>Sidewalk.CmdExStatus</code>	Command runtime status. Response-type messages shall include the status code, <code>COMMAND_EXEC_STATUS_SUCCESS</code> . However, notifications might not include the status code.	Enumeration	No
<code>Sidewalk.NackExStatus</code>	Response nack status, which can be <code>RADIO_TX_ERROR</code> or <code>MEMORY_ERROR</code> .	Array of strings	No

Event notifications for Amazon Sidewalk Integration for AWS IoT Core

AWS IoT Wireless can publish messages to notify you of events for Amazon Sidewalk devices that you onboard to AWS IoT Core. For example, you can be notified of events such as when the Sidewalk devices in your account have been provisioned or registered.

How your resources can be notified of events

Event notifications are published when certain events occur. For example, events are generated when your Sidewalk device is provisioned. Each event causes a single event notification to be sent. Event notifications are published over MQTT with a JSON payload. The content of the payload depends on the type of event.

Note

Event notifications are published at least once. It's possible for them to be published more than once. The ordering of event notifications is not guaranteed.

Policy for receiving Sidewalk event notifications

To receive event notifications, your device must use an appropriate policy that allows it to connect to the AWS IoT device gateway and subscribe to MQTT event topics. You must also subscribe to the appropriate topic filters.

The following is an example of the policy required for receiving Sidewalk events.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iot:Subscribe",  
                "iot:Receive"  
            ],  
            "Resource": [  
                "arn:aws:iotwireless:region:account:/$aws/iotwireless/events/  
device_registration_state/*",  
                "arn:aws:iotwireless:region:account:/$aws/iotwireless/events/proximity/*"  
            ]  
        }]  
}
```

Format of MQTT topics for Sidewalk events

To send you notifications of events for your Sidewalk devices, AWS IoT uses MQTT reserved topics that begin with a dollar (\$) sign. You can publish and subscribe to these reserved topics. However, you can't create new topics that begin with a dollar sign.

Note

MQTT topics are specific to your AWS account and use the format

`arn:aws:iotwireless:aws-region:AWS-account-ID:topic/Topic`. For more information, see [MQTT topics \(p. 93\)](#).

Reserved MQTT topics for Sidewalk devices use the format:

```
$aws/iotwireless/events/{Event name}/{Possible event Type}/sidewalk/Resource  
Type/Specific resource identifier/{SomeResourceId}
```

For example, the following event can be used to notify you of registration events:

```
$aws/iotwireless/events/device_registration_state/registered/sidewalk/  
sidewalk_accounts/amazon_id/{id}
```

If you've subscribed to these topics, you'll be notified when a message is published to one of the event notification topics. For more information, see [Reserved topics \(p. 95\)](#).

Enable events for Sidewalk devices

Before subscribers to the reserved topics can receive messages, you must enable event notifications. To do this, you can use the AWS Management Console, or the API or CLI.

- To enable event messages from the console, go to the [Settings](#) tab of the AWS IoT console, and in the **Sidewalk event notification** section, choose **Manage resource events**. You can then specify the events that you want to manage.
- To control which event types are published by using the API or CLI, call the [UpdateResourceEventConfiguration](#) API or use the `update-resource-event-configuration` CLI command. For example:

```
aws iotwireless update-resource-event-configuration \
--event-configurations "{\"DeviceRegistrationState\":{\"Sidewalk\": \
{\"AmazonIdEventTopic\": \"Enabled\"}}}"
```

Note

All quotation marks ("") are escaped with a backslash (\).

You can get the current event configuration by calling the [GetResourceEventConfiguration](#) API or by using the `get-resource-event-configuration` CLI command. For example:

```
aws iotwireless get-resource-event-configuration
```

Pricing for Sidewalk events

For information about pricing for subscribing to Sidewalk events and for receiving notifications, see [AWS IoT Core pricing](#).

Event types for Sidewalk devices

You can use the AWS Management Console or AWS IoT Wireless APIs to notify you of events for your Sidewalk devices. These events can be:

- Device events that notify you of changes to the state of your Sidewalk device, such as when the device has been registered and is ready to use.
- Proximity events that notify you when AWS IoT Wireless receives a notification from Amazon Sidewalk that a beacon has been received.

The following topics provide more information about these different events.

Topics

- [Device registration state events \(p. 1107\)](#)
- [Proximity events \(p. 1109\)](#)

Device registration state events

Device registration state events publish event notifications when there is a change in the device registration state, such as when a Sidewalk device has been provisioned or registered. The events provide you information about the different states that the device goes through from when it's provisioned to when it has been registered.

Enable notifications for device registration state events

Before subscribers to the device registration state reserved topics can receive messages, you must enable event notifications for them from the AWS Management Console, or by using the API or CLI. To receive notifications, provide your unique Sidewalk Amazon ID. All Sidewalk devices that are registered to this Amazon ID will then be notified of any device registration state events.

- To enable event notifications from the console:
 1. In the [Settings](#) tab of the AWS IoT console, go to the **Sidewalk event notification** section and choose **Manage resource events**.

2. Choose your **Amazon ID** and choose whether you want to enable event notifications for only **Sidewalk: device registration state** or for **Sidewalk: proximity** events as well. To disable events, choose your **Amazon ID** and clear the check boxes for events you want to disable. Choose **Confirm**.

You'll see the events that you've added in the **Sidewalk event notification** section.

3. To receive notifications, choose the events that you've added, and choose **Subscribe**.

All Sidewalk devices that are registered to this Sidewalk Amazon ID will receive a notification when an event occurs.

- To use the API or CLI to control which event types are published, call the [UpdateResourceEventConfiguration](#) API or use the [update-resource-event-configuration](#) CLI command. For example:

```
aws iotwireless update-resource-event-configuration \
--event-configurations "{\"DEVICE_REGISTRATION_STATE\":{\"Sidewalk\":{ \
\"AmazonIdEventTopic\": \"Enabled\"}}}"
```

To disable events, use the [UpdateResourceEventConfiguration](#) API or the [update-resource-event-configuration](#) CLI command and set `AmazonIdEventTopic` to `DISABLED`.

Format of MQTT topics for device registration state events

To notify you of device registration state events, you can subscribe to MQTT reserved topics that begin with a dollar (\$) sign. For more information, see [MQTT topics \(p. 93\)](#).

Reserved MQTT topics for Sidewalk device registration state events use the format:

```
$aws/iotwireless/events/device_registration_state/{Possible event Type}/
sidewalk/sidewalk_accounts/amazon_id/{id}
```

where {event Type} can be `provisioned` or `registered`.

You can also use the + wildcard character to subscribe to multiple topics at the same time. The + wildcard character matches any string in the level that contains the character. For example, if you want to be notified of all possible event types (`provisioned` and `registered`) and for all devices registered to a particular Amazon ID, you can use the following topic filter:

```
$aws/iotwireless/events/device_registration_state/+sidewalk/
sidewalk_accounts /amazon_id/+
```

Note

You can't use the wildcard character # to subscribe to the reserved topics. For more information about topic filters, see [Topic filters \(p. 94\)](#).

Message payload for device registration state events

After you enable notifications for device registration state events, event notifications are published over MQTT with a JSON payload. These events contain the following example payload:

```
{
  "eventId": "string",
  "eventType": "provisioned|registered",
  "WirelessDeviceId": "string",
  "timestamp": "timestamp",
  "operation": "create|deregister|register",
  "Sidewalk": {
    "AmazonId": "string",
```

```
    }
```

The payload contains the following attributes:

`eventId`

A unique event ID (string).

`eventType`

The type of event that occurred. Can be `provisioned` or `registered`.

`wirelessDeviceId`

Can be `provisioned` or `registered`.

`timestamp`

The UNIX timestamp of when the event occurred.

`operation`

The operation that triggered the event. Valid values are `create`, `register`, and `deregister`.

`sidewalk`

The Sidewalk Amazon ID for which you want to receive event notifications.

How device registration state events work

When you onboard your Sidewalk device with Amazon Sidewalk and AWS IoT Wireless, AWS IoT Wireless performs a `create` operation and adds your Sidewalk device to your AWS account. Your device then enters the provisioned state and the `eventType` becomes `provisioned`. For more information about onboarding your device, see [Getting started with Amazon Sidewalk Integration for AWS IoT Core \(p. 1099\)](#).

After the device has been provisioned, Amazon Sidewalk performs a `register` operation to register your Sidewalk device with AWS IoT Wireless. The registration process starts, where the encryption and session keys are set up with AWS IoT. When the device is registered, the `eventType` becomes `registered`, and your device is ready to use.

After the device has been `registered`, Sidewalk can send a request to `deregister` your device. AWS IoT Wireless then fulfills the request and changes the device state back to `provisioned`. For more information about the device states, see [DeviceState](#).

Proximity events

Proximity events publish event notifications when AWS IoT receives a beacon from the Sidewalk device. When your Sidewalk device approaches Amazon Sidewalk, beacons that are sent from your device are filtered by Amazon Sidewalk at regular intervals and received by AWS IoT Wireless. AWS IoT Wireless then notifies you of these events when a beacon is received.

Enable notifications for proximity events

Before subscribers to the proximity event topics can receive notifications, you must enable event notifications for them from the AWS Management Console, or by using the API or CLI. To receive notifications, provide your unique Sidewalk Amazon ID. All Sidewalk devices that are registered to this Amazon ID will then be notified of any proximity events.

- To enable event notifications from the console:
 1. In the [Settings](#) tab of the AWS IoT console, go to the **Sidewalk event notification** section and choose **Manage resource events**.
 2. Choose your **Amazon ID** and choose whether you want to enable event notifications for only **Sidewalk: proximity** events or for **Sidewalk: device registration state** events as well. To disable events, choose your **Amazon ID** and clear the check boxes for events you want to disable. Choose **Confirm**.You'll see the events that you've added in the **Sidewalk event notification** section.
 - 3. To receive notifications, choose the events that you've added, and choose **Subscribe**.All Sidewalk devices that are registered to this Sidewalk Amazon ID will receive a notification when an event occurs.
- To use the API or CLI to control which event types are published, call the [UpdateResourceEventConfiguration](#) API or use the `update-resource-event-configuration` CLI command. For example:

```
aws iotwireless update-resource--event-configuration \
  --event-configurations "{\"Proximity\":{\"Sidewalk\":{\"AmazonIdEventTopic\": \
    \"Enabled\"}}}"
```

To disable events, use the [UpdateResourceEventConfiguration](#) API or the `update-resource-event-configuration` CLI command and set `AmazonIdEventTopic` to `DISABLED`.

Format of MQTT topics for proximity events

To notify you of proximity events, you can subscribe to MQTT reserved topics that begin with a dollar (\$) sign. For more information, see [MQTT topics \(p. 93\)](#).

Reserved MQTT topics for Sidewalk proximity events use the format:

```
$aws/iotwireless/events/proximity/{Possible event Type}/sidewalk/
sidewalk_accounts/amazon_id/{id}
```

where {event Type} can be `beacon_discovered` or `beacon_lost`.

You can also use the + wildcard character to subscribe to multiple topics at the same time. The + wildcard character matches any string in the level that contains the character. For example, if you want to be notified of all possible event types (`beacon_discovered` and `beacon_lost`) and for all devices registered to a particular Amazon ID, you can use the following topic filter:

```
$aws/iotwireless/events/proximity/+(sidewalk/sidewalk_accounts /amazon_id/+
```

Note

You can't use the wildcard character # to subscribe to the reserved topics. For more information about topic filters, see [Topic filters \(p. 94\)](#).

Message payload for proximity events

After you enable notifications for proximity events, event messages are published over MQTT with a JSON payload. These events contain the following example payload:

```
{
  "eventId": "string",
  "eventType": "beacon_discovered|beacon_lost",
  "WirelessDeviceId": "string",
  "timestamp": "1234567890123",
```

```
    "Sidewalk": {  
        "AmazonId": "string"  
    }  
}
```

The payload contains the following attributes:

eventId

A unique event ID, which is a string.

eventType

The type of event that occurred. Can be `beacon_discovered` or `beacon_lost`.

wirelessDeviceId

Can be provisioned or registered.

timestamp

The UNIX timestamp of when the event occurred.

sidewalk

The Sidewalk Amazon ID for which you want to receive event notifications.

How proximity events work

Proximity events notify you when AWS IoT receives a beacon. Your Sidewalk devices can emit beacons any time. When your device is near Amazon Sidewalk, Sidewalk receives the beacons and forwards them to AWS IoT Wireless at regular time intervals. Amazon Sidewalk has configured this time interval as 10 minutes. When AWS IoT Wireless receives the beacon from Sidewalk, you'll be notified of the event.

Proximity events will notify you when a beacon is discovered or when a beacon is lost. You can configure the intervals at which you're notified of the proximity event.

Alexa Voice Service (AVS) Integration for AWS IoT

Alexa Voice Service (AVS) Integration for AWS IoT is a new feature that cost-effectively brings Alexa Voice to any connected device without incurring [messing costs](#). AVS for AWS IoT reduces the cost and complexity of integrating Alexa. This feature leverages AWS IoT to offload intensive computational and memory audio tasks from the device to the cloud. Because of the resulting reduction in the engineering bill of materials (eBoM) cost, device makers can now cost-effectively bring Alexa to resource-constrained IoT devices and make it possible for consumers to talk directly to Alexa in parts of their home, office, or hotel rooms for an ambient experience.

Currently, smart home IoT devices are built with low-cost microcontrollers (MCU) that have limited memory to run real-time operating systems. Previously, AVS solutions for Alexa built-in products required expensive application processor-based devices with more than 50 MB memory running on Linux or Android. These expensive hardware requirements made it cost-prohibitive to integrate Alexa Voice on resource-constrained IoT devices. AVS for AWS IoT enables Alexa built-in functionality on MCUs, such as the Arm Cortex-M series processors with less than 1 MB embedded RAM. To do so, AVS offloads memory and compute tasks to a virtual Alexa built-in device in the cloud. This reduces eBoM cost by up to 50 percent.

For more information about the Arm Cortex-M series processors, see [Arm](#) or [Wikipedia](#). For more information about hardware requirements for Alexa built-in products, see [Sizing Up CPU, Memory, and Storage for Your Alexa Built-in Device](#) on the Amazon Alexa developer portal.

Note

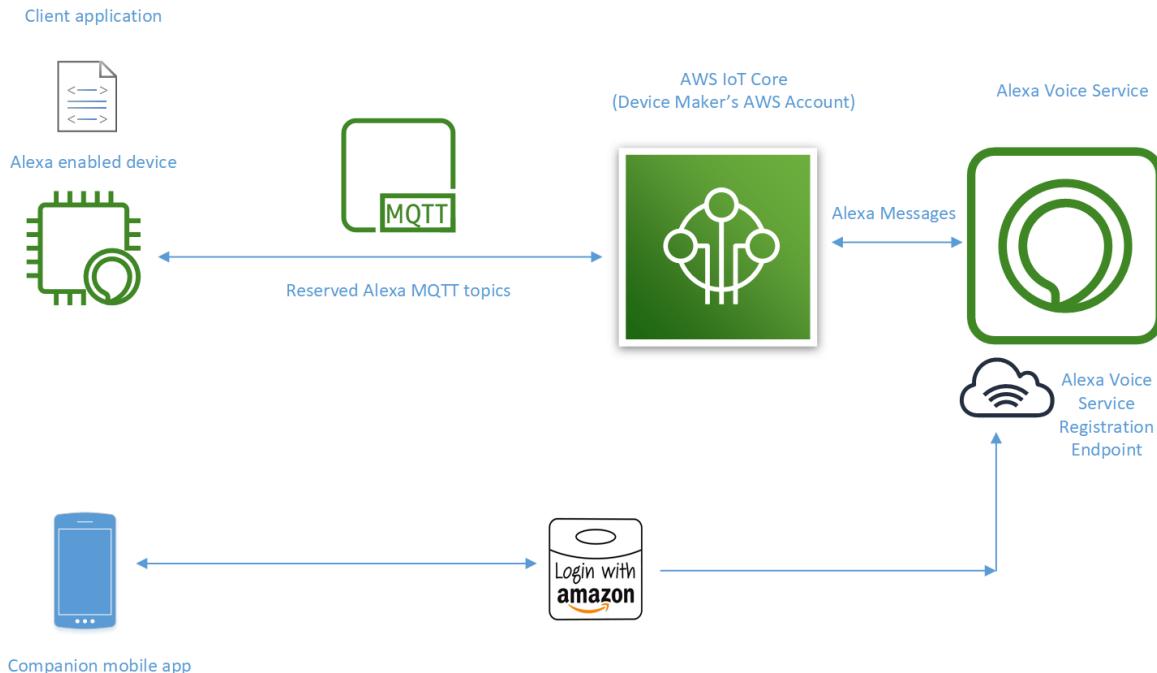
AVS for AWS IoT is available in all AWS Regions where AWS IoT is available except in the China (Beijing and Ningxia) Regions. For the current list of AWS Regions, see the [AWS Region Table](#).

AVS for AWS IoT has three components:

- A set of reserved MQTT topics to transfer audio messages between Alexa enabled devices and AVS.
- A virtual Alexa enabled device in the cloud that shifts tasks related to media retrieval, audio decoding, audio mixing, and state management from the physical device to the virtual device.
- A set of APIs that support receiving and sending messages over the reserved topics, interfacing with the device microphone and speaker, and managing device state.

The following diagram illustrates how these components work together. It also demonstrates how device makers use the Login with Amazon service to authenticate AVS.

Alexa Voice Service (AVS) Integration for AWS IoT Core



Device manufacturers have two options to get started with AVS Integration for AWS IoT .

- **Development kits** – Development kits launched by our partners make it easy to get started. The [NXP i.MX RT 106 A](#), [Qualcomm Home Hub 100 Development Kit for Amazon AVS](#), and [STM32 STEVAL-VOICE-UI](#) are some of the kits available on the market. You can find them on [Development Kits for AVS](#). The kits include out-of-the box connectivity to AWS IoT, AVS qualified Audio Algorithms for Far-Field voice pickup, Echo Cancellation, Alexa Wake Word, and AVS for AWS IoT application code. You can use the feature application code to quickly prototype a device and port the implementation to your chosen MCU design for testing and device production when you're ready.
- **Custom device-side application code** – Developers can also write a custom AVS for AWS IoT application by using the publicly available API. Documentation for this API is available on the [AVS developer page](#). You can download the FreeRTOS and AWS IoT Device SDK from the FreeRTOS console (<https://console.aws.amazon.com/freertos/>) or [GitHub](#).

To see an example of how to get started with a development kit, see [Getting Started with Alexa Voice Service \(AVS\) Integration for AWS IoT on an NXP Device](#).

Getting started with Alexa Voice Service (AVS) Integration for AWS IoT on an NXP device

With the NXP i.MX RT106A development kit, you can preview Alexa Voice Service (AVS) Integration for AWS IoT using a preconfigured NXP account. After you preview the functionality with the NXP account, you customize the firmware (application source code) to use your own account. This topic walks you through the steps to preview with the preconfigured account and to customize your device with your own account.

Topics

- [Preview Alexa Voice Service \(AVS\) Integration for AWS IoT with a preconfigured NXP account \(p. 1114\)](#)
- [Interact with Alexa \(p. 1124\)](#)
- [Use your AWS and Alexa Voice Service developer accounts to set up AVS for AWS IoT \(p. 1126\)](#)

Preview Alexa Voice Service (AVS) Integration for AWS IoT with a preconfigured NXP account

Prerequisites

To follow these steps, you need the following resources.

- [NXP i.MX RT106A development kit](#)

This kit is preloaded with software that enables both [zero touch setup \(p. 1124\)](#) and [user-guided setup \(p. 1115\)](#).

- A Mac, Windows 7 or 10, or Linux computer
- An Android or iOS mobile device
- The [Amazon Alexa app for iOS](#) or the [Amazon Alexa app for Android](#)
- An [Amazon Alexa account](#)

Turn on the development kit

Verify that your development kit box contains a USB Type-C to dual Type-A cable. Connect both of the USB-A connections to your computer. Connect the USB-C connector to your kit. Your configuration will look like the following image.



Note

If the box contains a quick start card, disregard it and refer to these instructions instead.

When the board has power, the status indicator LED lights up and displays various colors. These are status indicators for the various stages of the boot process. The colors and the blink rate indicate the status of the device. Your device is ready for setup when the status indicator light turns solid blue, as shown in the following image.



The development kit supports the following setups, depending on your environment.

- [User-guided setup \(p. 1115\)](#): Use this setup when the device arrives in factory state and doesn't meet the conditions for zero touch setup (ZTS).

You also use user-guided setup when someone has already performed ZTS on the device. ZTS can occur only once in the lifetime of a product.

- [Zero touch setup \(ZTS\) \(p. 1124\)](#): Use this setup when your environment meets the following conditions.
 - You purchased the kit from Amazon.com.
 - You didn't purchase the kit or receive the kit as a gift.
 - You've already installed a provisioner device in the Wi-Fi network that you're using with the kit.

A provisioner device is an Amazon device (such as an Echo (3rd Gen)) that is registered to an Amazon customer account.

For a list of Amazon devices that qualify as provisioning devices, see [Testing Your Device in Understanding Frustration-Free Setup](#).

- Your kit is within the Bluetooth Low Energy (BLE) range of the provisioner device.
- Your Wi-Fi credentials are available in the Amazon Wi-Fi locker.
- You have an [Alexa skill](#) linked to your Amazon account.
- You have implemented [Login with Amazon](#).

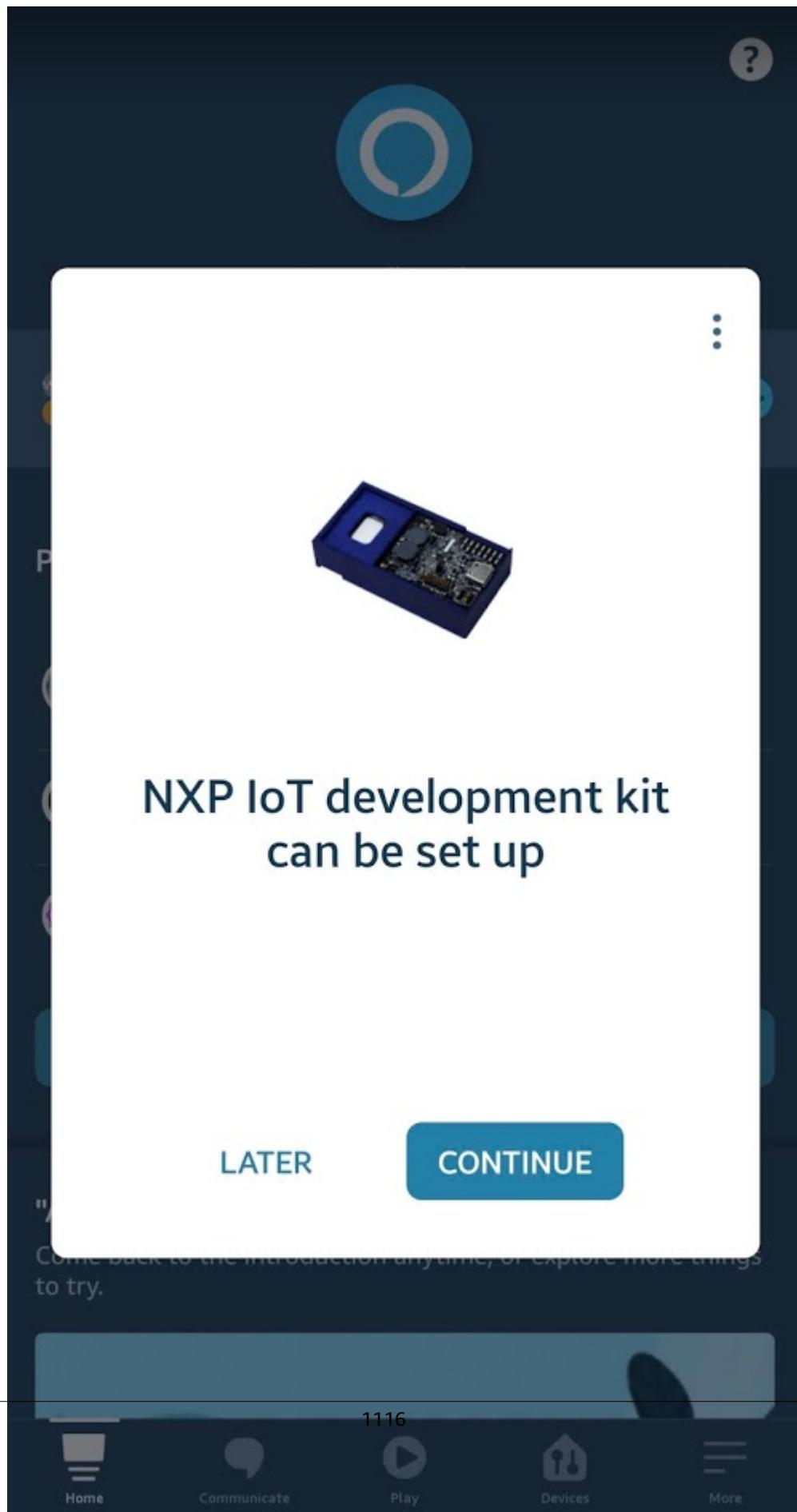
For more information about this kind of setup, see [Zero Touch Setup](#).

User-guided setup

When a kit that doesn't meet the requirements for ZTS turns on, it waits for user-guided setup to occur through the Amazon Alexa app on your phone. Make sure that the Amazon Alexa app is installed on your phone and that the Bluetooth and location permissions are enabled for the app.

The following procedure describes how to perform user-guided setup.

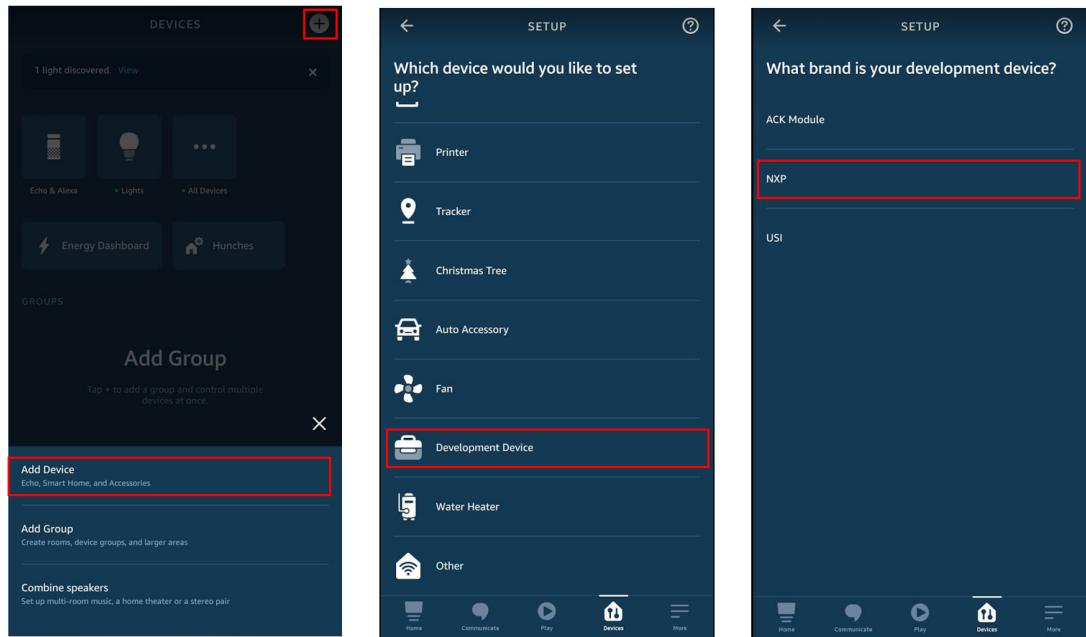
1. Open the Alexa App and log in to your Amazon Alexa account. The app detects that a nearby device is waiting for user-guided setup and displays the page in the following image. Choose **Continue**.



If you choose **Later** or if the app doesn't display this page, use the following steps to start user-guided setup.

1. Choose the **Devices** tab, and then select the plus sign (+) in the window that appears.
2. Choose **Add Device**.
3. Choose **Development Device**.
4. On the **What brand is your development device?** page, choose **NXP**, and then choose **Next**.

The following images show how the prompts described in these steps appear in the app.



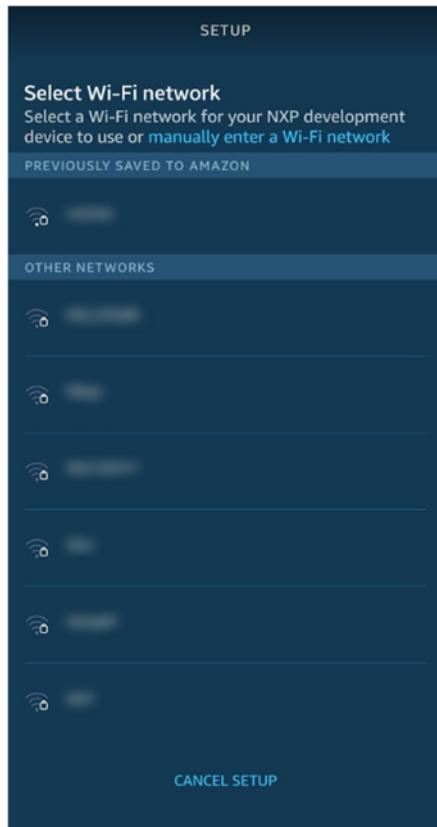
When the app connects to the device, the status indicator light blinks orange, as in the following images.



Note

If user guided setup is interrupted (for example, if you close the app), the device returns to discovery mode, and the status indicator light displays solid blue.

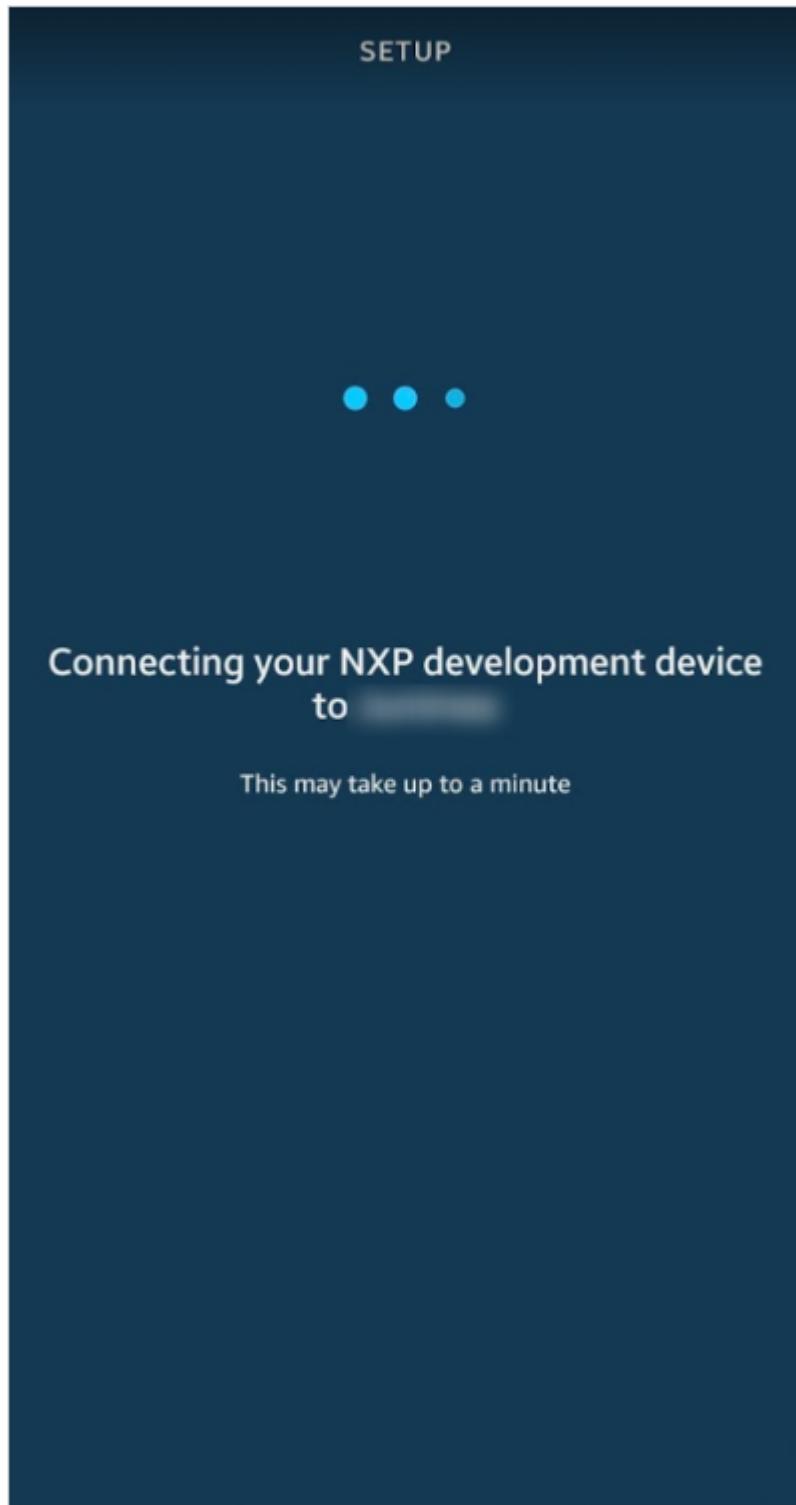
2. The app asks the kit to scan the environment for Wi-Fi networks and return a list of networks that it detects. Choose the network to which the device should connect. The following image shows how this list appears in the app.



Note

If you've already saved the selected network selected in your Amazon account, you don't need to enter the Wi-Fi password.

When you select the Wi-Fi network, the screen displays the following message as Wi-Fi provisioning and communication with the setup servers takes place: **Connecting your NXP development device to *Wi-Fi Network Name***. The following image shows how this screen appears in the app.



The status indicator light continues to blink orange until the kit's registration is complete. When registration is complete, the device says, "Your Alexa device is ready." The kit then reboots.

The following describes the steps that the kit takes after it reboots and reconnects to the Wi-Fi network that you selected.

1. As it reboots, the kit again displays various colors and alternates between blinks and solid colors as it progresses through the boot process.
2. The device then tries to reconnect to the Wi-Fi network that you selected. As it does this, the status indicator light blinks yellow at 500 millisecond (ms) intervals. After it connects to the Wi-Fi network, it blinks yellow faster, at 250 ms intervals. The following images show how this blinking appears in the kit.



3. The kit connects to AWS IoT. While it connects, the status indicator light blinks green at 500 ms intervals. When the kit is done connecting, the status indicator light blinks green at 250 ms intervals. The following images show how this blinking appears in the kit.



4. The kit plays a chime sound that indicates that you can use it to interact with Alexa.

When the kit connects to AWS IoT, the screen in the following image appears in the app.

SETUP



NXP light connected

NXP Development Kit has been added to your Alexa account. Next, let's continue setup.

The **NXP light connected** message appears in the app because the kit implements the smart home capabilities for an NXP light device.

Zero touch setup (ZTS)

If your environment fulfills all of the prerequisites for ZTS, the provisioning device discovers your kit and starts ZTS setup when you turn the kit on. The Amazon Alexa app is also a ZTS provisioner, so opening the Amazon Alexa app can also start ZTS setup.

As the provisioning process continues, the status indicator light state follows the same patterns as the ones described in the user-guided setup section. During provisioning, log messages are sent to the SLN-ALEXA-IOT console over its virtual COM port. When provisioning is complete, the kit plays the chime sound that indicates that you can use it to interact with Alexa.

Note

ZTS setup can occur only once in the lifetime of a device, even if you return it to factory settings.

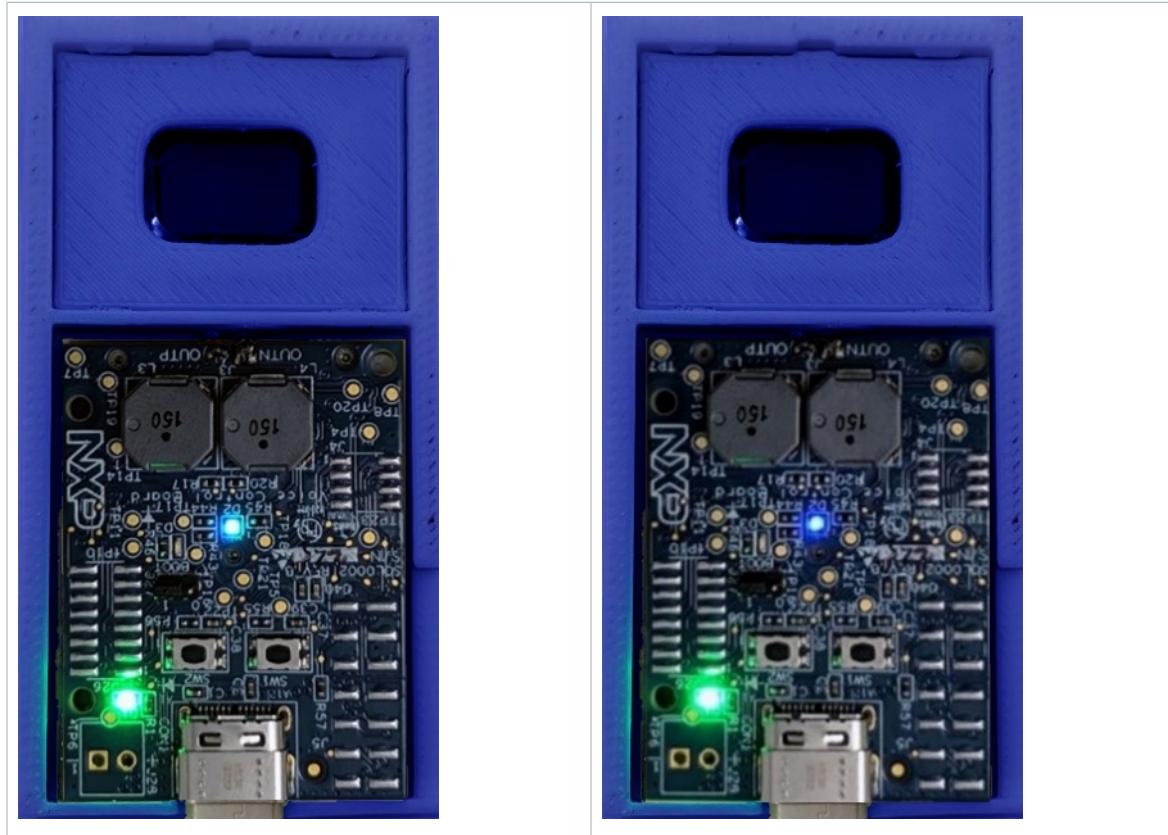
Interact with Alexa

You can start using the kit to interact with Alexa by asking it a question. Even a simple question, such as "Alexa, how is the weather?" goes through several states as Alexa processes and responds to it.

You see the first indication that the kit is listening when you speak the Alexa wake word. When the kit detects this word, the kit starts listening and sending information from the microphone to AVS through AWS IoT. The status indicator light displays solid cyan, as in the following image.



When the device finishes sending information from the microphone to AVS through AWS IoT, the device stops listening and switches to a thinking state. This state indicates that AVS is processing the question and is determining the best response. While the kit is in this state, the status indicator LED blinks cyan and blue at 200 ms intervals. The following images show how this blinking appears in the kit.



When the device finishes thinking, it starts to respond. Before the kit begins to speak, the status indicator light switches to a speaking state. The kit blinks cyan and blue at 500 ms intervals.

The response from Alexa plays out of the kit's speaker while the status indicator light blinks cyan and blue. Alexa describes weather conditions based on the location of your Alexa consumer account. When the response is complete, the status indicator light stops blinking and turns off. This indicates that the kit is in an idle state and waiting for the Alexa wake word.

Use your AWS and Alexa Voice Service developer accounts to set up AVS for AWS IoT

The preconfigured NXP account is only for evaluating the kit. When you use your own account, you get the following benefits.

- Full control of over the air (OTA) jobs and deployments, such as remote firmware updates.
- Control over AWS services.
- Customization of smart home skills.

To migrate from the preconfigured NXP account to your own account, download the **MCU Alexa Voice Solution Migration Guide** from the **Getting Started** section of the [EdgeReady MCU Based Solution for Alexa for IOT](#) page. Follow the steps in this guide.

Note

To download this file, you need an NXP account.

AWS IoT Device SDKs, Mobile SDKs, and AWS IoT Device Client

This page summarizes the AWS IoT Device SDKs, open-source libraries, developer guides, sample apps, and porting guides to help you build innovative IoT solutions with AWS IoT and your choice of hardware platforms.

These SDKs are for use on your IoT device. If you're developing an IoT app for use on a mobile device, see the [AWS Mobile SDKs \(p. 1129\)](#). If you're developing an IoT app or server-side program, see the [AWS SDKs \(p. 69\)](#).

AWS IoT Device SDKs

The AWS IoT Device SDKs include open-source libraries, developer guides with samples, and porting guides so that you can build innovative IoT products or solutions on your choice of hardware platforms.

These SDKs help you connect your IoT devices to AWS IoT using the MQTT and WSS protocols.

C++

AWS IoT C++ Device SDK

The AWS IoT C++ Device SDK allows developers to build connected applications using AWS and the AWS IoT APIs. Specifically, this SDK was designed for devices that are not resource constrained and require advanced features such as message queuing, multi-threading support, and the latest language features. For more information, see the following:

- [AWS IoT Device SDK C++ v2 on GitHub](#)
- [AWS IoT Device SDK C++ v2 Readme](#)
- [AWS IoT Device SDK C++ v2 Samples](#)
- [AWS IoT Device SDK C++ v2 API documentation](#)

Python

AWS IoT Device SDK for Python

The AWS IoT Device SDK for Python makes it possible for developers to write Python scripts to use their devices to access the AWS IoT platform through MQTT or MQTT over the WebSocket protocol. By connecting their devices to AWS IoT, users can securely work with the message broker, rules, and shadows provided by AWS IoT and with other AWS services like AWS Lambda, Kinesis, and Amazon S3, and more.

- [AWS IoT Device SDK for Python v2 on GitHub](#)
- [AWS IoT Device SDK for Python v2 Readme](#)
- [AWS IoT Device SDK for Python v2 Samples](#)

- [AWS IoT Device SDK for Python v2 API documentation](#)

JavaScript

AWS IoT Device SDK for JavaScript

The aws-iot-device-sdk.js package makes it possible for developers to write JavaScript applications that access AWS IoT using MQTT or MQTT over the WebSocket protocol. It can be used in Node.js environments and browser applications. For more information, see the following:

- [AWS IoT Device SDK for JavaScript v2 on GitHub](#)
- [AWS IoT Device SDK for JavaScript v2 Readme](#)
- [AWS IoT Device SDK for JavaScript v2 Samples](#)
- [AWS IoT Device SDK for JavaScript v2 API documentation](#)

Java

AWS IoT Device SDK for Java

The AWS IoT Device SDK for Java makes it possible for Java developers to access the AWS IoT platform through MQTT or MQTT over the WebSocket protocol. The SDK is built with shadow support. You can access shadows by using HTTP methods, including GET, UPDATE, and DELETE. The SDK also supports a simplified shadow access model, which allows developers to exchange data with shadows by just using getter and setter methods, without having to serialize or deserialize any JSON documents. For more information, see the following:

- [AWS IoT Device SDK for Java v2 on GitHub](#)
- [AWS IoT Device SDK for Java v2 Readme](#)
- [AWS IoT Device SDK for Java v2 Samples](#)
- [AWS IoT Device SDK for Java v2 API documentation](#)

AWS IoT Device SDK for Embedded C

Note

This SDK is intended for use by experienced embedded-software developers.

The AWS IoT Device SDK for Embedded C (C-SDK) is a collection of C source files under the MIT open source license that can be used in embedded applications to securely connect IoT devices to AWS IoT Core. It includes an MQTT, JSON Parser, and AWS IoT Device Shadow library. It is distributed in source form and intended to be built into customer firmware along with application code, other libraries and, optionally, an RTOS (Real Time Operating System).

For Fleet Provisioning, use the v4_beta_deprecated version of the AWS IoT Device SDK for Embedded C at https://github.com/aws/aws-iot-device-sdk-embedded-C/tree/v4_beta_deprecated. Please review the README in this branch for more details.

The AWS IoT Device SDK for Embedded C is generally targeted at resource constrained devices that require an optimized C language runtime. You can use the SDK on any operating system and host it on any processor type (for example, MCUs and MPUs).

For more information, see the following:

- [AWS IoT Device SDK for Embedded C on GitHub](#)
- [AWS IoT Device SDK for Embedded C Readme](#)
- [AWS IoT Device SDK for Embedded C Samples](#)

Earlier AWS IoT Device SDKs versions

These are earlier versions of AWS IoT Device SDKs that have been replaced by the newer versions listed above. These SDKs are receiving only maintenance and security updates. They will not be updated to include new features and should not be used on new projects.

- [AWS IoT C++ Device SDK on GitHub](#)
- [AWS IoT C++ Device SDK Readme](#)
- [AWS IoT Device SDK for Python v1 on GitHub](#)
- [AWS IoT Device SDK for Python v1 Readme](#)
- [AWS IoT Device SDK for Java on GitHub](#)
- [AWS IoT Device SDK for Java Readme](#)
- [AWS IoT Device SDK for JavaScript on GitHub](#)
- [AWS IoT Device SDK for JavaScript Readme](#)
- [Arduino Yún SDK on GitHub](#)
- [Arduino Yún SDK Readme](#)

AWS Mobile SDKs

The AWS Mobile SDKs provide mobile app developers platform-specific support for the APIs of the AWS IoT Core services, IoT device communication using MQTT, and the APIs of other AWS services.

Android

AWS Mobile SDK for Android

The AWS Mobile SDK for Android contains a library, samples, and documentation for developers to build connected mobile applications using AWS. This SDK also includes support for MQTT device communications and calling the APIs of the AWS IoT Core services. For more information, see the following:

- [AWS Mobile SDK for Android on GitHub](#)
- [AWS Mobile SDK for Android Readme](#)
- [AWS Mobile SDK for Android Samples](#)
- [AWS Mobile SDK for Android API reference](#)
- [AWSIoTClient Class reference documentation](#)

iOS

AWS Mobile SDK for iOS

The AWS Mobile SDK for iOS is an open-source software development kit, distributed under an Apache Open Source license. The AWS Mobile SDK for iOS provides a library, code samples, and documentation to help developers build connected mobile applications using AWS. This SDK also includes support for MQTT device communications and calling the APIs of the AWS IoT Core services. For more information, see the following:

- [AWS Mobile SDK for iOS on GitHub](#)
- [AWS Mobile SDK for iOS Readme](#)
- [AWS Mobile SDK for iOS Samples](#)
- [AWSIoT Class reference docs in the AWS Mobile SDK for iOS](#)

AWS IoT Device Client

The AWS IoT Device Client provides code to help your device connect to AWS IoT, perform fleet provisioning tasks, support device security policies, connect using secure tunneling, and process jobs on your device. You can install this software on your device to handle these routine device tasks so you can focus on your specific solution.

Note

The AWS IoT Device Client works with microprocessor-based IoT devices with x86_64 or ARM processors and common Linux operating systems.

C++

AWS IoT Device Client

For more information about the AWS IoT Device Client in C++, see the following:

- [AWS IoT Device Client in C++ source code on GitHub](#)
- [AWS IoT Device Client in C++ Readme](#)

Troubleshooting AWS IoT

The following information might help you troubleshoot common issues in AWS IoT.

Tasks

- [Diagnosing connectivity issues \(p. 1131\)](#)
- [Diagnosing rules issues \(p. 1133\)](#)
- [Diagnosing problems with shadows \(p. 1135\)](#)
- [Diagnosing Salesforce IoT input stream action issues \(p. 1136\)](#)
- [Fleet indexing troubleshooting guide \(p. 1137\)](#)
- [Troubleshooting "Stream limit exceeded for your AWS account" \(p. 1138\)](#)
- [AWS IoT Device Defender troubleshooting guide \(p. 1138\)](#)
- [Device Advisor troubleshooting guide \(p. 1141\)](#)
- [Troubleshooting device fleet disconnects \(p. 1142\)](#)
- [AWS IoT errors \(p. 1143\)](#)

Diagnosing connectivity issues

A successful connection to AWS IoT requires:

- A valid connection
- A valid and active certificate
- A policy that allows the desired connection and operation

Connection

How do I find the correct endpoint?

- The `endpointAddress` returned by `aws iot describe-endpoint --endpoint-type iot:Data-ATS`
or
 - The `domainName` returned by `aws iot describe-domain-configuration --domain-configuration-name "domain_configuration_name"`

How do I find the correct Server Name Indication (SNI) value?

The correct SNI value is the `endpointAddress` returned by the `describe-endpoint` or `describe-domain-configuration` commands. It's the same address as the endpoint in the previous step.

Authentication

Devices must be [authenticated \(p. 282\)](#) to connect to AWS IoT endpoints. For devices that use [X.509 client certificates \(p. 282\)](#) for authentication, the certificates must be registered with AWS IoT and be active.

How do my devices authenticate AWS IoT endpoints?

Add the AWS IoT CA certificate to your client's trust store. Refer to the documentation on [Server Authentication in AWS IoT Core](#) and then follow the links to download the appropriate CA certificate.

What is checked when a device connects to AWS IoT?

When a device attempts to connect to AWS IoT:

1. AWS IoT checks for a valid certificate and Server Name Indication (SNI) value.
2. AWS IoT checks to see that the certificate used is registered with the AWS IoT Account and that it has been activated.
3. When a device attempts to perform any action in AWS IoT, such as to subscribe to or publish a message, the policy attached to the certificate it used to connect is checked to confirm that the device is authorized to perform that action.

How can I validate a correctly configured certificate?

Use the OpenSSL s_client command to test a connection to the AWS IoT endpoint:

```
openssl s_client -connect custom_endpoint.iot.aws-region.amazonaws.com:8443 -  
CAfile CA.pem -cert cert.pem -key privateKey.pem
```

For more information about using openssl s_client, see [OpenSSL s_client documentation](#).

How do I check the status of a certificate?

- **List the certificates**

If you don't know the certificate ID, you can see the status of all your certificates by using the aws iot list-certificates command.

- **Show a certificate's details**

If you know the certificate's ID, this command shows you more detailed information about the certificate.

```
aws iot describe-certificate --certificate-id "certificateId"
```

- **Review the certificate in the AWS IoT Console**

In the [AWS IoT console](#), in the left menu, choose **Secure**, and then choose **Certificates**.

Choose the certificate that you are using to connect from the list to open its detail page.

In the certificate's detail page, you can see its current status.

The certificate's status can be changed by using the **Actions** menu in the upper-right corner of the details page.

Authorization

AWS IoT resources use [AWS IoT Core policies \(p. 314\)](#) to authorize those resources to perform [actions \(p. 315\)](#). For an action to be authorized, the specified AWS IoT resources must have a policy document attached to it that grants permission to perform that action.

I received a PUBNACK or SUBNACK response from the broker. What do I do?

Make sure that there is a policy attached to the certificate you are using to call AWS IoT. All publish/subscribe operations are denied by default.

Make sure the attached policy authorizes the [actions \(p. 315\)](#) you are trying to perform.

Make sure the attached policy authorizes the [resources \(p. 317\)](#) that are trying to perform the authorized actions.

I have an *AUTHORIZATION_FAILURE* entry in my logs.

Make sure that there is a policy attached to the certificate you are using to call AWS IoT. All publish/subscribe operations are denied by default.

Make sure the attached policy authorizes the [actions \(p. 315\)](#) you are trying to perform.

Make sure the attached policy authorizes the [resources \(p. 317\)](#) that are trying to perform the authorized actions.

How do I check what the policy authorizes?

In the [AWS IoT console](#), in the left menu, choose **Secure**, and then choose **Certificates**.

Choose the certificate that you are using to connect from the list to open its detail page.

In the certificate's detail page, you can see its current status.

In the left menu of the certificate's detail page, choose **Policies** to see the policies attached to the certificate.

Choose the desired policy to see its details page.

In the policy's details page, review the policy's **Policy document** to see what it authorizes.

Choose **Edit policy document** to make changes to the policy document.

Security and identity

When you provide the server certificates for AWS IoT custom domain configuration, the certificates have a maximum of four domain names.

For more information, see [AWS IoT Core endpoints and quotas](#).

Diagnosing rules issues

This section describes some of the things to check when you encounter a problem with rule.

Configuring CloudWatch Logs for troubleshooting

The best way to debug issues you are having with rules is to use CloudWatch Logs. When you enable CloudWatch Logs for AWS IoT, you can see which rules are triggered and their success or failure. You also get information about whether WHERE clause conditions match. For more information, see [Monitor AWS IoT using CloudWatch Logs \(p. 421\)](#).

The most common rules issue is authorization. The logs show if your role is not authorized to perform AssumeRole on the resource. Here is an example log generated by [fine-grained logging \(p. 404\)](#):

```
{
```

```
"timestamp": "2017-12-09 22:49:17.954",
"logLevel": "ERROR",
"traceId": "ff563525-6469-506a-e141-78d40375fc4e",
"accountId": "123456789012",
"status": "Failure",
"eventType": "RuleExecution",
"clientId": "iotconsole-123456789012-3",
"topicName": "test-topic",
"ruleName": "rule1",
"ruleAction": "DynamoAction",
"resources": {
    "ItemHashKeyField": "id",
    "Table": "trashbin",
    "Operation": "Insert",
    "ItemHashKeyValue": "id",
    "IsPayloadJSON": "true"
},
"principalId": "ABCDEFG1234567ABCD890:outis",
"details": "User: arn:aws:sts::123456789012:assumed-role/dynamo-testbin/5aUMInJH
is not authorized to perform: dynamodb:PutItem on resource: arn:aws:dynamodb:us-
east-1:123456789012:table/testbin (Service: AmazonDynamoDBv2; Status Code: 400; Error Code:
AccessDeniedException; Request ID: AKQJ987654321AKQJ123456789AKQJ987654321AKQJ987654321)"}
```

Here is a similar example log generated by [global logging \(p. 403\)](#):

```
2017-12-09 22:49:17.954 TRACEID:ff562535-6964-506a-e141-78d40375fc4e
PRINCIPALID:ABCDEFG1234567ABCD890:outis [ERROR] EVENT:DynamoActionFailure
TOPICNAME:test-topic CLIENTID:iotconsole-123456789012-3
MESSAGE:Dynamo Insert record failed. The error received was User:
arn:aws:sts::123456789012:assumed-role/dynamo-testbin/5aUMInJI is not authorized to
perform: dynamodb:PutItem on resource: arn:aws:dynamodb:us-east-1:123456789012:table/
testbin
(Service: AmazonDynamoDBv2; Status Code: 400; Error Code: AccessDeniedException; Request
ID: AKQJ987654321AKQJ987654321AKQJ987654321).
Message arrived on: test-topic, Action: dynamo, Table: trashbin, HashKeyField: id,
HashKeyValue: id, RangeKeyField: None, RangeKeyValue: 123456789012
No newer events found at the moment. Retry.
```

For more information, see [the section called “Viewing AWS IoT logs in the CloudWatch console” \(p. 421\)](#).

Diagnosing external services

External services are controlled by the end user. Before rule execution, make sure that the external services you have linked to your rule are set up and have enough throughput and capacity units for your application.

Diagnosing SQL problems

If your SQL query is not returning the data you expect:

- **Review the logs for error messages.**
- **Confirm that your SQL syntax matches the JSON document in the message.**

Review the object and property names used in the query with those used in the JSON document of the topic's message payload. For more information about the JSON formatting in SQL queries, see [JSON extensions \(p. 584\)](#).

- **Check to see if the JSON object or property names include reserved or numeric characters.**

For more information about reserved characters in JSON object references in SQL queries, see [JSON extensions \(p. 584\)](#).

Diagnosing problems with shadows

Diagnosing shadows

Issue	Troubleshooting guidelines
A device's shadow document is rejected with <code>Invalid JSON</code> document.	If you are unfamiliar with JSON, modify the examples provided in this guide for your own use. For more information, see Shadow document examples (p. 628) .
I submitted correct JSON, but none or only parts of it are stored in the device's shadow document.	Be sure you are following the JSON formatting guidelines. Only JSON fields in the desired and reported sections are stored. JSON content (even if formally correct) outside of those sections is ignored.
I received an error that the device's shadow exceeds the allowed size.	The device's shadow supports 8 KB of data only. Try shortening field names inside of your JSON document or simply create more shadows by creating more things. A device can have an unlimited number of things/shadows associated with it. The only requirement is that each thing name must be unique in your account.
When I receive a device's shadow, it is larger than 8 KB. How can this happen?	Upon receipt, the AWS IoT service adds metadata to the device's shadow. The service includes this data in its response, but it does not count toward the limit of 8 KB. Only the data for desired and reported state inside the state document sent to the device's shadow counts toward the limit.
My request has been rejected due to incorrect version. What should I do?	Perform a GET operation to sync to the latest state document version. When using MQTT, subscribe to the <code>./update/accepted</code> topic to be notified about state changes and receive the latest version of the JSON document.
The timestamp is off by several seconds.	The timestamp for individual fields and the whole JSON document is updated when the document is received by the AWS IoT service or when the state document is published onto the <code>./update/accepted</code> and <code>./update/delta</code> message. Messages can be delayed over the network, which can cause the timestamp to be off by a few seconds.
My device can publish and subscribe on the corresponding shadow topics, but when I attempt to update the shadow document over the HTTP REST API, I get HTTP 403.	Be sure you have created policies in IAM to allow access to these topics and for the corresponding action (UPDATE/GET/DELETE) for the credentials you are using. IAM policies and certificate policies are independent.

Issue	Troubleshooting guidelines
Other issues.	The Device Shadow service logs errors to CloudWatch Logs. To identify device and configuration issues, enable CloudWatch Logs and view the logs for debug information.

Diagnosing Salesforce IoT input stream action issues

Execution trace

How do I see the execution trace of a Salesforce action?

See the [Monitor AWS IoT using CloudWatch Logs \(p. 421\)](#) section. After you have activated the logs, you can see the execution trace of the Salesforce action.

Action success and failure

How do I check that messages have been sent successfully to a Salesforce IoT input stream?

View the logs generated by execution of the Salesforce action in CloudWatch Logs. If you see `Action executed successfully`, then it means that the AWS IoT rules engine received confirmation from the Salesforce IoT that the message was successfully pushed to the targeted input stream.

If you are experiencing problems with the Salesforce IoT platform, contact Salesforce IoT support.

What do I do if messages have not been sent successfully to a Salesforce IoT input stream?

View the logs generated by execution of the Salesforce action in CloudWatch Logs. Depending on the log entry, you can try the following actions:

`Failed to locate the host`

Check that the `url` parameter of the action is correct and that your Salesforce IoT input stream exists.

`Received Internal Server Error from Salesforce`

Retry. If the problem persists, contact Salesforce IoT Support.

`Received Bad Request Exception from Salesforce`

Check the payload you are sending for errors.

`Received Unsupported Media Type Exception from Salesforce`

Salesforce IoT does not support a binary payload at this time. Check that you are sending a JSON payload.

`Received Unauthorized Exception from Salesforce`

Check that the `token` parameter of the action is correct and that your token is still valid.

`Received Not Found Exception from Salesforce`

Check that the `url` parameter of the action is correct and that your Salesforce IoT input stream exists.

If you receive an error that is not listed here, contact AWS IoT Support.

Fleet indexing troubleshooting guide

Troubleshooting aggregation queries for the fleet indexing service

If you are having type mismatch errors, you can use CloudWatch Logs to troubleshoot the problem. CloudWatch Logs must be enabled before logs are written by the Fleet Indexing service. For more information, see [Monitor AWS IoT using CloudWatch Logs \(p. 421\)](#).

When you make aggregation queries on non-managed fields, you can only specify a field you defined in the `customFields` argument passed to `UpdateIndexingConfiguration` or `update-indexing-configuration`. If the field value is inconsistent with the configured field data type, this value is ignored when you perform an aggregation query.

The Fleet Indexing service emits an error log to CloudWatch Logs when a field cannot be indexed because of a mismatched type. The error log contains the field name, the value that could not be converted, and the thing name for the device. The following is an example error log:

```
{  
  "timestamp": "2017-02-20 20:31:22.932",  
  "logLevel": "ERROR",  
  "traceId": "79738924-1025-3a00-a669-7bec69f7f07a",  
  "accountId": "000000000000",  
  "status": "SucceededWithIssues",  
  "eventType": "IndexingCustomFieldFailed",  
  "thingName": "thing0",  
  "failedCustomFields": [  
    {  
      "Name": "attributeName1",  
      "Value": "apple",  
      "ExpectedType": "String"  
    },  
    {  
      "Name": "attributeName2",  
      "Value": "2",  
      "ExpectedType": "Boolean"  
    }  
  ]  
}
```

If a device has been disconnected for approximately an hour, the connectivity status `timestamp` value might be missing. For persistent sessions, the value might be missing after a client has been disconnected longer than the configured time-to-live (TTL) for the persistent session. The connectivity status data is indexed only for connections where the client ID has a matching thing name. (The client ID is the value used to connect a device to AWS IoT Core.)

Troubleshooting fleet metrics

Can't create a fleet metric

Downgrading data sources by updating fleet indexing configuration is not supported.

If you try to create a fleet metric with downgraded data sources (for example, previously the data sources were registry data, shadow data, and device connectivity data, and now the data sources are

registry data and shadow data and without device connectivity data), you'll see errors and you won't be able to create a fleet metric.

Modifying custom fields used by existing fleet metrics is not supported.

Can't see data points in CloudWatch

If you're able to create a fleet metric but you can't see data points in CloudWatch, it's likely that you don't have a thing that meets the query string criteria.

See this example command of how to create a fleet metric:

```
aws iot create-fleet-metric --metric-name "example_FM" --query-string  
"thingName:TempSensor* AND attributes.temperature>80" --period 60 --aggregation-field  
"attributes.temperature" --aggregation-type name=Statistics,values=count
```

If you don't have a thing that meets the query string criteria --query-string "thingName:TempSensor* AND attributes.temperature>80":

- With values=count, you'll be able to create a fleet metric and there'll be data points to show in CloudWatch. The data points of the value count is always 0.
- With values other than count, you'll be able to create a fleet metric but you won't see the fleet metric in CloudWatch and there'll be no data points to show in CloudWatch.

Troubleshooting "Stream limit exceeded for your AWS account"

If you see "Error: You have exceeded the limit for the number of streams in your AWS account.", you can clean up the unused streams in your account instead of requesting a limit increase.

To clean up an unused stream that you created using the AWS CLI or SDK:

```
aws iot delete-stream -stream-id value
```

For more details, see [delete-stream](#).

Note

You can use the `list-streams` command to find the stream IDs.

AWS IoT Device Defender troubleshooting guide

General

Q: Are there any prerequisites for using AWS IoT Device Defender?

A: If you want to use device-reported metrics, you must first deploy an agent on your AWS IoT connected devices or device gateways. Devices must provide a consistent client identifier or thing name.

Audit

Q: I enabled a check and my audit has been showing "In-Progress" for a long time. Is something wrong? When can I expect results?

A: When a check is enabled, data collection starts immediately. However, if your account has a large amount of data to collect (certificates, things, policies, and so on), the results of the check might not be available for some time after you have enabled it.

Detect

Q: How do I know the thresholds to set in an AWS IoT Device Defender security profile behavior?

A: Start by creating a security profile behavior with low thresholds and attach it to a thing group that contains a representative set of devices. You can use AWS IoT Device Defender to view the current metrics, and then fine-tune the device behavior thresholds to match your use case.

Q: I created a behavior, but it is not triggering a violation when I expect it to. How should I fix it?

A: When you define a behavior, you are specifying how you expect your device to behave normally. For example, if you have a security camera that only connects to one central server on TCP port 8888, you don't expect it to make any other connections. To be alerted if the camera makes a connection on another port, you define a behavior like this:

```
{
  "name": "Listening TCP Ports",
  "metric": "aws:listening-tcp-ports",
  "criteria": {
    "comparisonOperator": "in-port-set",
    "value": {
      "ports": [ 8888 ]
    }
  }
}
```

If the camera makes a TCP connection on TCP port 443, the device behavior would be violated and an alert would be triggered.

Q: One or more of my behaviors are in violation. How do I clear the violation?

A: Alarms clear after the device returns to expected behavior, as defined in the behavior profiles. Behavior profiles are evaluated upon receipt of metrics data for your device. If the device doesn't publish any metrics for more than two days, the violation event is set to `alarm-invalidated` automatically.

Q: I deleted a behavior that was in violation, but how do I stop the alerts?

A: Deleting a behavior stops all future violations and alerts for that behavior. Earlier alerts must be drained from your notification mechanism. When you delete a behavior, the record of violations of that behavior is retained for the same time period as all other violations in your account.

Device Metrics

Q: I'm submitting metrics reports that I know violate my behaviors, but no violations are being triggered. What's wrong?

A: Check that your metrics reports are being accepted by subscribing to the following MQTT topics:

```
$aws/things/THING_NAME/defender/metrics/FORMAT/rejected
$aws/things/THING_NAME/defender/metrics/FORMAT/accepted
```

where `THING_NAME` is the name of the thing reporting the metric and `FORMAT` is either "json" or "cbor", depending on the format of the metrics report submitted by the thing.

After you have subscribed, you should receive messages on these topics for each metric report submitted. A rejected message indicates that there was a problem parsing the metric report. An error message is included in the message payload to help you correct any errors in your metric report. An accepted message indicates the metric report was parsed properly.

Q: What happens if I send an empty metric in my metric report?

A: An empty list of ports or IP addresses is always considered in conformity with the corresponding behavior. If the corresponding behavior was in violation, the violation is cleared.

Q: Why do my device metric reports contain messages for devices that aren't in the AWS IoT registry?

If you have one or more security profiles attached to all things or to all unregistered things, AWS IoT Device Defender includes metrics from unregistered things. If you want to exclude metrics from unregistered things, you can attach the profiles to all registered devices instead of all devices.

Q: I'm not seeing messages from one or more unregistered devices even though I applied a security profile to all unregistered devices or all devices. How can I fix it?

Verify that you are sending a well-formed metrics report using one of the supported formats. For information, see [Device metrics document specification \(p. 906\)](#). Verify that the unregistered devices are using a consistent client identifier or thing name. Messages reported by devices are rejected if the thing name contains control characters or if the thing name is longer than 128 bytes of UTF-8 encoded characters.

Q: What happens if an unregistered device is added to the registry or a registered device becomes unregistered?

A: If a device is added to or removed from the registry:

- You see two separate violations for the device (one under its registered thing name, one under its unregistered identity) if it continues to publish metrics for violations. Active violations for the old identity stop appearing after two days, but are available in violations history for up to 14 days.

Q: Which value should I supply in the report ID field of my device metrics report?

A: Use a unique value for each metric report, expressed as a positive integer. A common practice is to use a [Unix epoch timestamp](#).

Q: Should I create a dedicated MQTT connection for AWS IoT Device Defender metrics?

A: A separate MQTT connection is not required.

Q: Which client ID should I use when connecting to publish device metrics?

For devices (things) that are in the AWS IoT registry, use the registered thing name. For devices that are not in the AWS IoT registry, use a consistent identifier when you connect to AWS IoT. This practice helps match the violations to the thing name.

Q: Can I publish metrics for a device with a different client ID?

It is possible to publish metrics on behalf of another thing. You can do this by publishing the metrics to the AWS IoT Device Defender reserved topic for that device. For example, Thing-1 would like to publish metrics for itself and also on behalf of Thing-2. Thing-1 collects its own metrics and publishes them on the MQTT topic:

```
$aws/things/Thing-1/defender/metrics/json
```

Thing-1 then obtains metrics from Thing-2 and publishes those metrics on the MQTT topic:

```
$aws/things/Thing-2/defender/metrics/json
```

Q: How many security profiles and behaviors can I have in my account?

A: See [AWS IoT Device Defender Endpoints and Quotas](#).

Q: What does a prototypical target role for an alert target look like?

A: A role that allows AWS IoT Device Defender to publish alerts on an alert target (SNS topic) requires two things:

- A trust relationship that specifies `iot.amazonaws.com` as the trusted entity.
- An attached policy that grants AWS IoT permission to publish on a specified SNS topic. For example:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": "sns:Publish",  
            "Resource": "<sns-topic-arn>"  
        }  
    ]  
}
```

- If the SNS topic used for publishing alerts is an encrypted topic, then along with the permission to publish to SNS topic, AWS IoT needs to be granted two more permissions. For example:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "sns:Publish",  
                "kms:Decrypt",  
                "kms:GenerateDataKey"  
            ],  
            "Resource": "<sns-topic-arn>"  
        }  
    ]  
}
```

Device Advisor troubleshooting guide

General

Q: Can I run multiple test suites in parallel?

A: No. Currently Device Advisor does not support running multiple test suites since only one Device Advisor endpoint is available per account. However, you can test multiple device types by sequencing the test suites one after the other.

Q: I saw from my device that the TLS connection got denied by Device Advisor. Is this expected?

A: Yes. Device Advisor denies the TLS connection before and after each test run. We recommend users to implement device retry mechanism in order to have a full-automated testing experience with Device Advisor. If you execute a test suite with more than one test case say - TLS connect, MQTT connect and MQTT publish then we recommend that you have a mechanism built for your device to try connecting to our test end point every 5 seconds for a minute to two. This will enable you to run multiple test cases in sequence in automated manner.

Q: Can I get a history of Device Advisor API calls made on my account for security analysis and operational troubleshooting purposes?

A: Yes. To receive a history of Device Advisor API calls made on your account, you simply turn on CloudTrail in the AWS IoT Management Console and filter the event source to be `iotdeviceadvisor.amazonaws.com`.

Q: How do I view Device Advisor logs in CloudWatch?

Logs generated during a test suite run are uploaded to CloudWatch if you add the required policy (for example, [CloudWatchFullAccess](#)) to your service role (see [Setting up \(p. 948\)](#)). A log group "aws/iot/deviceadvisor/\$testSuiteId" will be created. In this log group, two log streams will be created if there is at least one test case in your test suite. One is named "\$testRunId" and includes logs of actions taken before and after executing the test cases in your test suite, such as setup and cleanup steps. Another is "\$suiteRunId_\$testRunId" which is specific to a test suite run. Events sent from devices and AWS IoT Core will be logged to this log stream.

Q: What is the purpose of the device permission role?

A: Device Advisor stands between your test device and AWS IoT Core to simulate test scenarios. It accepts connections and messages from your test devices and forwards them to AWS IoT Core by assuming your device permission role and initiating a connection on your behalf. It's important to make sure the device role permissions are the same as those on the certificate you use for running tests. AWS IoT certificate policies are not enforced when Device Advisor initiates a connection to AWS IoT Core on your behalf by using the device permission role. However, the permissions from the device permission role you set are enforced.

Q: What Regions is Device Advisor supported in?

A: Device Advisor is supported in us-east-1, us-west-2, ap-northeast-1, and eu-west-1 Regions.

Q: What if I see inconsistent results?

A: One of the primary causes of inconsistent results is setting a test's `EXECUTION_TIMEOUT` to a value that is too low. For more information about recommended and default `EXECUTION_TIMEOUT` values, see [Device Advisor test cases](#).

Q: What MQTT protocol does Device Advisor support?

A: Device Advisor supports MQTT with X509 client certificates.

Troubleshooting device fleet disconnects

AWS IoT device fleet disconnects can happen for multiple reasons. This article explains how to diagnose a disconnect reason and how to handle disconnects caused by regular maintenance of AWS IoT service or a throttling limit.

To diagnose the disconnect reason

You can check the [AWSLogLogsV2](#) log group in [CloudWatch](#) to identify the disconnect reason in the `disconnectReason` field of the log entry.

You can also use AWS IoT's [lifecycle events](#) feature to identify the disconnect reason. If you've subscribed to [lifecycle's disconnect event](#) (`$aws/events/presence/disconnected/clientId`), you'll get a notification from AWS IoT when the disconnect happens. You can identify the disconnect reason in the `disconnectReason` field of the notification.

For more information, see [CloudWatch AWS IoT log entries](#) and [Lifecycle events](#).

To troubleshoot disconnects due to AWS IoT service maintenance

Disconnects caused by AWS IoT's service maintenance are logged as SERVER_INITIATED_DISCONNECT in AWS IoT's lifecycle event and CloudWatch. To handle these disconnects, adjust your client-side setup to make sure your devices can be automatically reconnected to the AWS IoT platform.

To troubleshoot disconnects due to a throttling limit

Disconnects caused by a throttling limit are logged as THROTTLED in AWS IoT's lifecycle event and CloudWatch. To handle these disconnects, you can request [message broker limit increases](#) as the device count grows.

For more information, see [AWS IoT Core Message Broker](#).

AWS IoT errors

This section lists the error codes sent by AWS IoT.

Message broker error codes

Error code	Error description
400	Bad request.
401	Unauthorized.
403	Forbidden.
503	Service unavailable.

Identity and security error codes

Error code	Error description
401	Unauthorized.

Device shadow error codes

Error code	Error description
400	Bad request.
401	Unauthorized.
403	Forbidden.
404	Not found.
409	Conflict.
413	Request too large.
422	Failed to process request.
429	Too many requests.
500	Internal error.
503	Service unavailable.

AWS IoT quotas

For AWS IoT Core quotas information, see [AWS IoT Core Endpoints and Quotas](#) in the *AWS General Reference*.