AWS IoT

Developer Guide

AWS IoT: Developer Guide

Table of Contents

What Is AWS IoT?

AWS IoT provides secure, bi-directional communication between Internet-connected things (such as sensors, actuators, embedded devices, or smart appliances) and the AWS cloud. This enables you to collect telemetry data from multiple devices and store and analyze the data. You can also create applications that enable your users to control these devices from their phones or tablets.

AWS IoT Components

AWS IoT consists of the following components:

Device gateway

Enables devices to securely and efficiently communicate with AWS IoT.

Message broker

Provides a secure mechanism for things 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. You can use the HTTP REST interface to publish.

Rules engine

Provides message processing and integration with other AWS services. You can use a SQL-based language to select data from message payloads, process and send the data to other services, such as Amazon S3, Amazon DynamoDB, and AWS Lambda. You can also use the message broker to republish messages to other subscribers.

Security and Identity service

Provides shared responsibility for security in the AWS cloud. Your things must keep their credentials safe in order 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.

Thing registry

Sometimes referred to as the device registry. Organizes the resources associated with each thing. You register your things and associate up to three custom attributes with each thing. You can also associate certificates and MQTT client IDs with each thing to improve your ability to manage and troubleshoot your things.

Thing shadow

Sometimes referred to as a device shadow. A JSON document used to store and retrieve current state information for a thing (device, app, and so on).

Thing Shadows service

Provides persistent representations of your things in the AWS cloud. You can publish updated state information to a thing shadow, and your thing can synchronize its state when it connects. Your things can also publish their current state to a thing shadow for use by applications or devices.

How to Get Started with AWS IoT

- To learn more about AWS IoT, see How AWS IoT Works (p. 3).
- To learn how to connect a thing to AWS IoT, see Getting Started with AWS IoT (p. 19).

Accessing AWS IoT

AWS IoT provides the following interfaces to create and interact with your things:

- **AWS Command Line Interface (AWS CLI)**—Run commands for AWS IoT on Windows, OS X, and Linux. These commands allow you to create and manage things, certificates, rules, 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 Command Line Interface Reference*.
- **AWS IoT API**—Build your IoT applications using HTTP or HTTPS requests. These API allow you to programmatically create and manage things, 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.
- **AWS IoT Device SDKs**—Build applications that run on your devices that send messages to and receive messages from AWS IoT. For more information see, AWS IoT SDKs

Related Services

AWS IoT integrates directly with the following AWS services:

- **Amazon Simple Storage Service**—Provides scalable storage in the AWS cloud. For more information, see Amazon S3.
- **Amazon DynamoDB**—Provides managed NoSQL databases. For more information, see Amazon DynamoDB.
- **Amazon Kinesis**—Enables real-time processing of streaming data at a massive scale. For more information, see Amazon Kinesis.
- **AWS Lambda**—Runs your code on virtual servers from Amazon EC2 in response to events. For more information, see AWS Lambda.
- **Amazon Simple Notification Service**—Sends or receives notifications. For more information, see Amazon SNS.
- **Amazon Simple Queue Service**—Stores data in a queue to be retrieved by applications. For more information, see Amazon SQS.

How AWS IoT Works

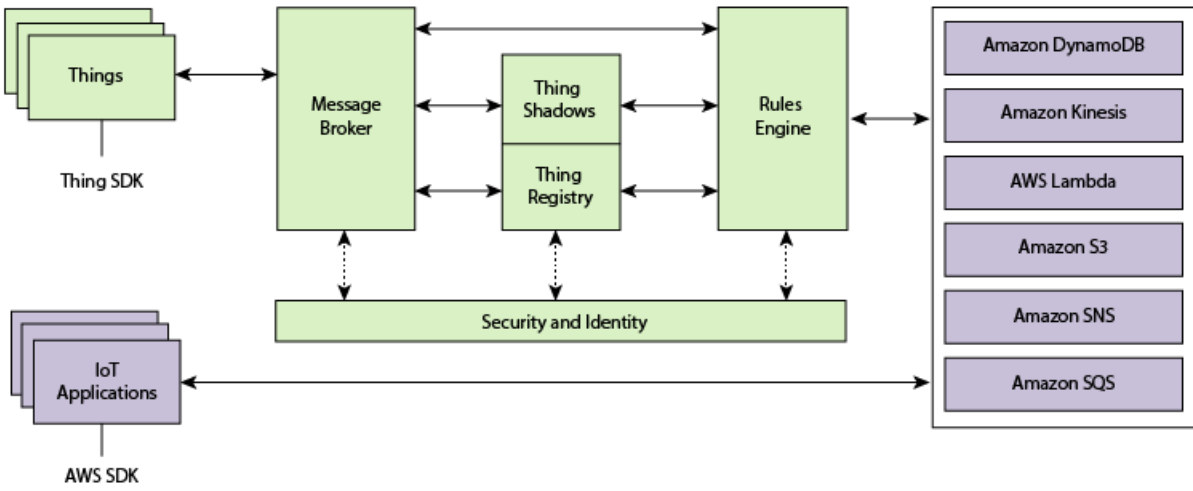
AWS IoT enables Internet-connected things to connect to the AWS cloud and lets applications in the cloud interact with Internet-connected things. Common IoT applications either collect and process telemetry from devices or enable users to control a device remotely.

Things report their state by publishing messages, in JSON format, on MQTT topics. Each MQTT topic has a hierarchical name that identifies the thing whose state is being updated. When a message is published on an MQTT topic, the message is sent to the AWS IoT MQTT message broker, which is responsible for sending all messages published on an MQTT topic to all clients subscribed to that topic.

Communication between a thing and AWS IoT is protected through the use of X.509 certificates. AWS IoT can generate a certificate for you or you can use your own. In either case, the certificate must be registered and activated with AWS IoT, and then copied onto your thing. When your thing communicates with AWS IoT, it presents the certificate to AWS IoT as a credential.

We recommend all things that connect to AWS IoT have an entry in the thing registry. The thing registry stores information about a thing and the certificates that are used by the thing to secure communication with AWS IoT.

You can create rules that define one or more actions to perform based on the data in a message. For example, you can insert, update, or query a DynamoDB table or invoke a Lambda function. Rules use expressions to filter messages. When a rule matches a message, the rules engine invokes the action using the selected properties. Rules also contain an IAM role that grants AWS IoT permission to the AWS resources used to perform the action.



Each thing has a thing shadow that stores and retrieves state information. Each item in the state information has two entries: the state last reported by the thing and the desired state requested by an application. An application can request the current state information for a thing. The shadow responds to the request by providing a JSON document with the state information (both reported and desired), metadata, and a version number. An application can control a thing by requesting a change in its state. The shadow accepts the state change request, updates its state information, and sends a message to indicate the state information has been updated. The thing receives the message, changes its state, and then reports its new state.

AWS IoT Button Quickstarts

The two quickstarts in this section show you how to configure and use the AWS IoT button. You can use the AWS IoT button wizard in the AWS Lambda console to easily and quickly configure your AWS IoT button. The AWS Lambda console contains a blueprint that will automate the process of setting up your AWS IoT button by:

- Creating and activating an X.509 certificate and private key for authenticating with AWS IoT.
- Walking you through the configuration of your AWS IoT button in order to connect to your Wi-Fi network.
- Walking you through the copying of your certificate and private key to your AWS IoT button.
- Creating and attaching to the certificate an AWS IoT policy that gives the button permission to make calls to AWS IoT.
- Creating an AWS IoT rule that invokes a Lambda function when your AWS IoT button is pressed.
- Creating an IAM role and policy that allows the Lambda function to send email messages using Amazon SNS.
- Creating a Lambda function that sends an email message to the address specified in the Lambda function code.

You can also configure the AWS IoT button by using an AWS CloudFormation template. The second quickstart shows you how to configure the AWS IoT resources required to process the MQTT messages that are sent when the AWS IoT button is pressed, by using an AWS CloudFormation template.

If you do not have a button, you can purchase one here. For more information about AWS IoT, see What Is AWS IoT (p. 1).

Topics

AWS IoT Button Wizard Quickstart

The AWS IoT button wizard is a Lambda blueprint, so you need to sign in to the AWS Lambda console in order to use it. If you do not have an AWS account, you can create one by following these steps.

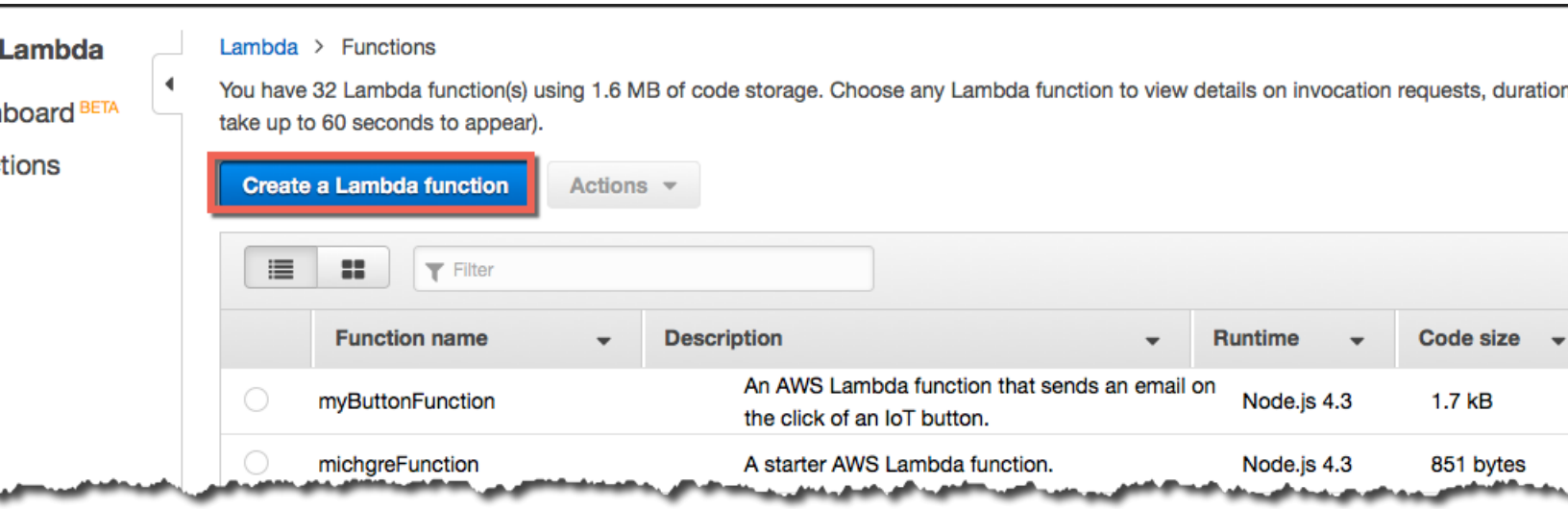
To create an AWS account

1. Open the AWS home page and choose **Create an AWS Account**.
2. Follow the online instructions. Part of the sign-up procedure involves receiving a phone call and entering a PIN using your phone's keypad.

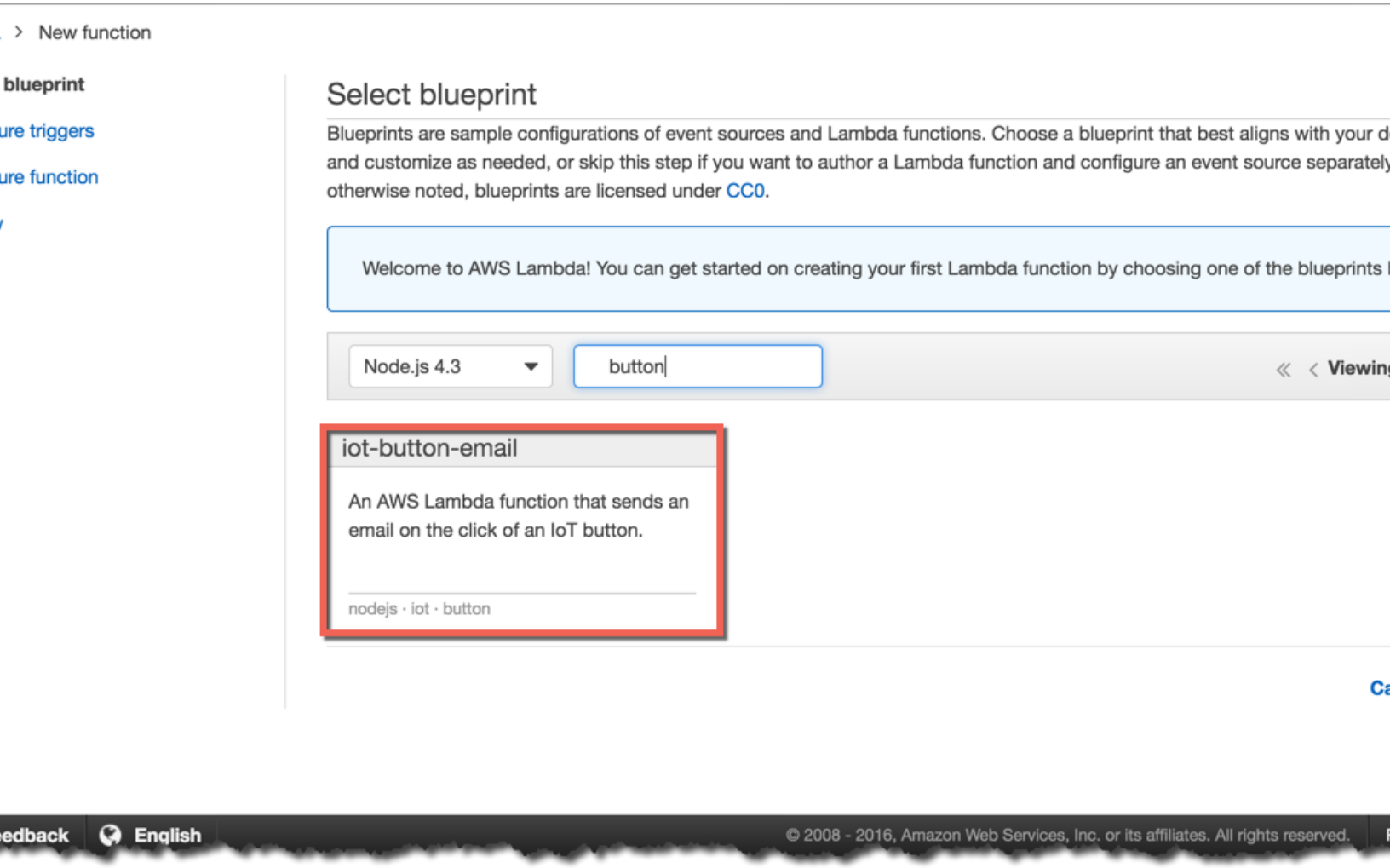
To configure the AWS IoT Button

1. Sign in to the AWS Management Console and open the AWS Lambda console.
2. If this is your first time in the AWS Lambda console, you will see the following page. Choose the **Get Started Now** button.

If you have used the AWS Lambda console before, you will see the following page. Choose the **Create a Lambda function** button.



3. On the **Select blueprint** page, from the **Runtime** drop-down menu, choose **Node.js 4.3**. In the filter text box, type `button`. To choose the **iot-button-email** blueprint, double-click it or choose the **Next** button.

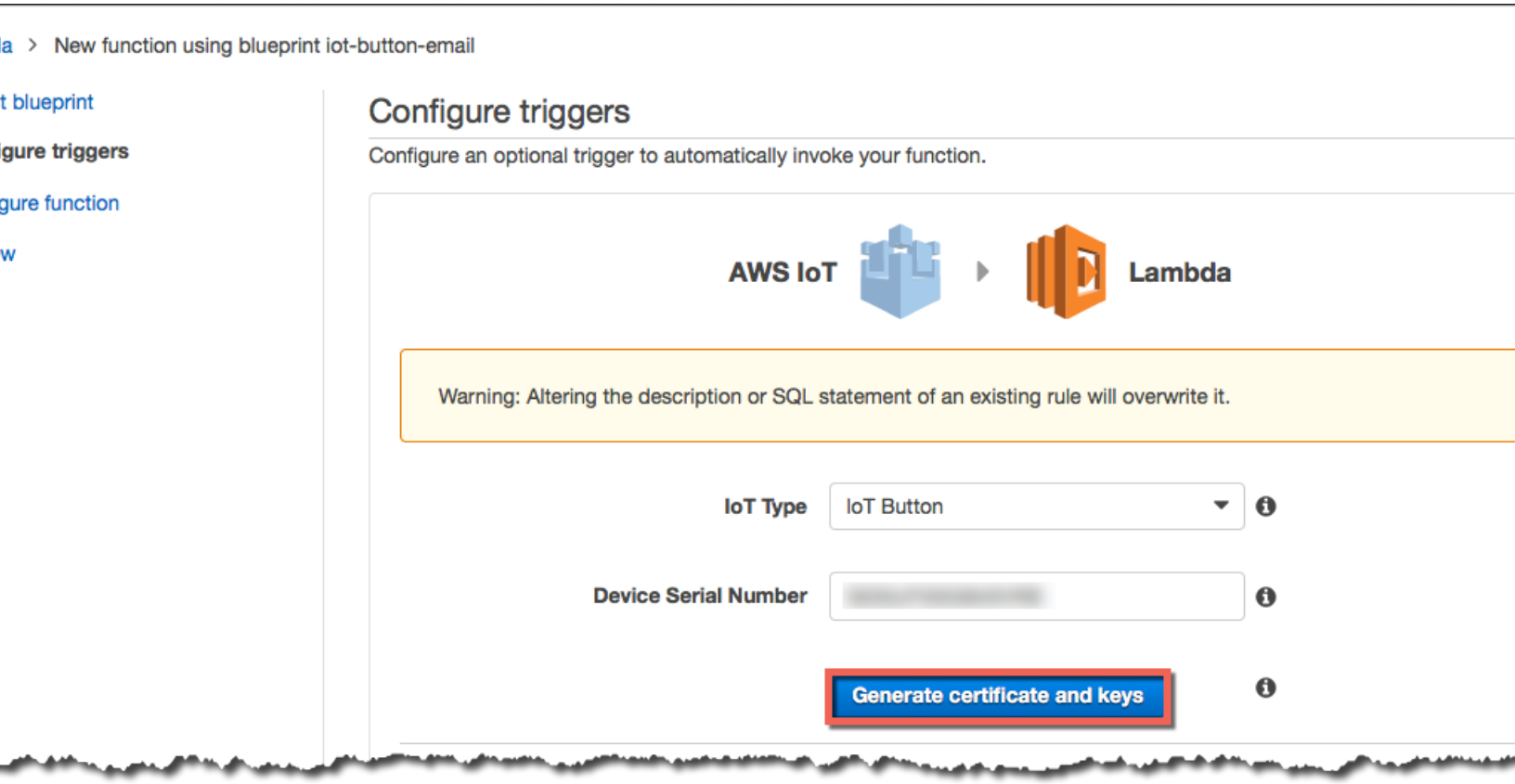


4. On the **Configure triggers** page, from the **IoT Type** drop-down menu, choose **IoT Button**.

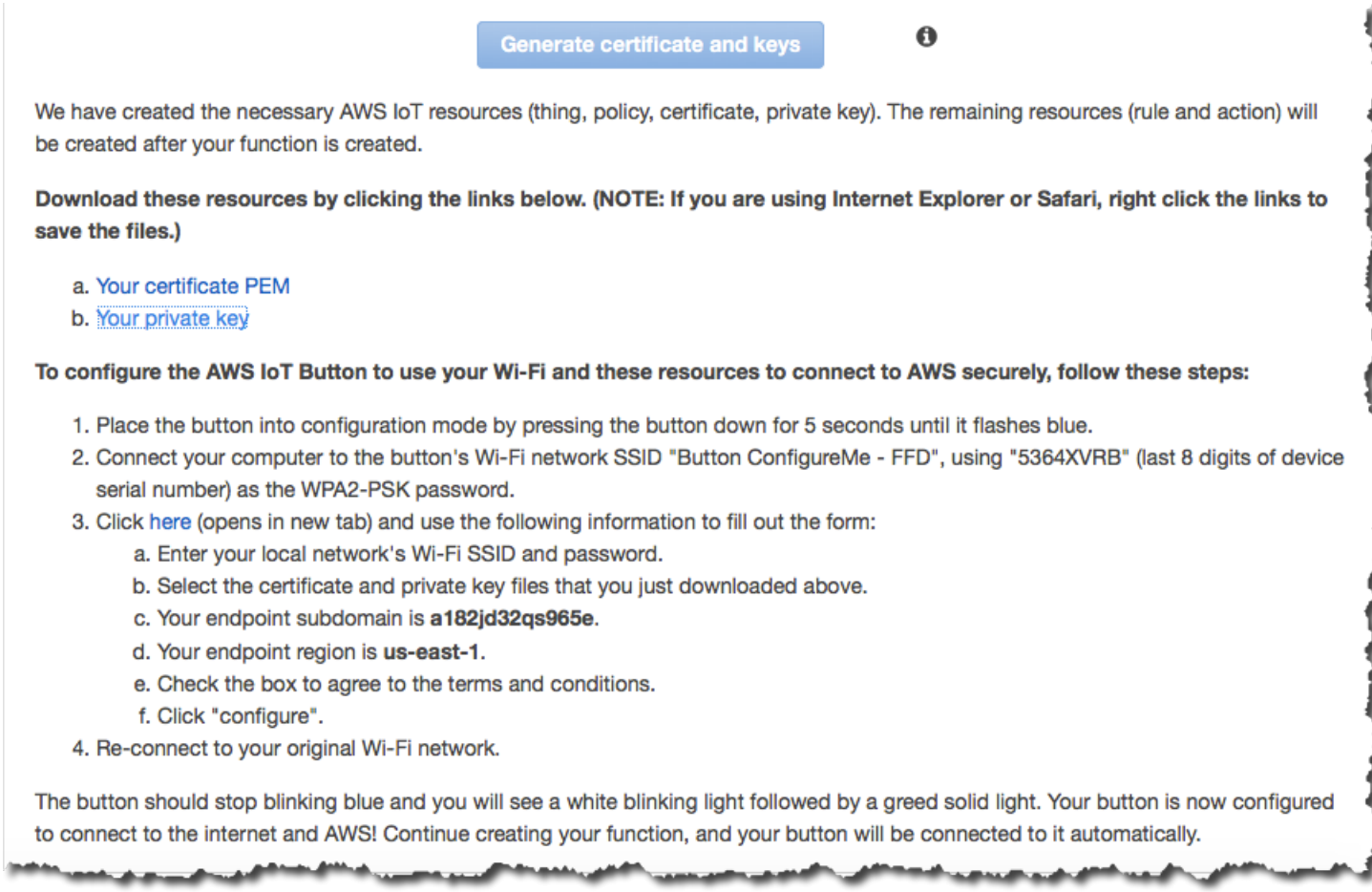
   Type the serial number for your device. You'll find the device serial number (DSN) on the back of the button.

   Choose **Generate certificate and keys**.

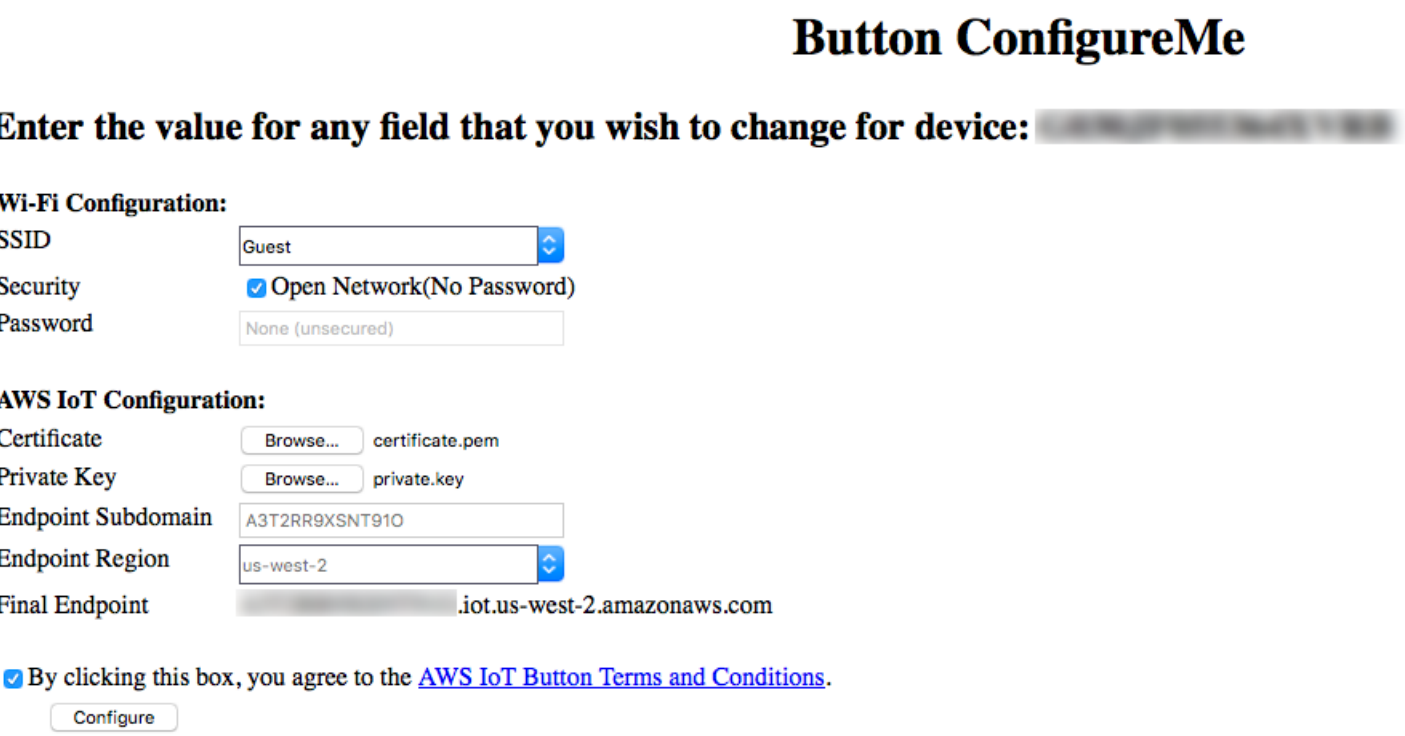
   Note
   You only need to generate a certificate and private key once. Then you can navigate to http://192.168.0.1/index.html in a browser to configure your button.



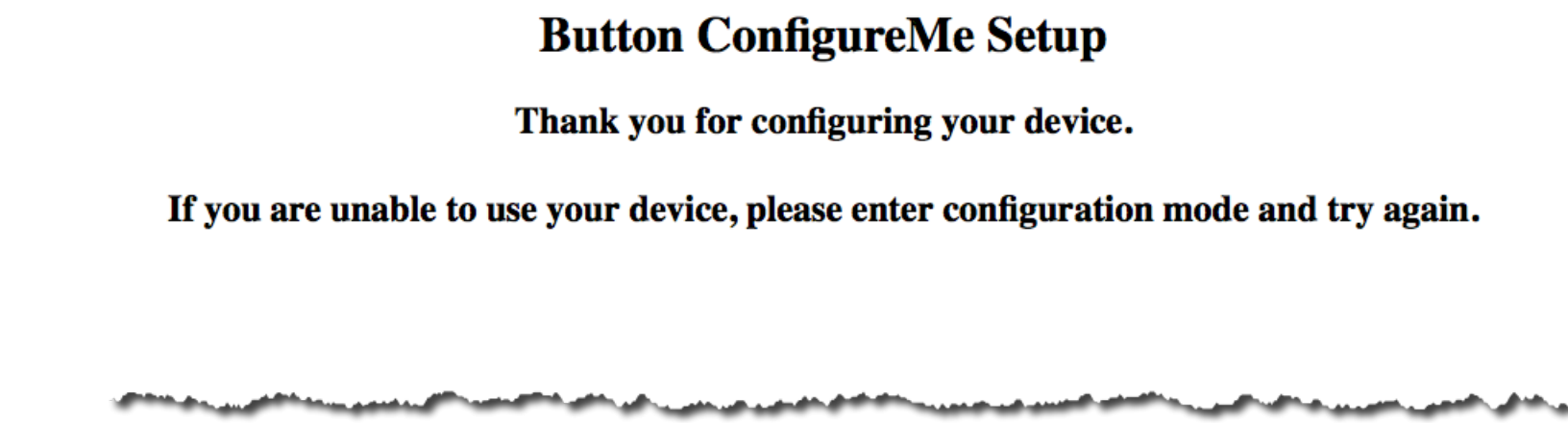
   Use the links on the page to download the device certificate and the private key.



The page also includes instructions for configuring your AWS IoT button. On step 3, you will choose a link to open a web page that allows you to connect the AWS IoT button to your network. Under **Wi-Fi Configuration**, type the network ID (SSID) and network password for your Wi-Fi network. Under **AWS IoT Configuration**, choose the certificate and private key you downloaded earlier. This will copy your certificate and private key to your AWS IoT button. Select the check box to agree to the AWS IoT button terms and conditions, and then choose the **Configure** button.

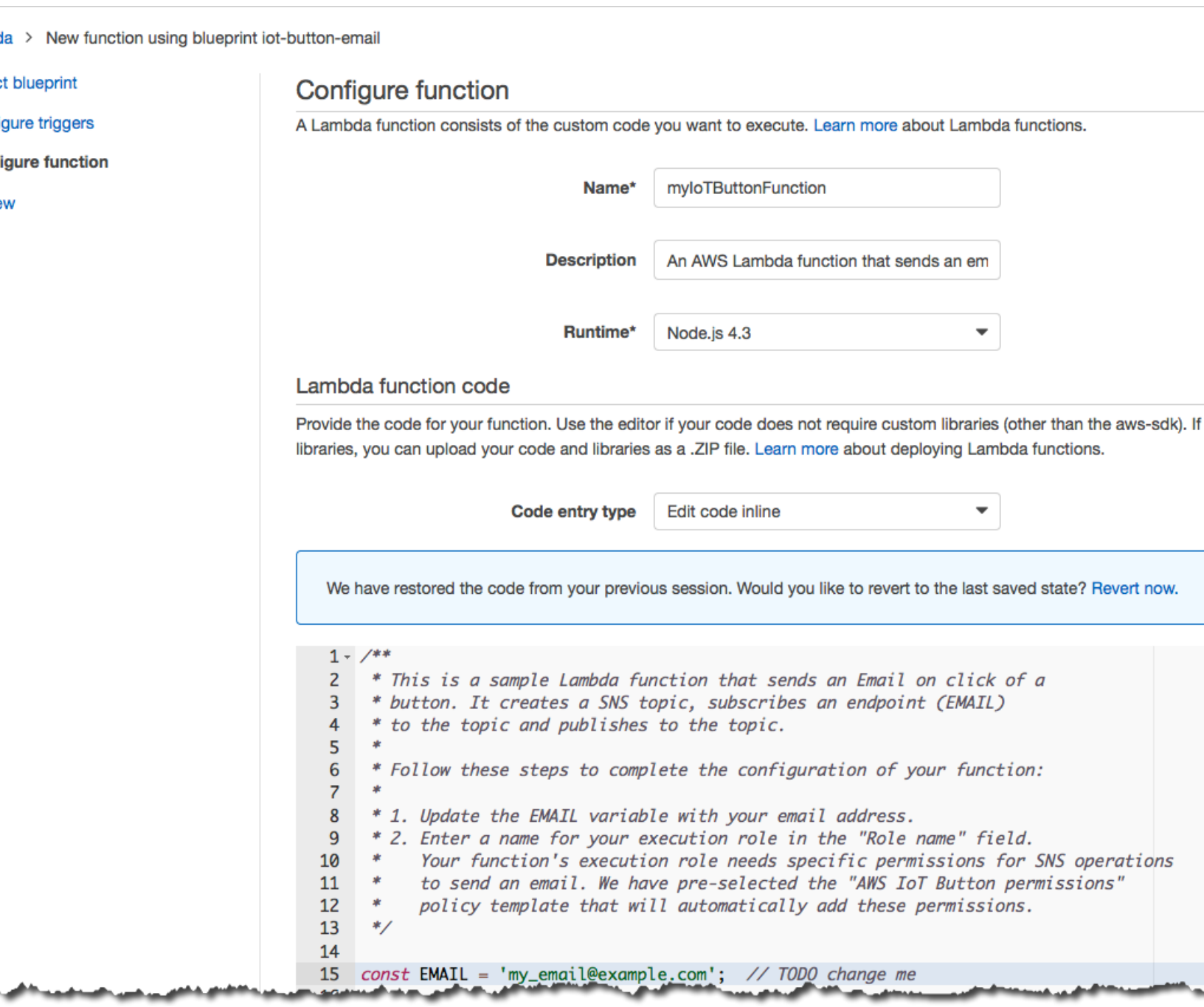


A configuration confirmation page will be displayed.



5. Close the **Configure** tab and go back to the AWS Lambda console page. Choose **Enable trigger**, and then choose **Next**.

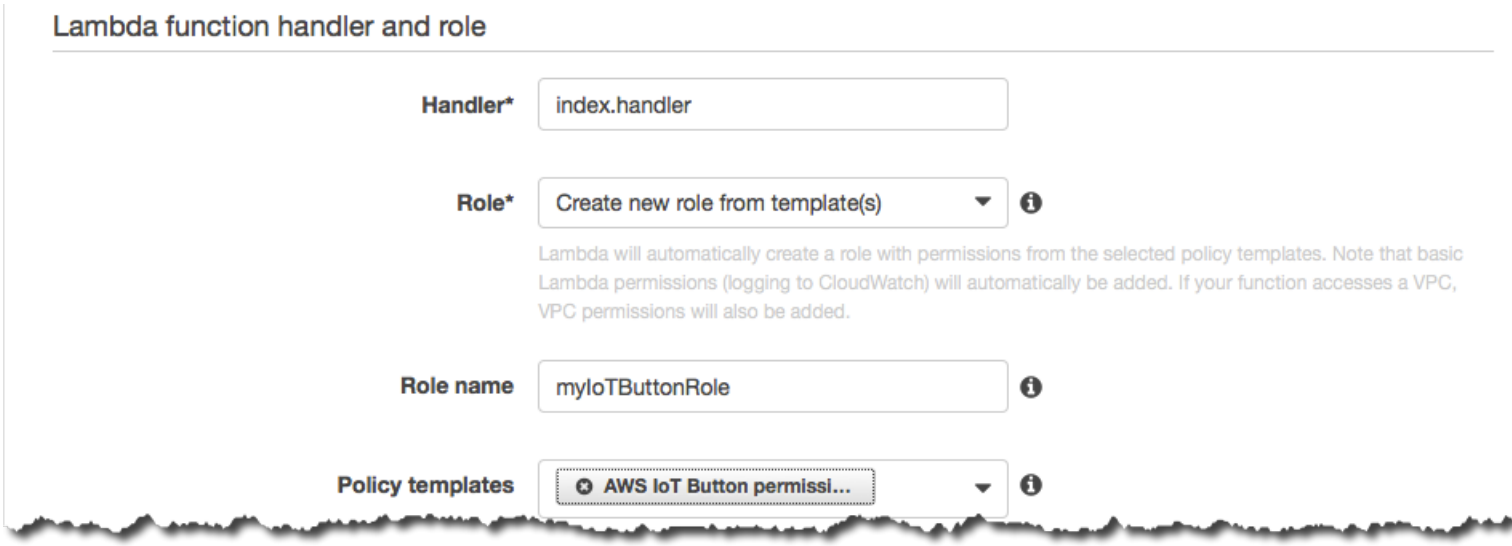
   On the **Configure function** page, type a name for your function. The description, runtime, and Lambda function code will be entered for you.



In the Lambda function code, replace the example email address with your own email address.

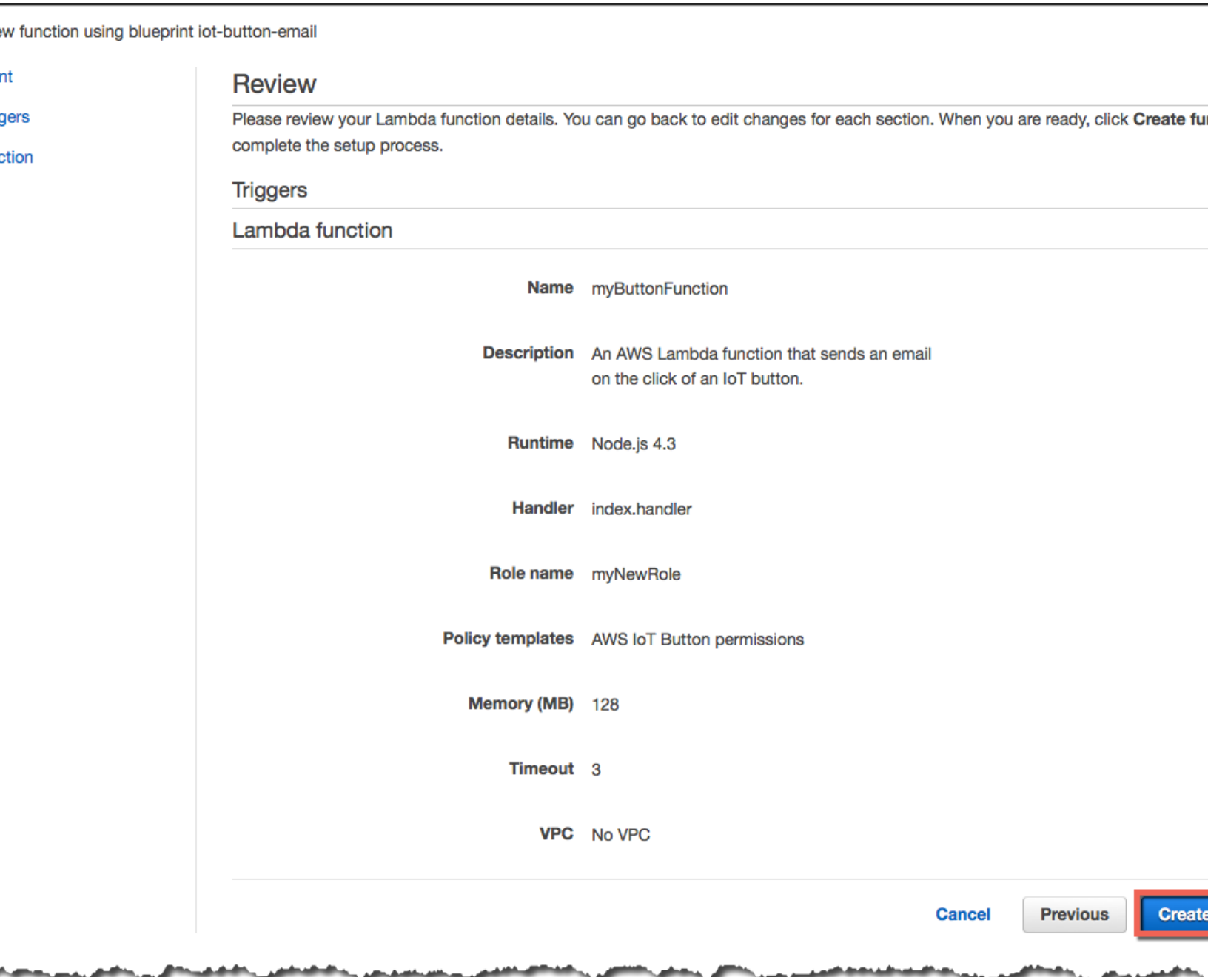
```
/**
 * This is a sample Lambda function that sends an Email on click of a
 * button. It creates a SNS topic, subscribes an endpoint (EMAIL)
 * to the topic and publishes to the topic.
 *
 * Follow these steps to complete the configuration of your function:
 *
 * 1. Update the EMAIL variable with your email address.
 * 2. Enter a name for your execution role in the "Role name" field.
 *    Your function's execution role needs specific permissions for SNS operations
 *    to send an email. We have pre-selected the "AWS IoT Button permissions"
 *    policy template that will automatically add these permissions.
 */

const EMAIL = 'my_email@example.com';  // TODO change me

const AWS = require('aws-sdk');
const SNS = new AWS.SNS({ apiVersion: '2010-03-31' });

function findExistingSubscription(topicArn, nextToken, cb) {
    const params = {
        TopicArn: topicArn,
        NextToken: nextToken || null,
    };
    SNS.listSubscriptionsByTopic(params, (err, data) => {
        if (err) {
```

In the **Lambda function handler and role** section, from the **Role** drop-down menu, choose **Create new role from template(s)**. Type a unique name for the role.

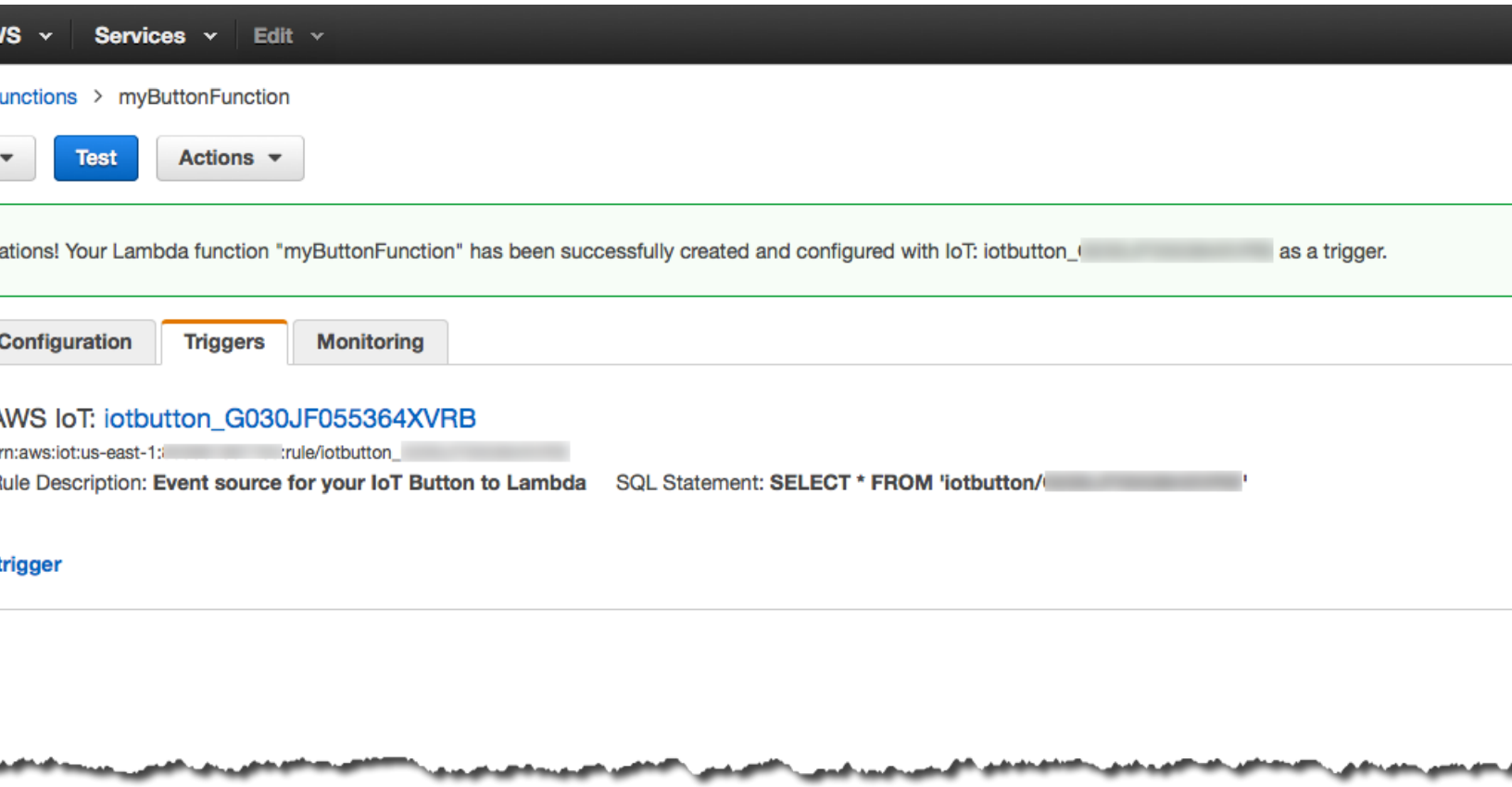


At the bottom of the page, choose **Next**.

Review the settings for the Lambda function, and then choose **Create function**.



You should see a page that confirms your Lambda function has been created:



6. To test your Lambda function, choose the **Test** button. After about a minute, you should receive an email message with `AWS Notification - Subscription Confirmation` in the subject line. Choose the link in the email message to confirm the subscription to an SNS topic created by the Lambda function. When AWS IoT receives a message from your button, it will send a message to Amazon SNS. The Lambda function created a subscription to the Amazon SNS topic using the email address you added in the code. When Amazon SNS receives a message on this Amazon SNS topic, it will forward the message to your subscribed email address.

Press your button to send a message to AWS IoT. The message will cause your Lambda rule to be triggered, and then your Lambda function will be invoked. The Lambda function will check to see if your SNS topic exists. The Lambda function will then send the contents of the message to the Amazon SNS topic. Amazon SNS will then forward the message to the email address you specified in the Lambda function code.

AWS IoT Button AWS CloudFormation Quickstart

When the AWS IoT button is pressed, it sends basic information about the button to an Amazon SNS topic. The topic then forwards that information to you in an email message. This quickstart will show you how to use an AWS CloudFormation template to configure your AWS IoT button.

You will need an AWS account and an AWS IoT button to complete the steps in this quickstart.

1. Use the AWS IoT console to create an AWS IoT certificate:

   a. Open the AWS IoT console.

b. If a **Welcome** page appears, choose **Get started**.

c. In the AWS region selector, choose the AWS region where you want to create the AWS IoT certificate (for example, US East (N. Virginia)). You will be creating all supporting AWS resources (additional AWS IoT resources and an Amazon SNS resource) in the same AWS region.

d. On the **Dashboard**, in the left navigation pane, choose **Security**, then choose **Certificates**.

e. On the **Certificates** pane, choose **Create**.

f. Choose **One-click certificate creation** - **Create certificate**.

g. On the **Certificate created** page, choose **Download** for the certificate, private key and the root CA for AWS IoT, save each of them to your computer, and then choose **Activate**.

h. Choose **Done**.

i. On the **Certificates** page, choose the certificate you just created.

j. In the **Details** pane, make a note of the certificate ARN value (for example, `arn:aws:iot:region-ID:account-ID:cert/random-ID`). You will need it later in this procedure.

2. Use the AWS CloudFormation console at https://console.aws.amazon.com/cloudformation/ to create the AWS IoT resources, an Amazon SNS resource, and an IAM role:

   a. Save the following AWS CloudFormation template file named AWSIoTButtonQuickStart.template to your computer.

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Description": "Creates required AWS resources to allow an AWS IoT button to send
 information through an Amazon Simple Notification Service (Amazon SNS) topic to an
 email address.",
  "Parameters": {
    "IoTButtonDSN": {
   "Type": "String",
   "AllowedPattern": "G030JF05[0-9][0-5][0-9][1-7][0-9A-HJ-NP-X][0-9A-HJ-NP-X]
[0-9A-HJ-NP-X][0-9A-HJ-NP-X]",
   "Description": "The device serial number (DSN) of the AWS IoT Button. This
 can be found on the back of the button. The DSN must match the pattern of
 'G030JF05[0-9][0-5][0-9][1-7][0-9A-HJ-NP-X][0-9A-HJ-NP-X][0-9A-HJ-NP-X][0-9A-HJ-
NP-X]'."
 },
 "CertificateARN": {
   "Type": "String",
   "Description": "The Amazon Resource Name (ARN) of the existing AWS IoT
 certificate."
 },
 "SNSTopicName": {
   "Type": "String",
   "Default": "aws-iot-button-sns-topic",
   "Description": "The name of the Amazon SNS topic for AWS CloudFormation to
 create."
 },
 "SNSTopicRoleName": {
   "Type": "String",
   "Default": "aws-iot-button-sns-topic-role",
   "Description": "The name of the IAM role for AWS CloudFormation to create. This
 IAM role allows AWS IoT to send notifications to the Amazon SNS topic."
 },
 "EmailAddress": {
   "Type": "String",
   "Description": "The email address for the Amazon SNS topic to send information
 to."
 }
  },
  "Resources": {
    "IoTThing": {
      "Type": "AWS::IoT::Thing",
```

```
      "Properties": {
        "ThingName": {
    "Fn::Join" : [ "",
      [
        "iotbutton_",
              { "Ref": "IoTButtonDSN" }
   ]
    ]
  }
      }
    },
"IoTPolicy": {
      "Type" : "AWS::IoT::Policy",
      "Properties": {
        "PolicyDocument": {
    "Version": "2012-10-17",
          "Statement": [
            {
              "Action": "iot:Publish",
              "Effect": "Allow",
              "Resource": {
       "Fn::Join": [ "",
            [
              "arn:aws:iot:",
     { "Ref": "AWS::Region" },
     ":",
     { "Ref": "AWS::AccountId" },
     ":topic/iotbutton/",
     { "Ref": "IoTButtonDSN" }
         ]
          ]
     }
            }
          ]
        }
      }
    },
"IoTPolicyPrincipalAttachment": {
      "Type": "AWS::IoT::PolicyPrincipalAttachment",
      "Properties": {
        "PolicyName": {
    "Ref": "IoTPolicy"
  },
  "Principal": {
    "Ref": "CertificateARN"
  }
      }
    },
"IoTThingPrincipalAttachment": {
      "Type" : "AWS::IoT::ThingPrincipalAttachment",
      "Properties": {
        "Principal": {
    "Ref": "CertificateARN"
  },
  "ThingName": {
    "Ref": "IoTThing"
  }
      }
    },
"SNSTopic": {
      "Type": "AWS::SNS::Topic",
   "Properties": {
     "DisplayName": "AWS IoT Button Press Notification",
     "Subscription": [
    {
      "Endpoint": {
```

```
    "Ref": "EmailAddress"
  },
  "Protocol": "email"
   }
 ],
       "TopicName": {
   "Ref": "SNSTopicName"
 }
    }
   },
"SNSTopicRole": {
  "Type": "AWS::IAM::Role",
  "Properties": {
    "AssumeRolePolicyDocument": {
   "Version": "2012-10-17",
         "Statement": [
           {
             "Effect": "Allow",
             "Principal": {
               "Service": "iot.amazonaws.com"
             },
             "Action": "sts:AssumeRole"
           }
         ]
 },
    "Path": "/",
 "Policies": [
   {
     "PolicyDocument": {
    "Version": "2012-10-17",
             "Statement": [
      {
                 "Effect": "Allow",
                 "Action": "sns:Publish",
                 "Resource": {
       "Fn::Join": [ "",
               [
                 "arn:aws:sns:",
        { "Ref": "AWS::Region" },
        ":",
        { "Ref": "AWS::AccountId" },
        ":",
        { "Ref": "SNSTopicName" }
                     ]
          ]
           }
   }
             ]
  },
  "PolicyName": {
    "Ref": "SNSTopicRoleName"
  }
   }
 ]
  }
},
"IoTTopicRule": {
     "Type": "AWS::IoT::TopicRule",
     "Properties": {
       "RuleName": {
   "Fn::Join": [ "",
     [
      "iotbutton_",
      { "Ref": "IoTButtonDSN" }
  ]
   ]
```

```
  },
  "TopicRulePayload": {
    "Actions": [
      {
        "Sns": {
       "RoleArn": {
         "Fn::GetAtt": [ "SNSTopicRole", "Arn" ]
       },
       "TargetArn": {
         "Ref": "SNSTopic"
       }
      }
    }
     ],
     "AwsIotSqlVersion": "2015-10-08",
           "RuleDisabled": false,
           "Sql": {
       "Fn::Join": [ "",
         [
           "SELECT * FROM 'iotbutton/",
     { "Ref": "IoTButtonDSN" },
     "'"
         ]
    ]
     }
         }
       }
     }
  }
}
```

b. Open the AWS CloudFormation console at https://console.aws.amazon.com/cloudformation/.

c. Make sure the AWS region selector displays the region where you created the AWS IoT certificate (for example, US East (N. Virginia)).

d. Choose **Create Stack**.

e. On the **Select Template** page, choose **Upload a template to Amazon S3**, and then choose **Browse**.

f. Select the AWSIoTButtonQuickStart.template file you saved earlier, choose **Open**, and then choose **Next**.

g. On the **Specify Details** page, for **Stack name**, type a name for this AWS CloudFormation stack (for example, MyAWSIoTButtonStack).

h. For **CertificateARN**, type the Amazon Resource Name (ARN) of the AWS IoT certificate (the certificate ARN value) that you noted earlier.

i. For **EmailAddress**, type your email address.

j. For **IoTButtonDSN**, type the device serial number (DSN). You'll find it on the back of your AWS IoT button (for example, G030JF051234A5BC).

k. You can leave **SNSTopicName** and **SNSTopicRoleName** at their defaults, or specify a different Amazon SNS topic name and associated IAM role name. For example, if you plan to set up more AWS IoT buttons, you might want to change these values. Choose **Next**.

l. You do not need to do anything on the **Options** page. Choose **Next**.

m. On the **Review** page, select **I acknowledge that AWS CloudFormation might create IAM resources**, and then choose **Create**.

n. When CREATE_COMPLETE is displayed for MyAWSIoTButtonStack, check your email inbox for a message with a subject line of AWS IoT Button Press Notification. Choose the **Confirm subscription** link in the body of the email message.

3. Using the private key and certificate you created earlier, follow the steps in Configure Your Device to set up your AWS IoT button.

4. After you have set it up, press the button once. A white light should blink several times and then be followed by a steady green light for a few moments. Shortly afterward, you should receive an email message with AWS IoT Button Press Notification in the subject line. You will see information sent by the button in the body of the email message.
5. After you are finished experimenting, you can clean up the AWS resources created by the AWS CloudFormation template. To do this, return to the AWS CloudFormation console and delete MyAWSIoTButtonStack. After you delete MyAWSIoTButtonStack, delete the AWS IoT certificate as follows:

   a. Return to the AWS IoT console.
   b. In the list of resources, select the check box inside of the box that represents the AWS IoT certificate (the box with the handshake icon).
   c. For **Actions**, choose **Deactivate**, and then confirm.
   d. With the box that represents the AWS IoT certificate still selected, for **Actions**, choose **Delete**, and then confirm.
   e. The private key and certificate that you downloaded earlier will no longer be valid, so you can now delete them from your computer.

Next Steps

To learn more about the Lambda blueprint used to set up your button, see Getting Started with AWS IoT. To learn how to use AWS CloudFormation with the AWS IoT button, see http://docs.aws.amazon.com/iot/latest/developerguide/iot-button-cloud-formation.html

Getting Started with AWS IoT

This tutorial shows you how to create resources required to send, receive, and process MQTT messages from devices using AWS IoT.

You will need the following to complete this tutorial:

- A computer with Wi-Fi access.
- If you have an AWS IoT button (pictured here), you can use it to complete this tutorial.
- If you do not have a button, you can purchase one here or you can use the MQTT client in the AWS IoT console to emulate a device.

For more information about AWS IoT, see What Is AWS IoT (p. 1).

Topics

Sign in to the AWS IoT Console

If you do not have an AWS account, create one.

To create an AWS account:

1. Open the AWS home page and choose **Create an AWS Account**.
2. Follow the online instructions. Part of the sign-up procedure involves receiving a phone call and entering a PIN using your phone's keypad.
3. Sign in to the AWS Management Console and open the AWS IoT console.
4. On the **Welcome** page, choose **Get started**.

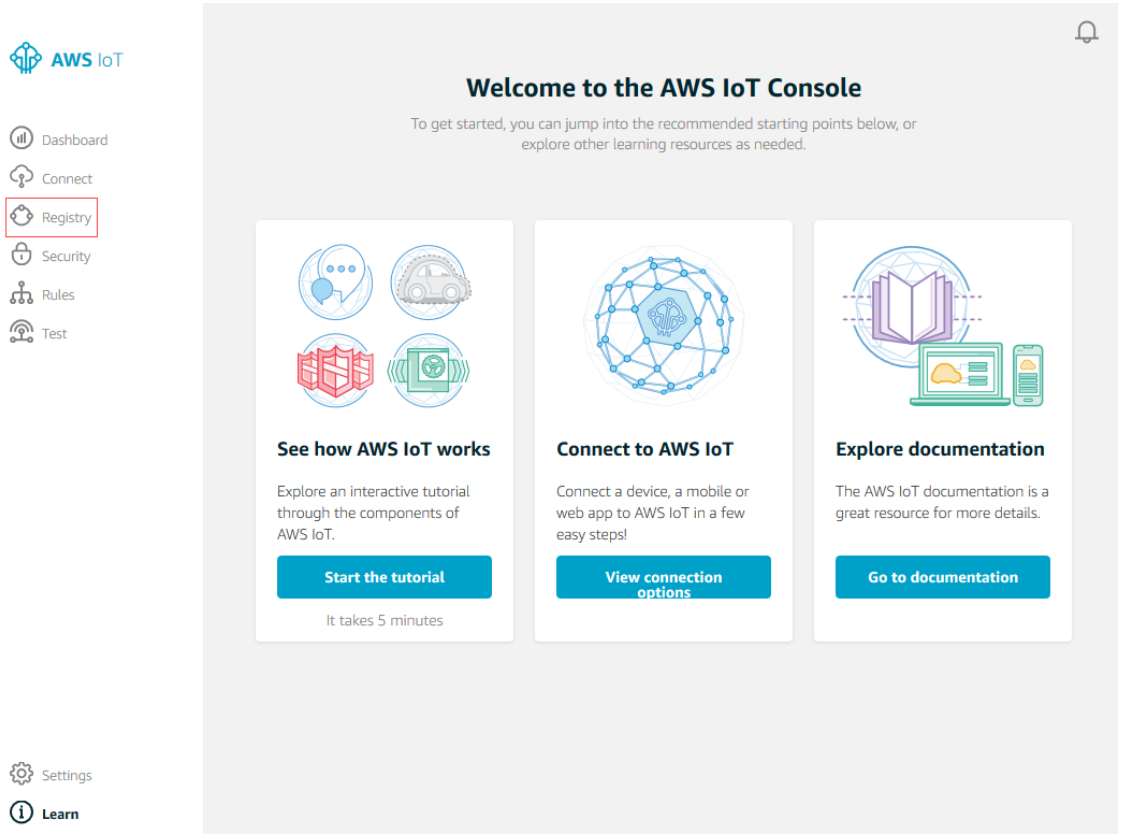
If this is your first time using the AWS IoT console, you will see the **Welcome to the AWS IoT Console** page.

Register a Device in the Thing Registry

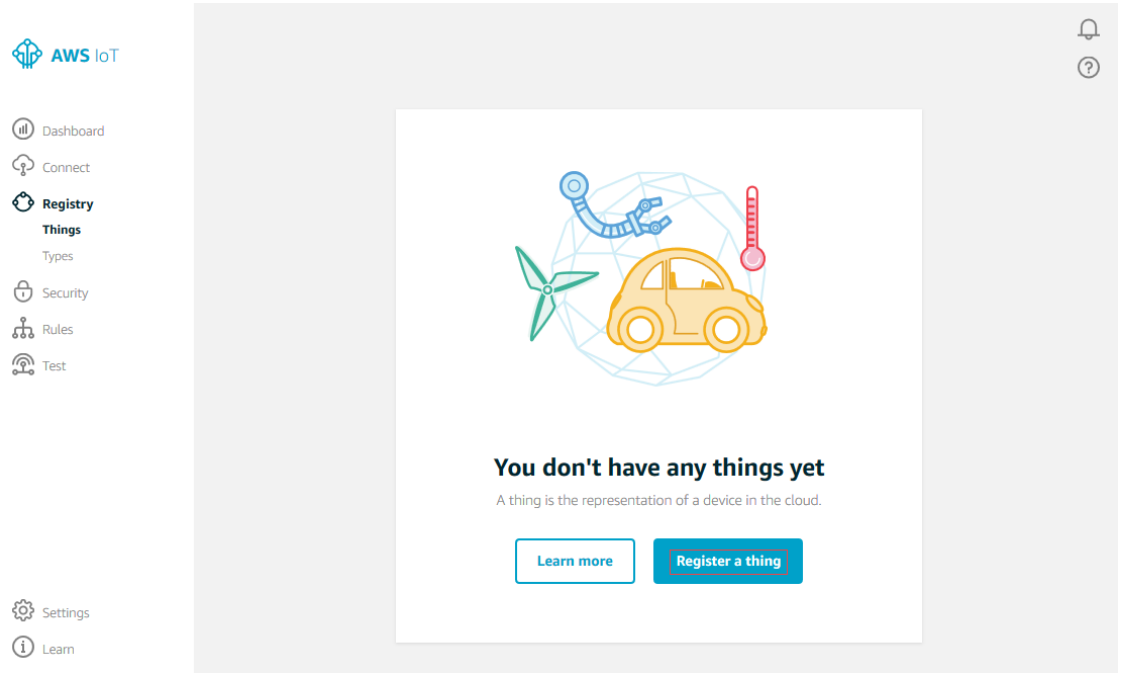
Devices connected to AWS IoT are represented by things in the thing registry. The thing registry allows you to keep a record of all of the devices that are connected to your AWS IoT account.

To register your device in the thing registry:

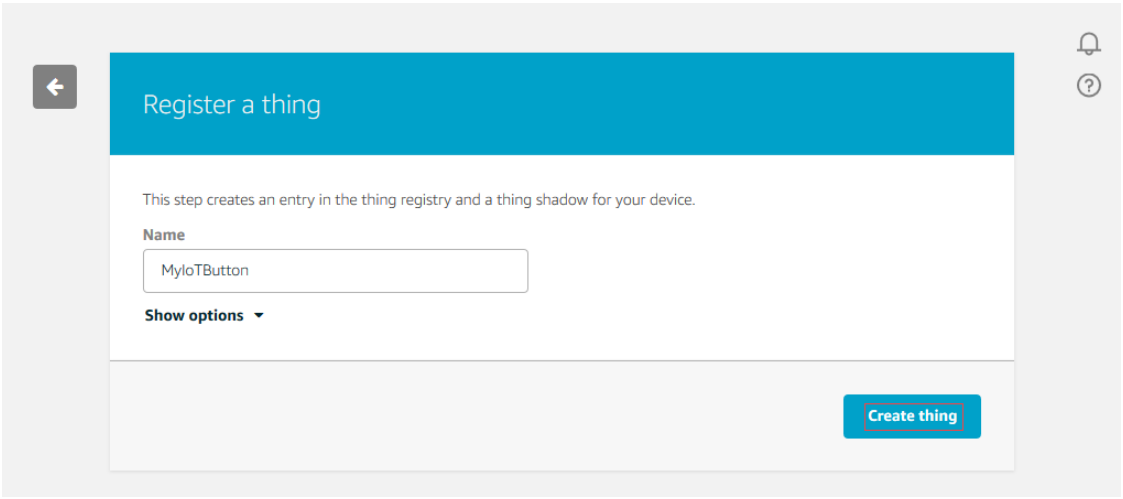
1. On the **Welcome to the AWS IoT Console** page, in the left navigation pane, choose **Registry** to expand the choices, and then choose **Things**.



2. On the page that says **You don't have any things yet**, choose **Register a thing**.



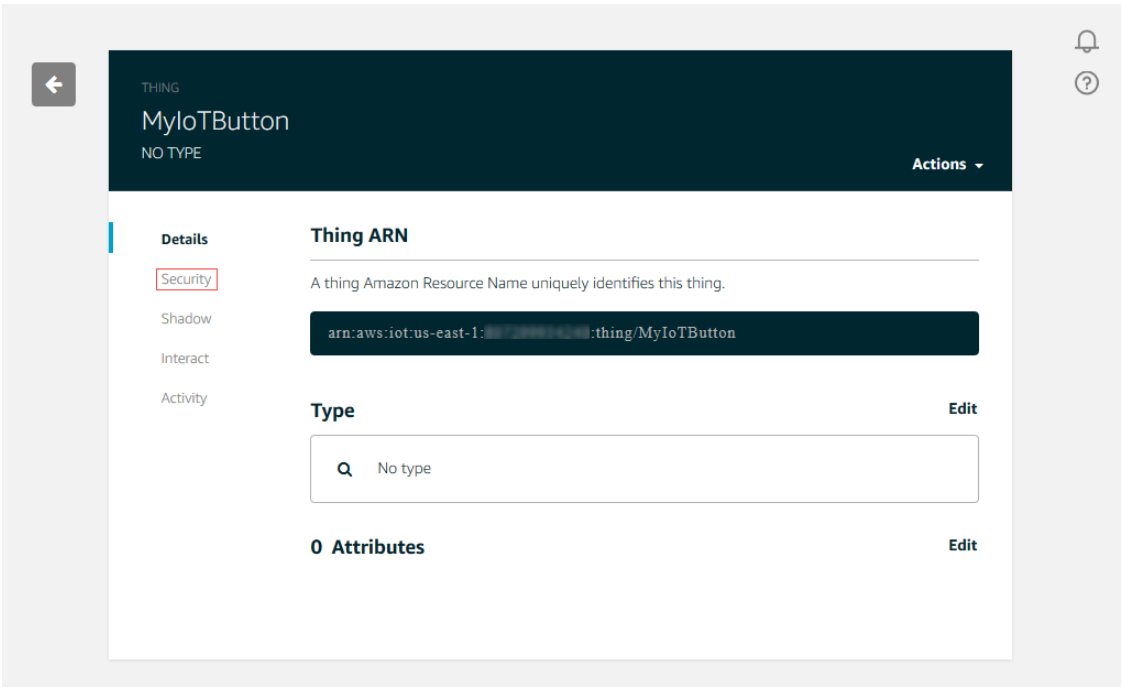
3. On the **Register a thing** page, in the **Name** field, type a name for your device. Choose **Create thing** to add your device to the thing registry.



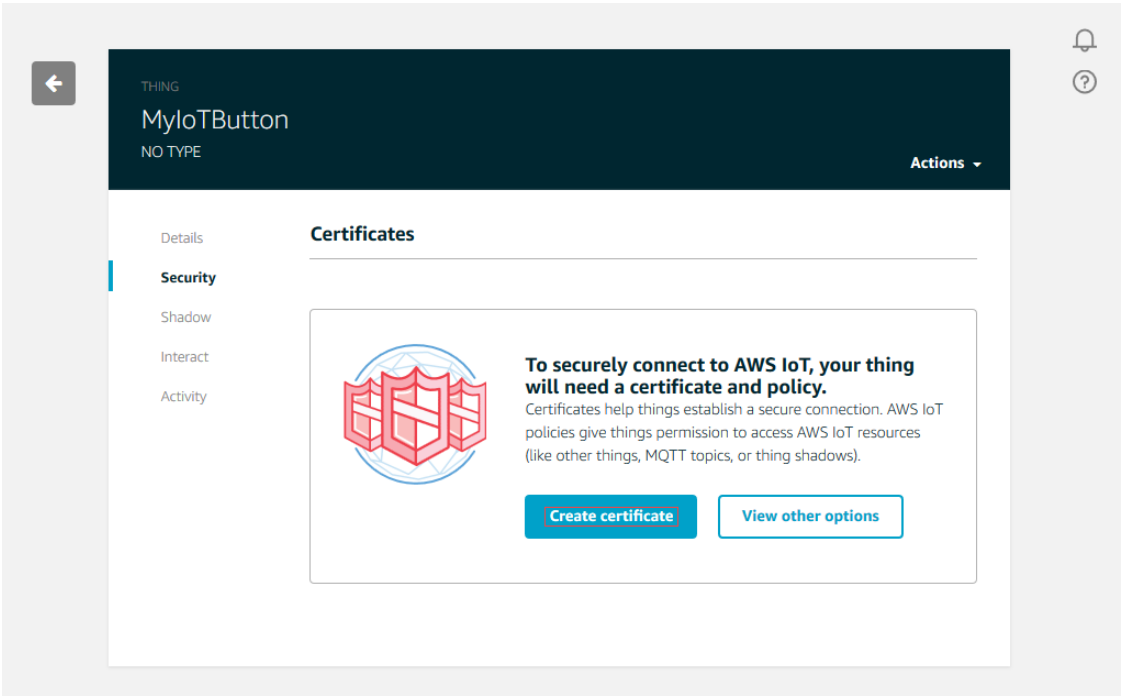
Create and Activate a Device Certificate

Communication between your device and AWS IoT is protected through the use of X.509 certificates. AWS IoT can generate a certificate for you or you can use your own X.509 certificate. This tutorial assumes that AWS IoT will generate the X.509 certificate for you. Certificates must be activated prior to use.

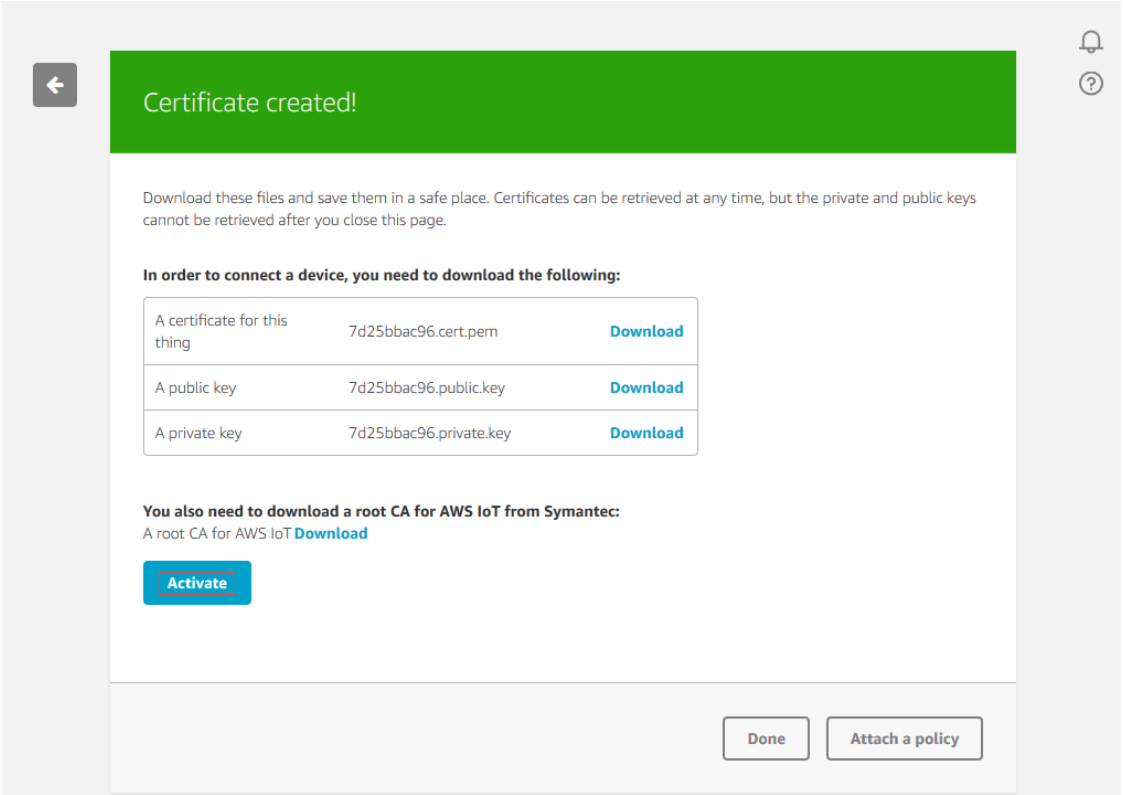
1. On the **Details** page, in the left navigation pane, choose **Security**.



2. On the **Certificates** page, choose **Create certificate**.



3. On the **Certificate created** page, choose **Download** for the certificate, private key, and the root CA for AWS IoT, save each of them to your computer, and then choose **Activate** to continue.



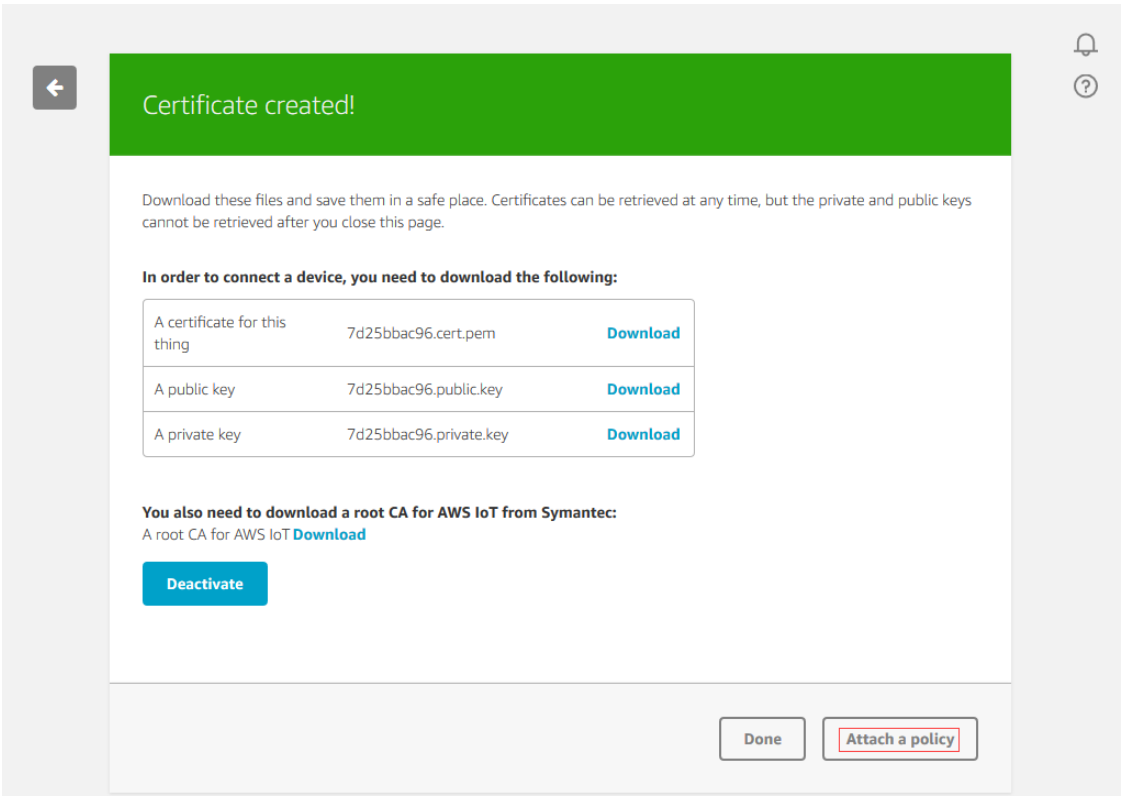
Note
Although it is unlikely root CA certificates are subject to expiration and/or revocation. If this should occur, you will need to copy new a root CA certificate onto your device.

Attach an AWS IoT Policy

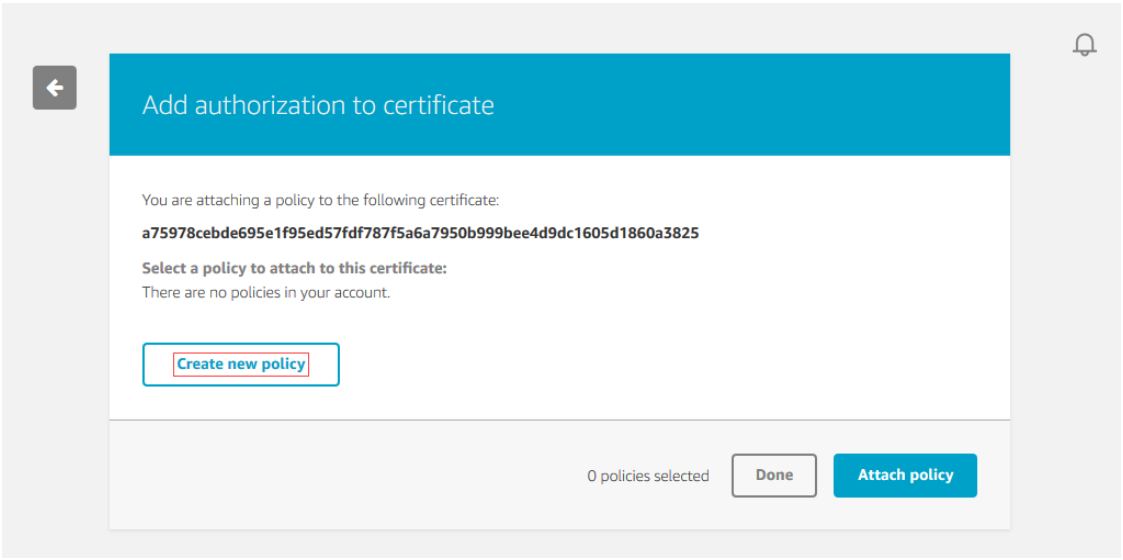
X.509 certificates are used to authenticate your device with AWS IoT. AWS IoT policies are used to authorize your device to perform AWS IoT operations, such as subscribing or publishing to MQTT topics. Your device will present its certificate when sending messages to AWS IoT. To allow your device to perform AWS IoT operations, you must create an AWS IoT policy and attach it to your device certificate.

To create an AWS IoT policy:

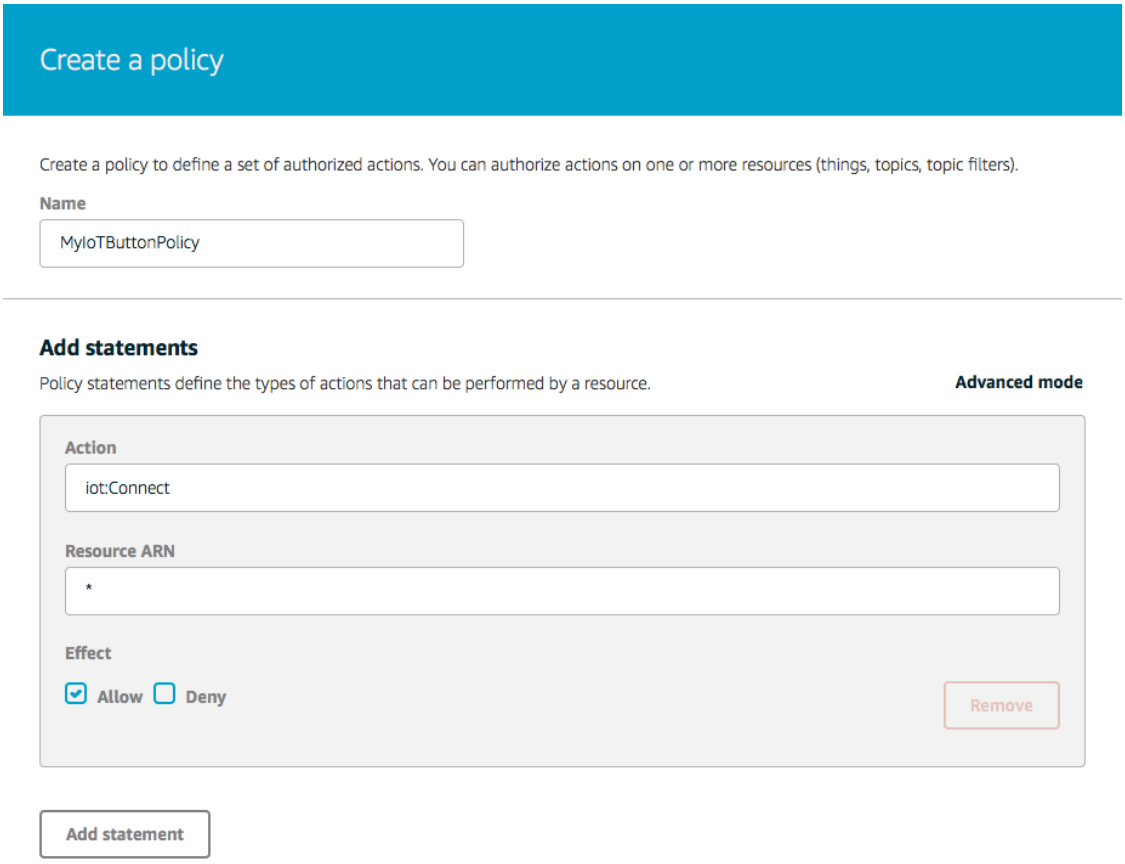
1. On the **Certificate created** page, choose **Attach a policy**.



2. On the **Add authorization to certificate** page, choose **Create new policy**.



3. On the **Create a policy** page, in the **Name** field, type a name for the policy (for example "myPolicy"). In the **Action** field, type **iot:Connect**. In the **Resource ARN** field, type `*`. Select the Allow checkbox. This allows all clients to connect to AWS IoT.



Note
You can restrict which clients (devices) are able to connect by specifying a client ARN as the resource. The client ARNs follow this format:

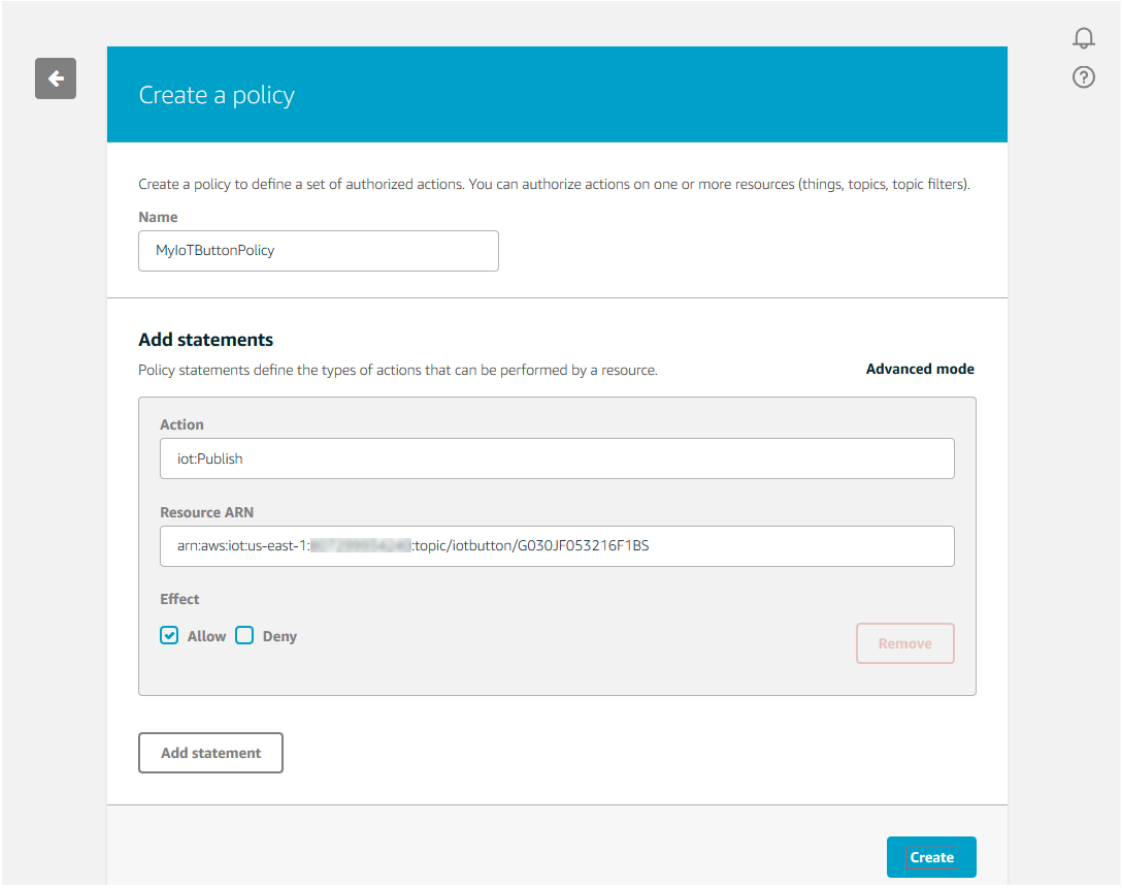
```
arn:aws:iot:your-region:your-aws-account:client/<my-client-id>
```

Select the **Add Statement** button to add another policy statement. In the **Action** field, type **iot:Publish**. In the **Resource ARN** field, type the ARN of the topic to which your device will publish.

> **Note**
> The topic ARN follows this format:
> `arn:aws:iot:your-region:your-aws-account:topic/iotbutton/your-button-serial-number`
> For example:
> `arn:aws:iot:us-east-1:123456789012:topic/iotbutton/G030JF055364XVRB`
> You can find the serial number on the bottom of your button.
> If you are not using an AWS IoT button, place the topic your device publishes to after `topic/` in the ARN. For example:
> `arn:aws:iot:us-east-1:123456789012:topic/my/topic/here`

Finally, select the **Allow** check box. This allows your device to publish messages to the specified topic.

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

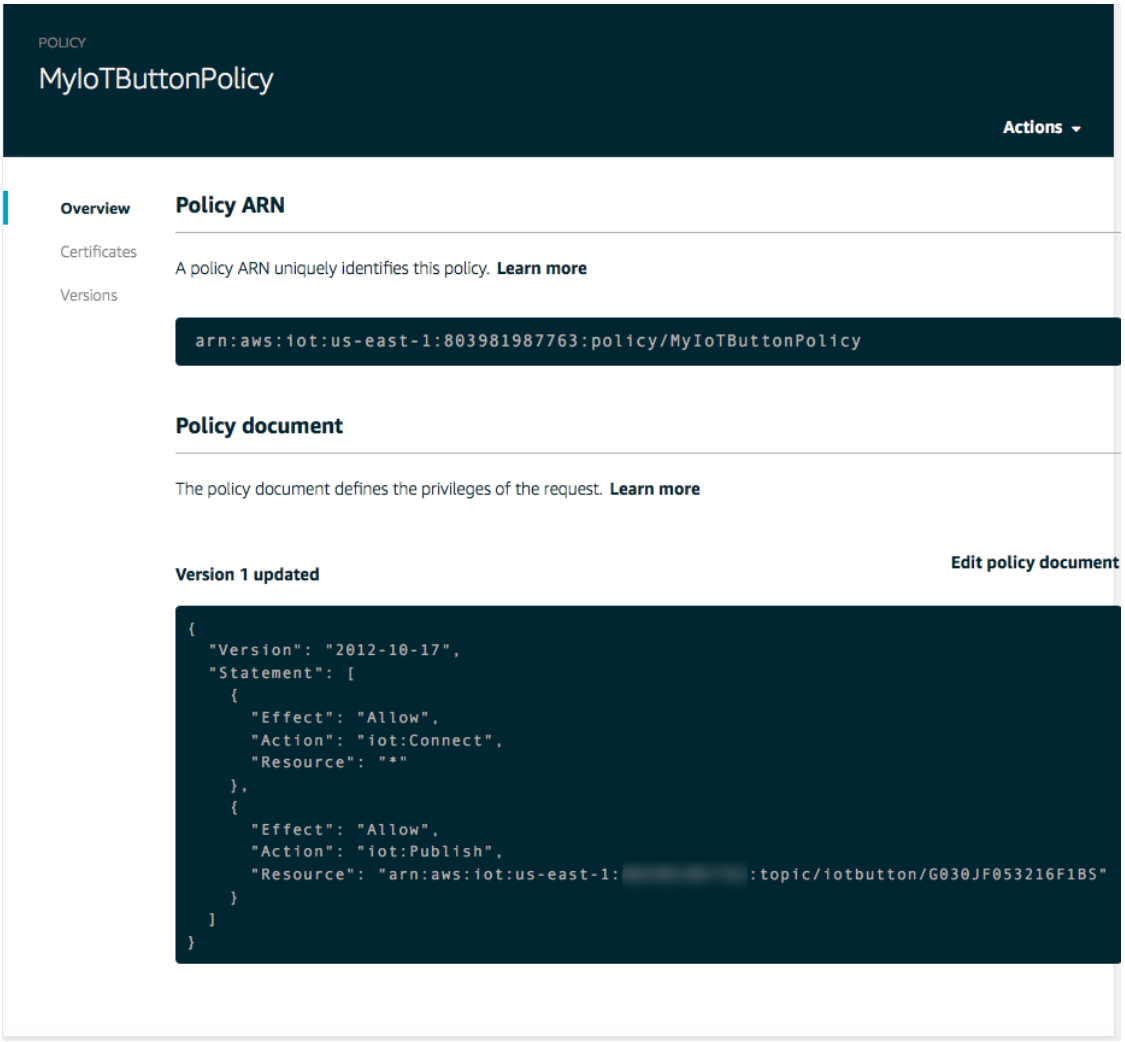


For more information, see Managing AWS IoT Policies.

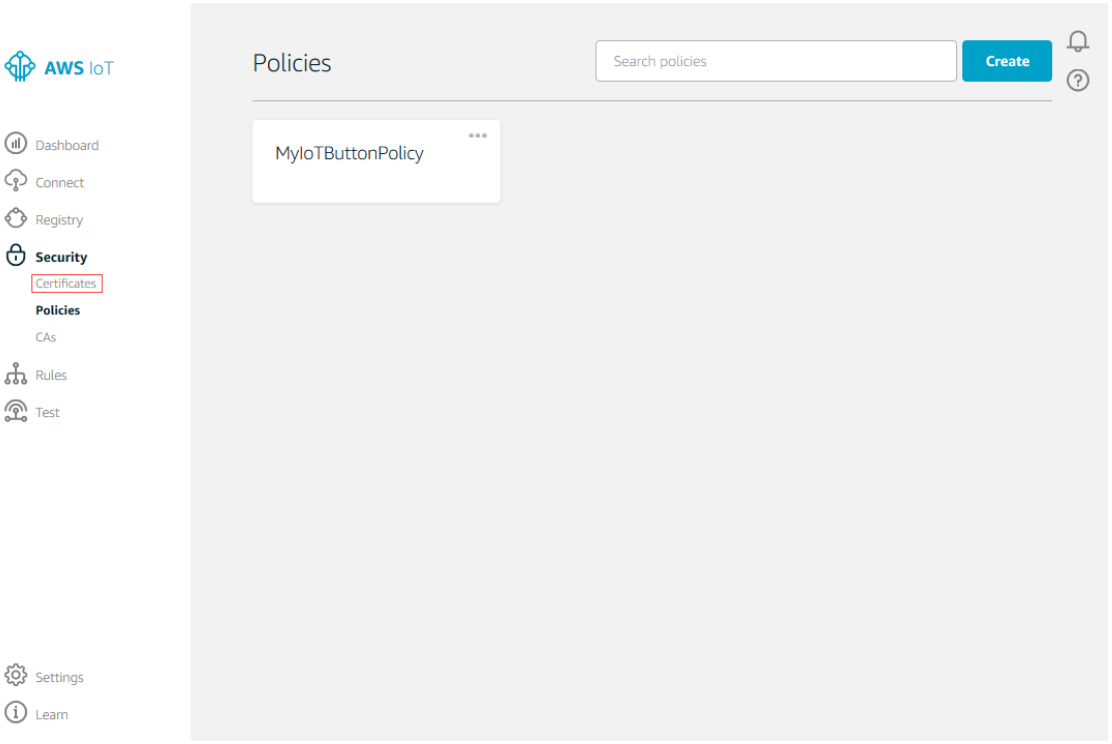
Attach an AWS IoT Policy to a Device Certificate

Now that you have created a policy, you must attach it to your device certificate. Attaching an AWS IoT policy to a certificate gives the device the permissions specified in the policy.

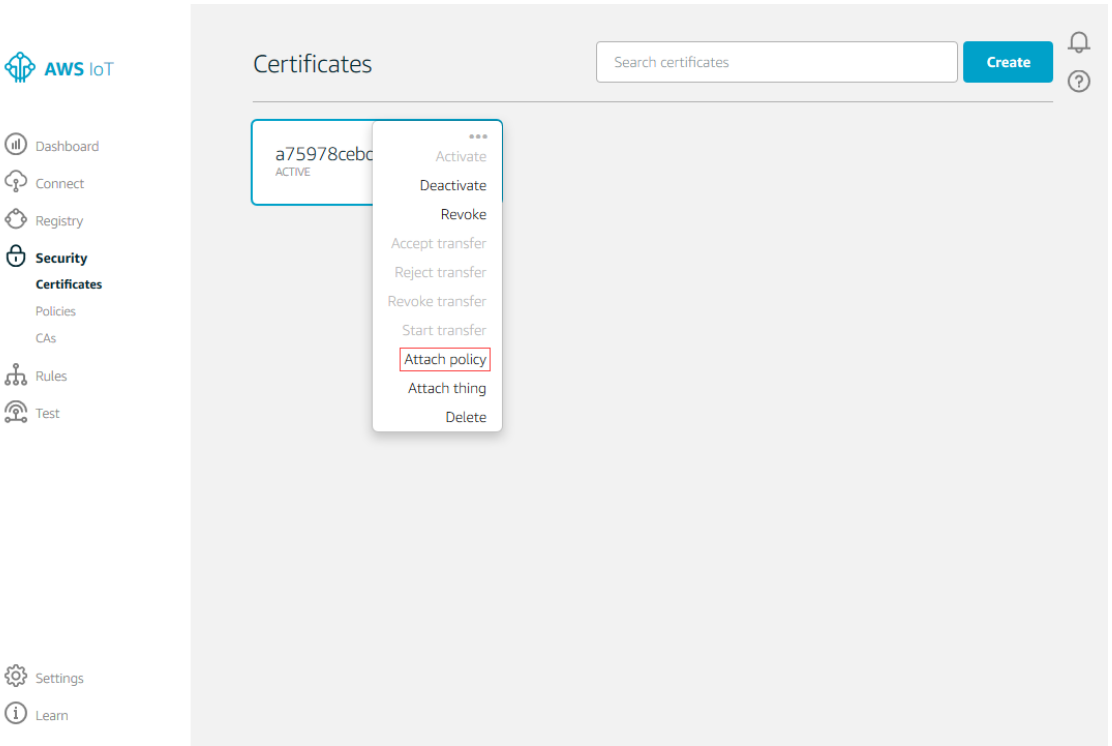
1. On the **Overview** page for the policy, in the left navigation area, choose the left arrow to go to the AWS IoT **Policies** page.



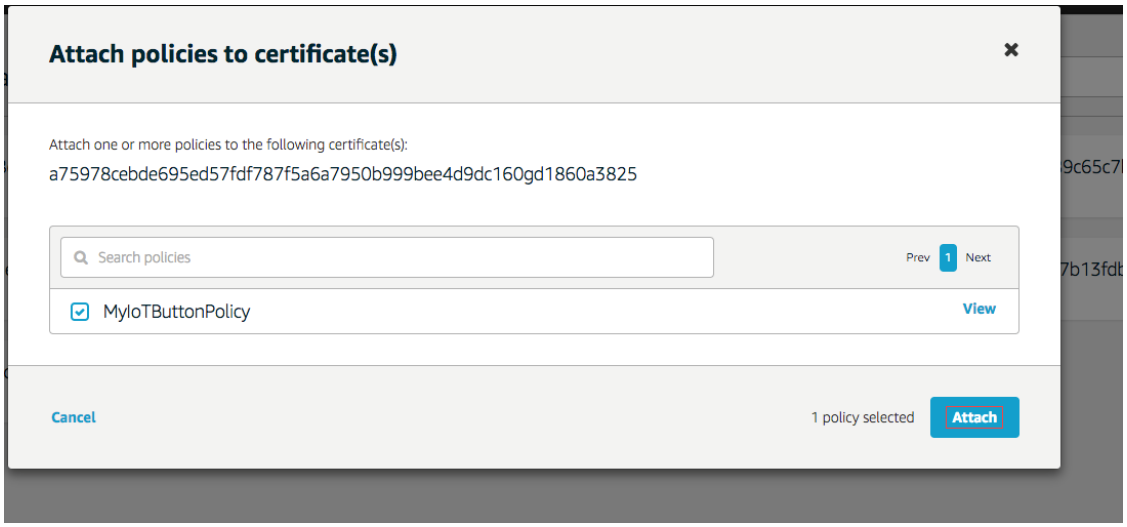
2. On the **Policies** page, in the left navigation pane, under **Security**, choose **Certificates**.



3. In the box for the certificate you created, choose **...** to open a drop-down menu, and then choose **Attach policy**.



4. In the **Attach policies to certificate(s)** dialog box, select the check box next to the policy you created in the previous step, and then choose **Attach**.

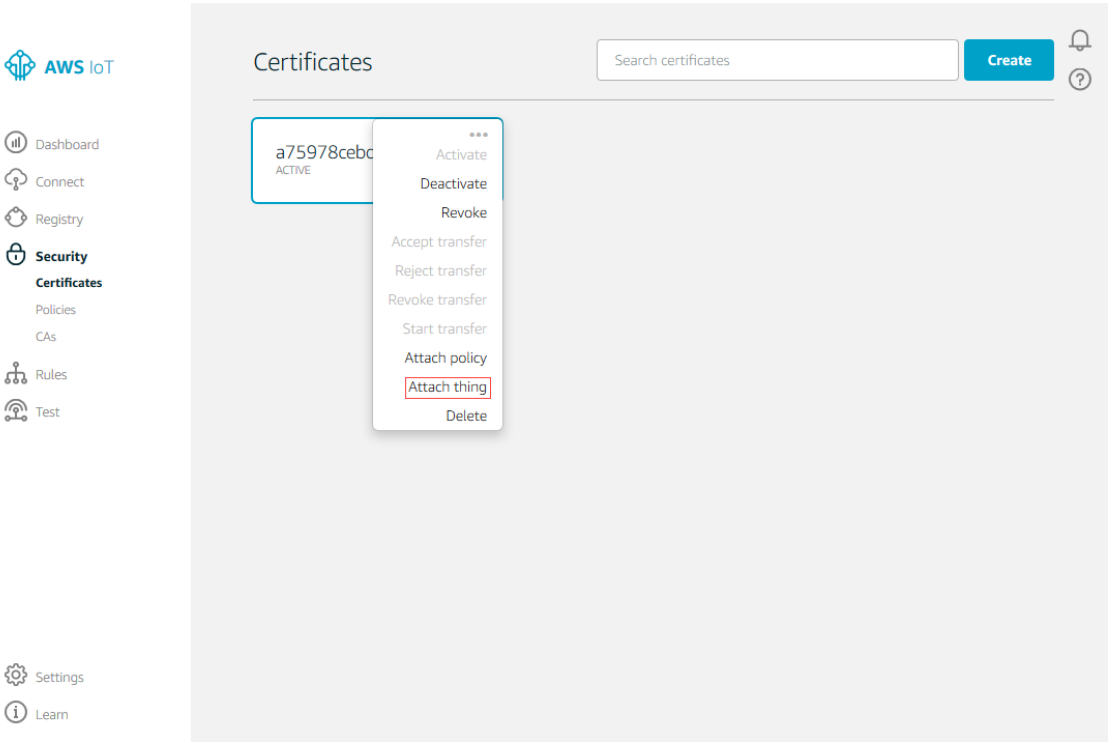


Attach a Certificate to a Thing

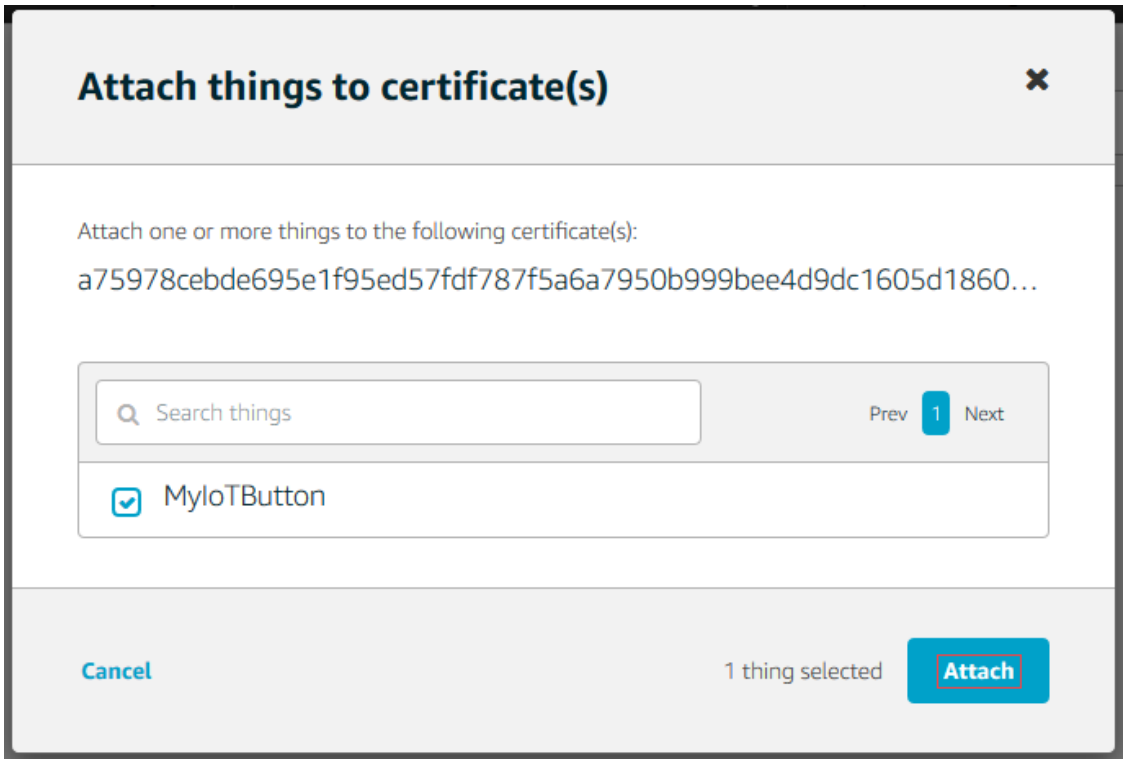
A device must have a certificate, private key and root CA certificate to authenticate with AWS IoT. We recommend that you also attach the device certificate to the thing that represents your device in AWS IoT. This allows you to create AWS IoT policies that grant permissions based on certificates attached to your things. For more information. see Thing Policy Variables (p. 109)

To attach a certificate to the thing representing your device in the thing registry:

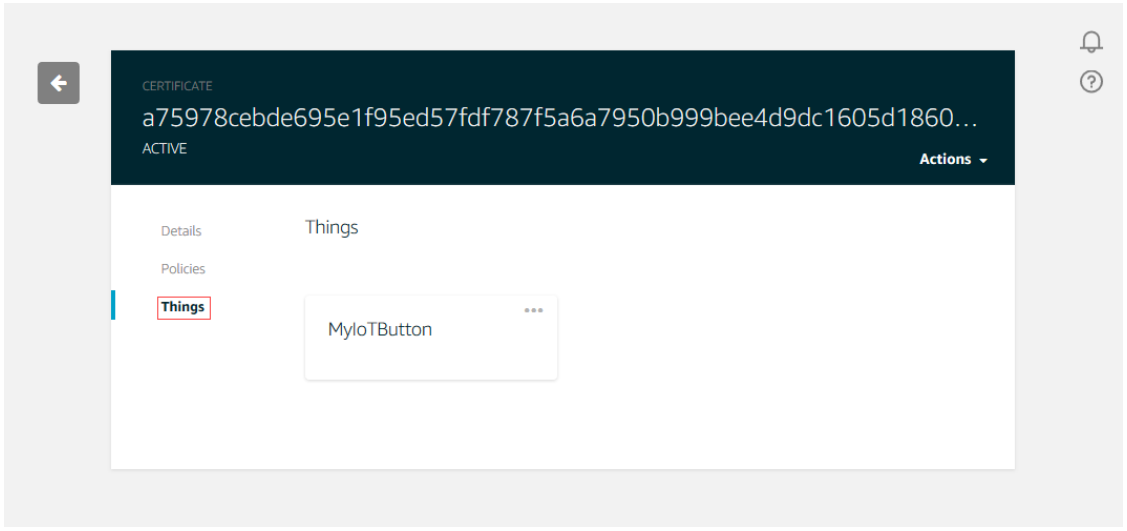
1. In the box for the certificate you created, choose **...** to open a drop-down menu, and then choose **Attach thing**.



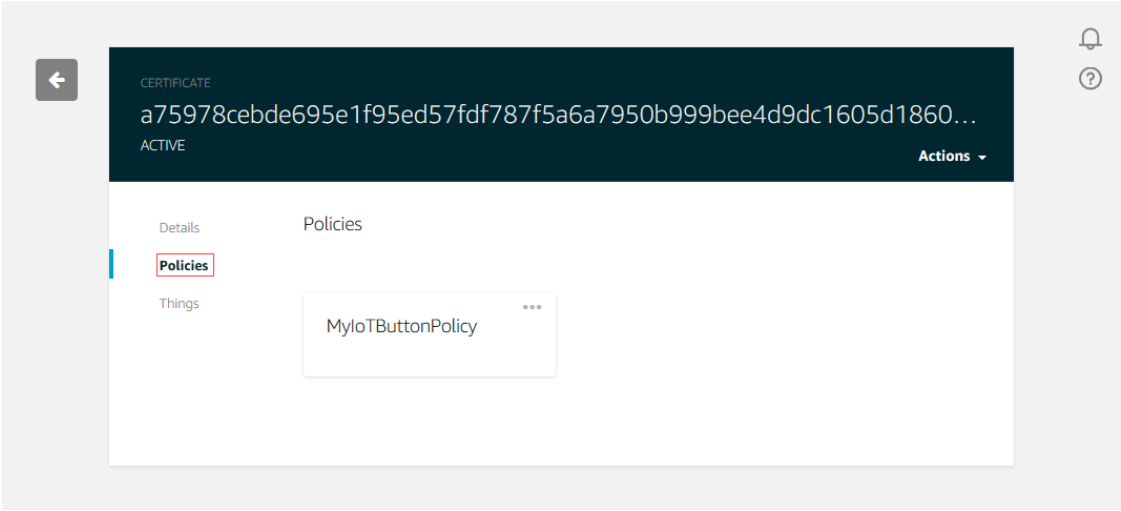
2. In the **Attach things to certificate(s)** dialog box, select the check box next to the thing you registered, and then choose **Attach**.



3. To verify the thing is attached, select the box representing the certificate. On the **Details** page for the certificate, in the left navigation pane, choose **Things**.



4. To verify the policy is attached, on the **Details** page for the certificate, in the left navigation pane, choose **Policies**.



Configure Your Device

Configuring your device allows it to connect to your Wi-Fi network. Your device must be connected to your Wi-Fi network to install required files and send messages to AWS IoT. All devices must install a device certificate, private key and root CA certificate in order to communicate with AWS IoT.

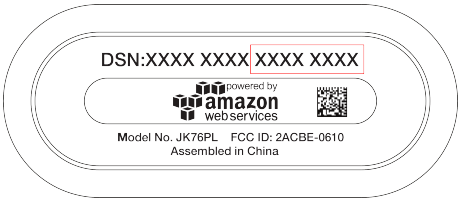
Note
If you have a smartphone, you can use our iOS or Android apps to get started quickly. Install the AWS IoT Button app from the iOS or Google Play app stores. Otherwise, follow the steps here.

Configure an AWS IoT Button

To configure your AWS IoT button:

Turn on your device

1. Remove the AWS IoT button from its packaging, and then press and hold the button until a blue blinking light appears. (This should take no longer than 15 seconds.)
2. The button acts as a Wi-Fi access point, so when your computer searches for Wi-Fi networks, it will find one called **Button ConfigureMe - XXX** where *XXX* is a three-character string generated by the button. Use your computer to connect to the button's Wi-Fi access point.
3. The first time you connect to the button's Wi-Fi access point, you will be prompted for the WPA2-PSK password. Type the last 8 characters of the device serial number (DSN). You'll find the DSN on the back of the device, as shown here:



Copy your device certificate and private key onto your AWS IoT button

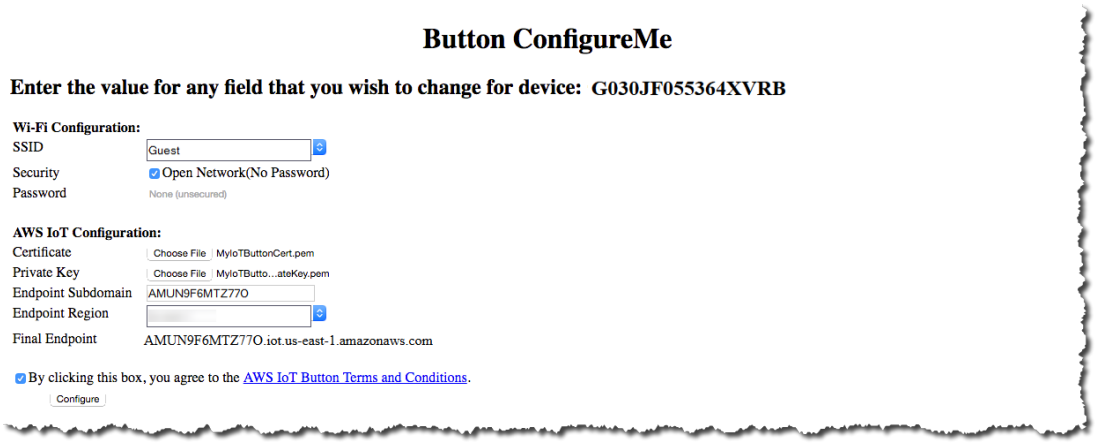
To connect to AWS IoT, you must copy your device certificate onto the AWS IoT button.

1. In a browser, navigate to http://192.168.0.1/index.html.
2. Complete the configuration form.

   - Type your Wi-Fi SSID and password.
   - Browse to and select your certificate and private key.
   - Find your custom endpoint in the AWS IoT console. (From the dashboard, in the left navigation pane, choose **Registry** to expand the selection, and then choose **Things**. Select the box representing your button to show its details page. On the details page, in the left navigation pane, choose **Interact** and look for the **HTTPS** section.) Your endpoint will look something like the following:

     ```
     ABCDEFG1234567.iot.us-east-2.amazonaws.com
     ```

     where `ABCDEFG1234567` is the subdomain and `us-east-2` is the region.
   - On the **Button ConfigureMe** page, type the subdomain, and then choose the region that matches the region in your AWS IoT endpoint.
   - Select the **Terms and Conditions** check box. Your settings should now look like the following:

   - Choose **Configure**. Your button should now connect to your Wi-Fi network.

Configure a Different Device

Consult your device's documentation to connect to it and copy your device certificate, private key and root CA certificate onto your device.

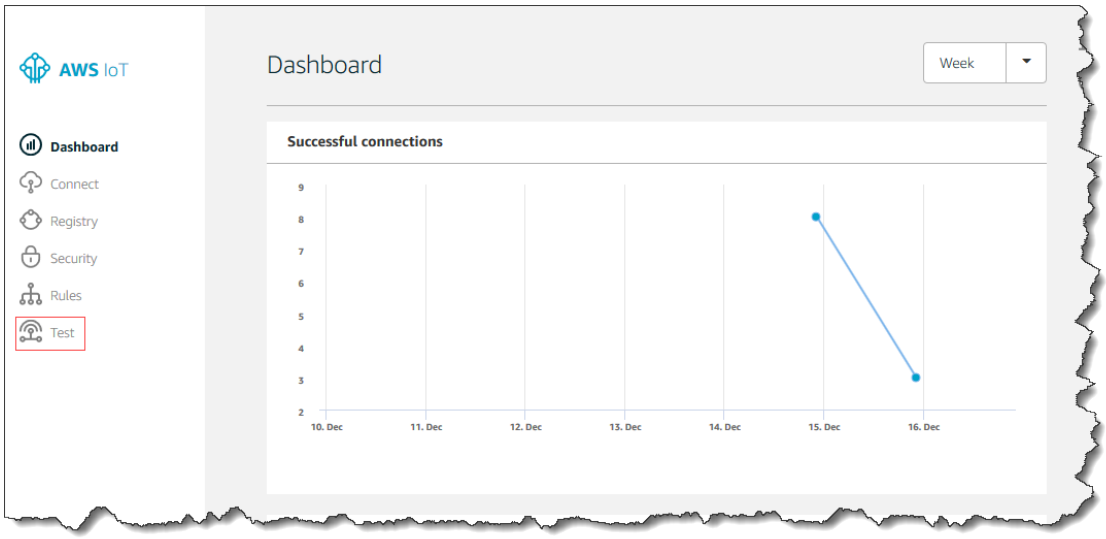
View Device MQTT Messages with the AWS IoT MQTT Client

You can use the AWS IoT MQTT client to better understand the MQTT messages sent by a device.

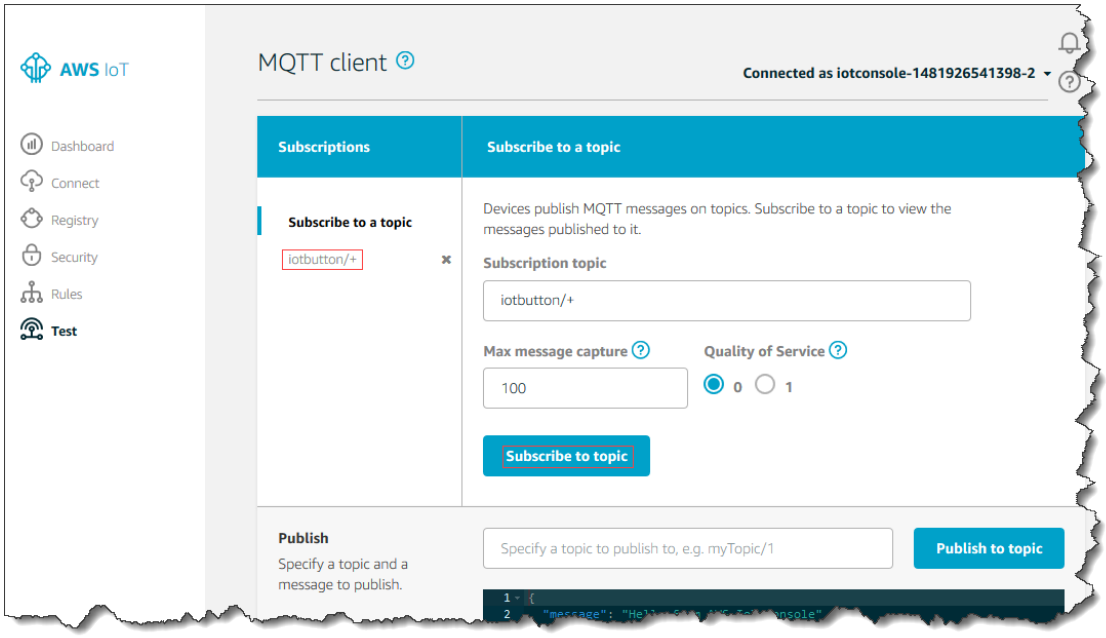
Devices publish MQTT messages on topics. You can use the AWS IoT MQTT client to subscribe to these topics to see these messages.

To view MQTT messages:

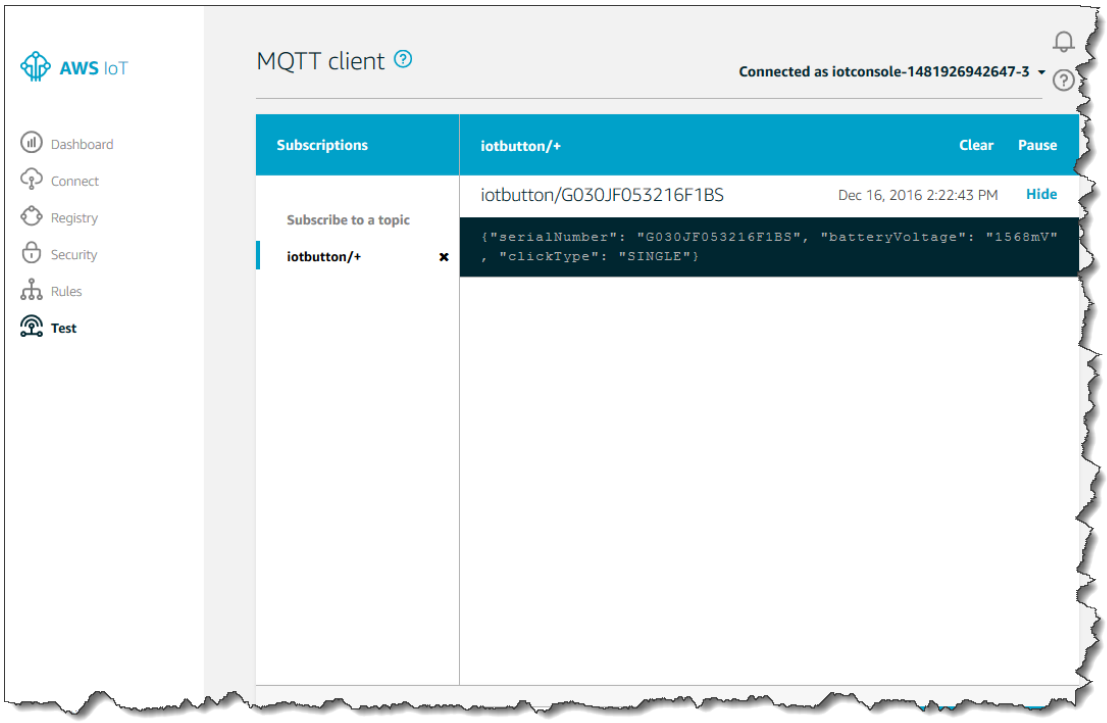
1. In the AWS IoT console, in the left navigation pane, choose **Test**.



2. Subscribe to the topic on which your thing publishes. In the case of the AWS IoT button, you can subscribe to `iotbutton/+`. In **Subscribe to a topic**, in the **Subscription topic** field, type **`iotbutton/+`**, and then choose **Subscribe to topic**. This topic should appear under **Subscriptions**. Choose it there.



3. Press your AWS IoT button, and then view the message in the AWS IoT MQTT client. If you do not have a button, you will simulate a button press in the next step.

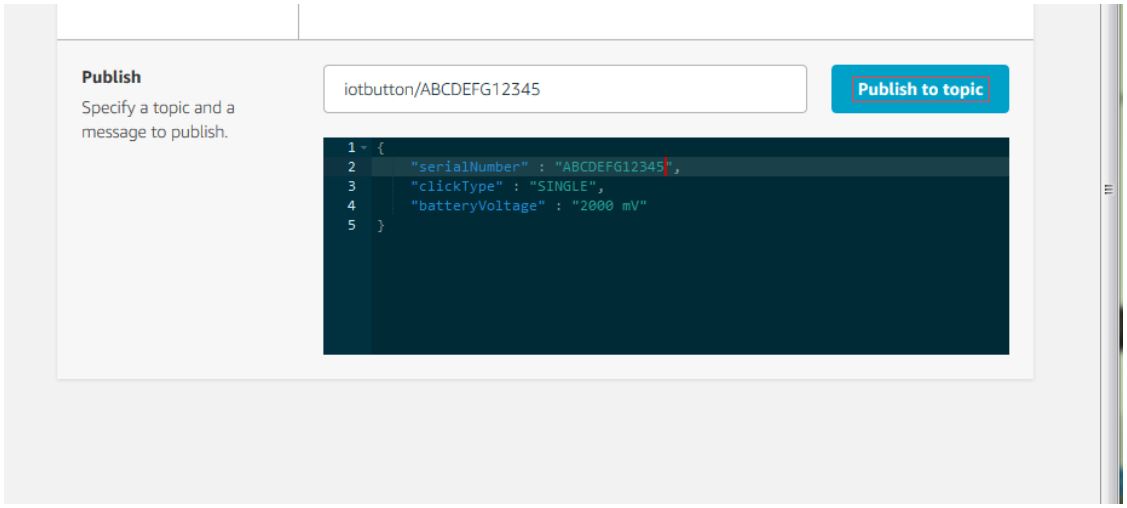


4. To publish a message using the AWS IoT console:

   On the MQTT client page, in the **Publish** section, in **Specify a topic**, type **`iotbutton/ABCDEFG12345`**. In the message payload section, type the following JSON:

```
{
    "serialNumber": "ABCDEFG12345",
    "clickType": "SINGLE",
    "batteryVoltage": "2000 mV"
}
```

   Choose **Publish to topic**. You should see the message in the AWS IoT MQTT client.



Configure and Test Rules

The AWS IoT rules engine listens for incoming MQTT messages that match a rule. When a matching message is received, the rule takes some action with the data in the MQTT message (for example, writing data to an Amazon S3 bucket, invoking a Lambda function, or sending a message to an Amazon SNS topic). In this step, you will create and configure a rule to send the data received from a device to an Amazon SNS topic. Specifically, you will:

- Create an Amazon SNS topic.
- Subscribe to the Amazon SNS topic using a cell phone number.
- Create a rule that will send a message to the Amazon SNS topic when a message is received from your device.
- Test the rule using your AWS IoT button or an MQTT client.

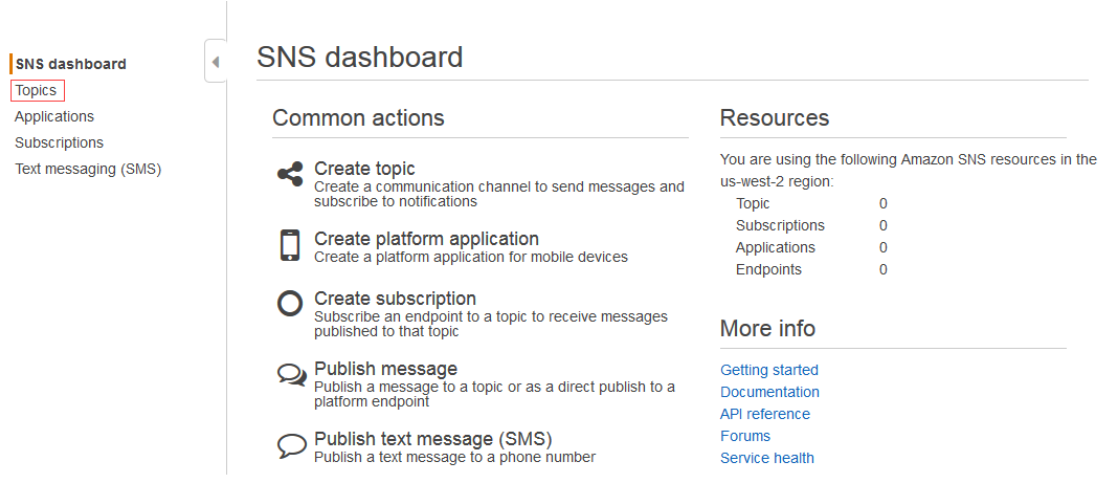
In the upper-right corner of this page, there is a **Filter View** drop-down list. For instructions for testing your rule by using the AWS IoT button, choose **AWS IoT Button**. For instructions for testing your rule by using the AWS IoT MQTT client, choose **MQTT Client**.

Create an SNS Topic

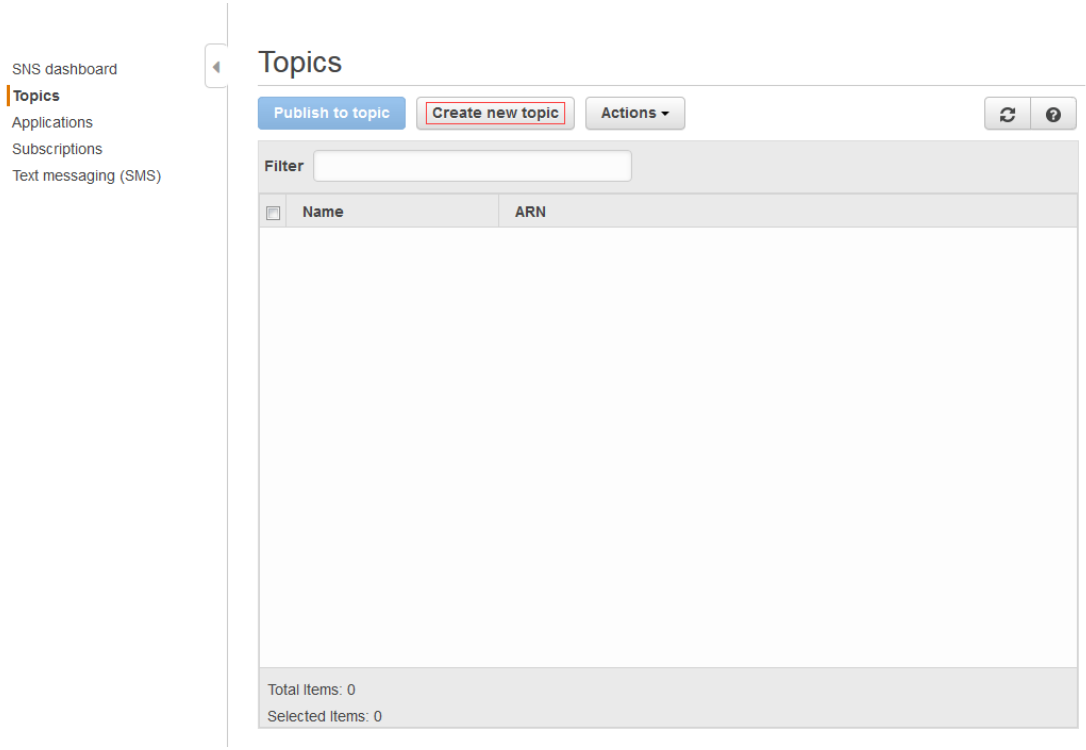
You will use the Amazon SNS console to create an Amazon SNS topic.

> **Note**
> Amazon SNS is not available in all AWS regions.

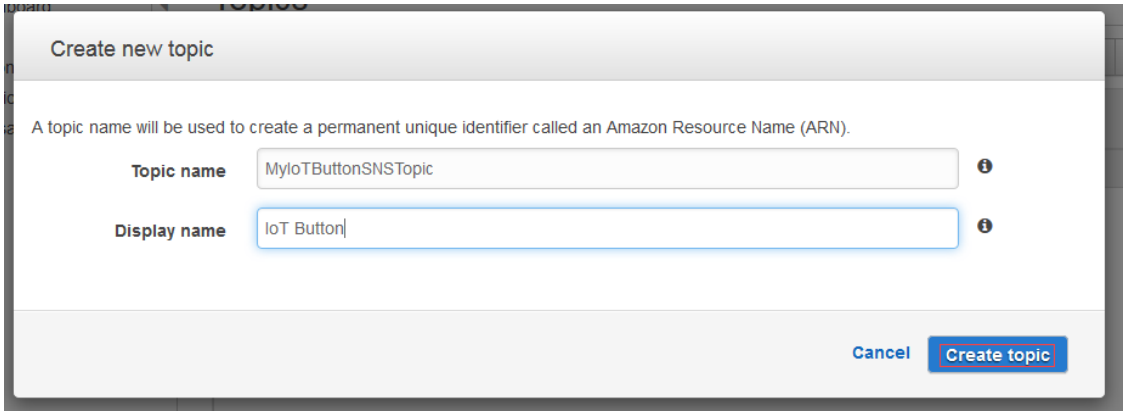
1. Open the Amazon SNS console.
2. On the left pane, choose **Topics**.



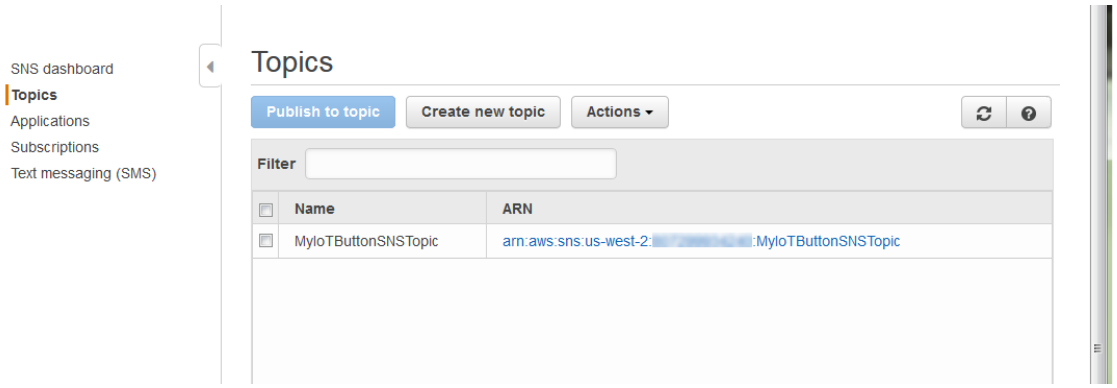
3. Choose **Create new topic**.



4. Type a topic name and a display name, and then choose **Create topic**.



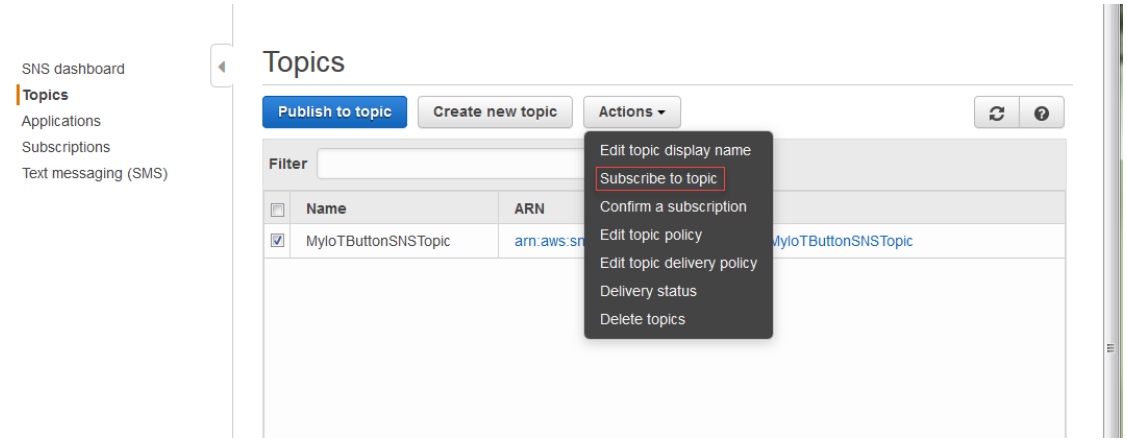
5. Make a note of the ARN for the topic you just created.



Subscribe to an Amazon SNS Topic

To receive SMS messages on your cell phone, you need to subscribe to the Amazon SNS topic.

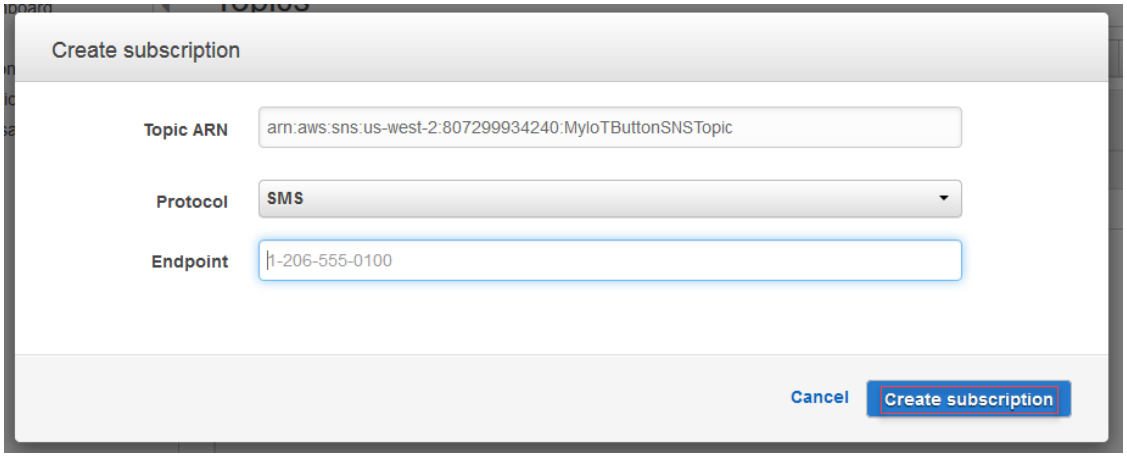
1. In the Amazon SNS console, select the check box next to the topic you just created. From the **Actions** menu, choose **Subscribe to topic**.



2. On **Create subscription**, from the **Protocol** drop-down list, choose **SMS**.

   In the **Endpoint** field, type the phone number of an SMS-enabled cell phone, and then choose **Create subscription**.

   Note
   Enter the phone number using numbers and dashes only.



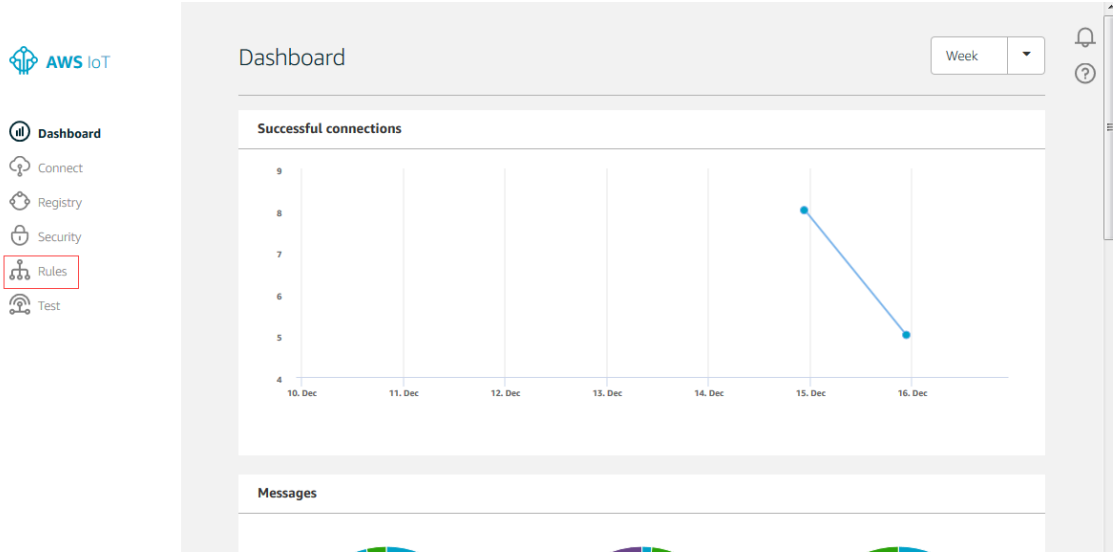
You will receive a text message that confirms you successfully created the subscription.

Create a Rule

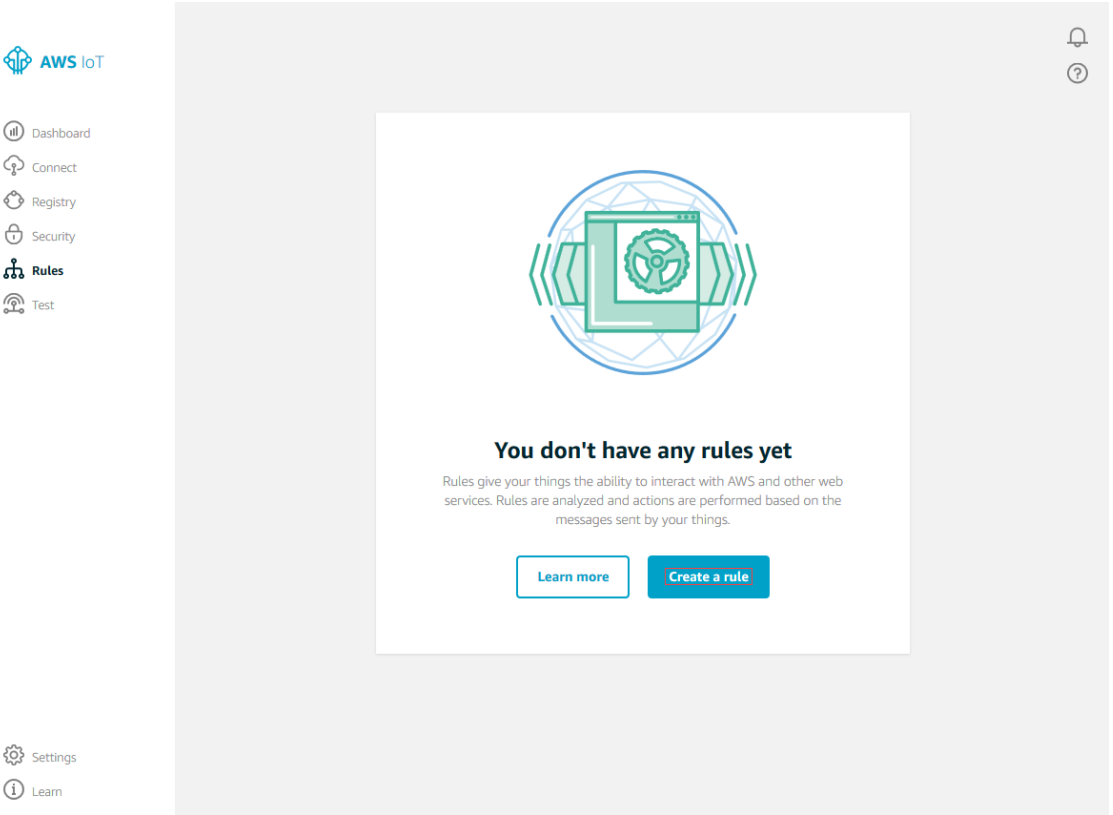
AWS IoT rules consist of a topic filter, a rule action, and, in most cases, an IAM role. Messages published on topics that match the topic filter trigger the rule. The rule action defines which action to take when the rule is triggered. The IAM role contains one or more IAM policies that determine which AWS services the rule can access. You can create multiple rules that listen on a single topic. Likewise, you can create a single rule that is triggered by multiple topics. The AWS IoT rules engine continuously processes messages published on topics that match the topic filters defined in the rules.

In this example, you will create a rule that uses Amazon SNS to send an SMS notification to a cell phone number.

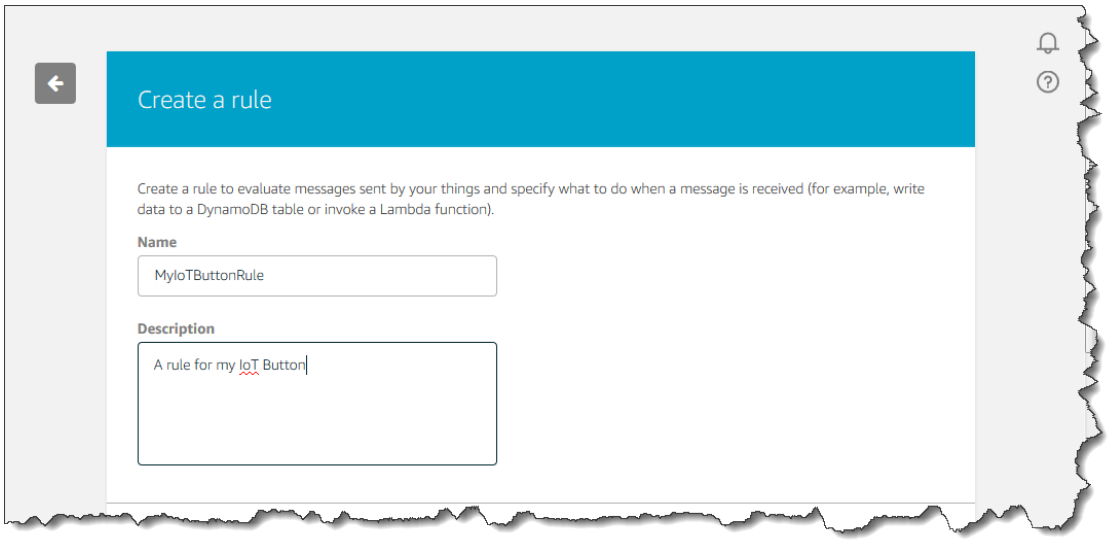
1. In the AWS IoT console, in the left navigation pane, choose **Rules**.



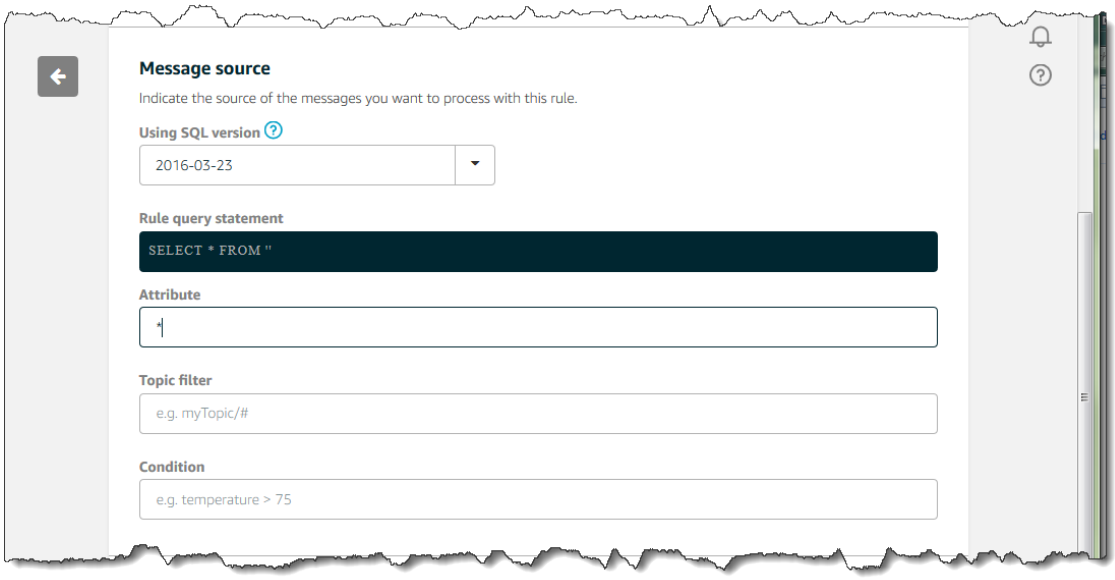
2. On the **Rules** page, choose **Create a rule**.



3. On the **Create a rule** page, in the **Name** field, type a name for your rule. In the **Description** field, type a description for the rule.



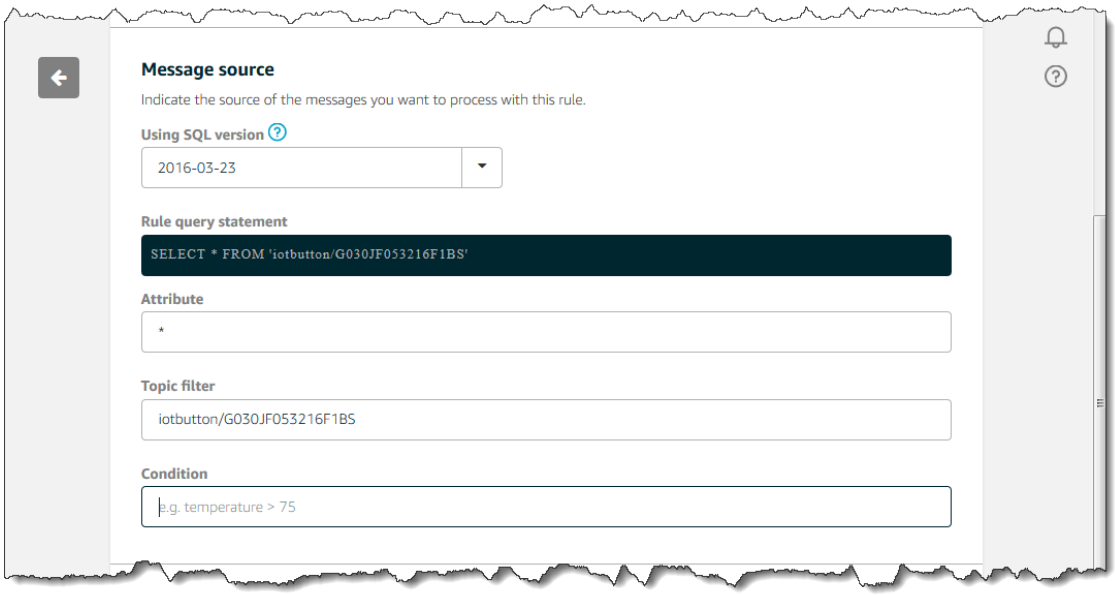
4. Scroll down to **Message source**. Choose the latest version from the **Using SQL version** drop-down list. In the **Attribute** field, type `*`. This specifies that you want to send the entire MQTT message that triggered the rule.



5. The rules engine uses the topic filter to determine which rules to trigger when an MQTT message is received. In the **Topic filter** field, type **`iotbutton/`*`your-button-DSN`***. If you are not using an AWS IoT button, type **`my/topic`** or the topic used in the rule.

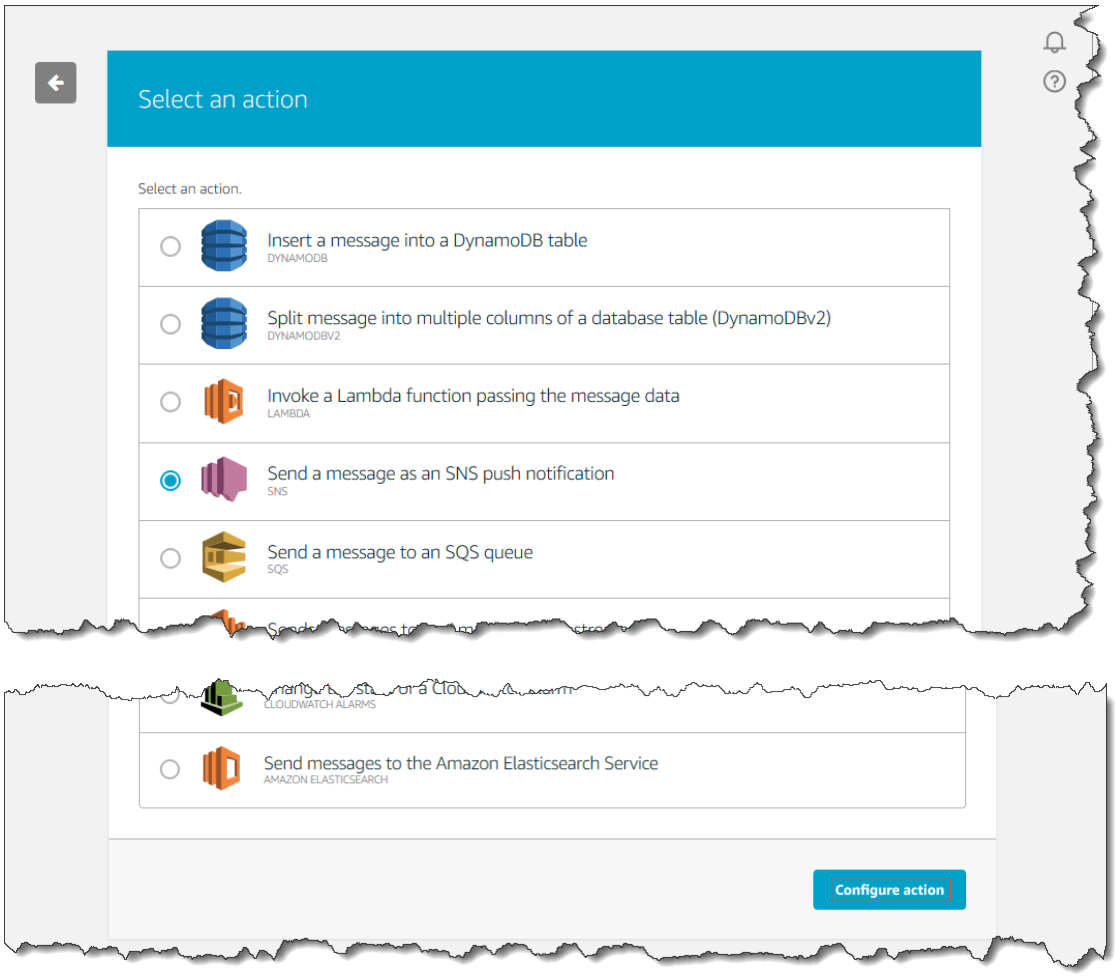
   Note
   You can find the DSN on the bottom of the button.

   Leave **Condition** blank.

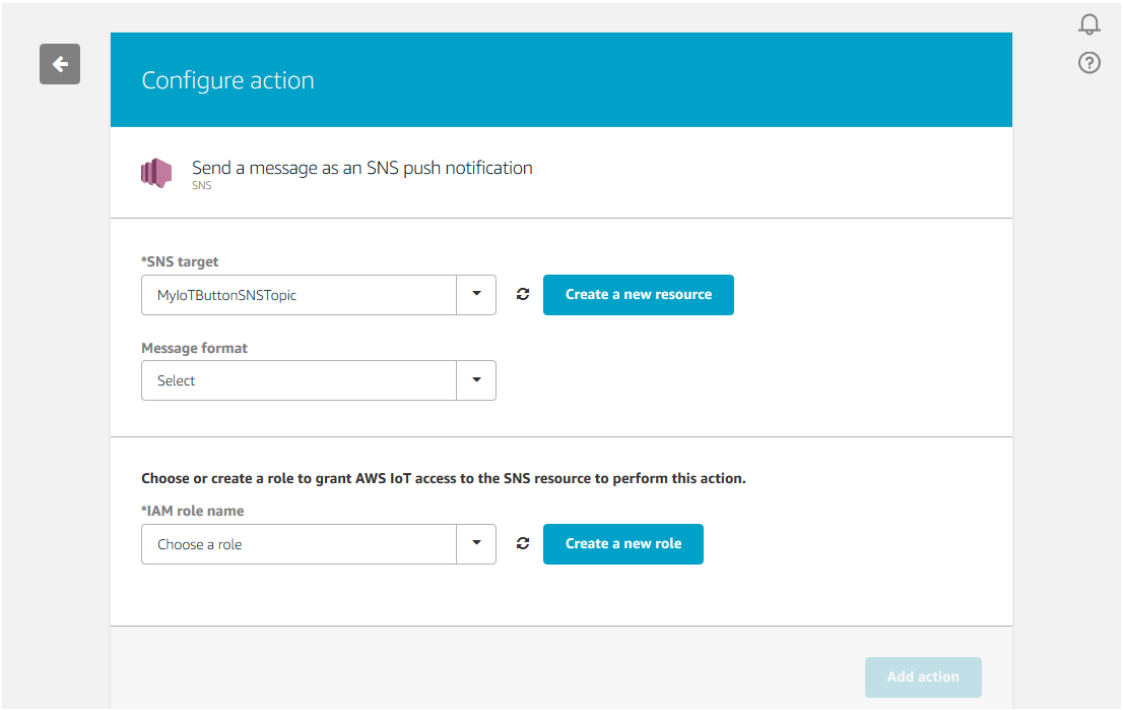


6. In **Set one or more actions**, choose **Add action**.

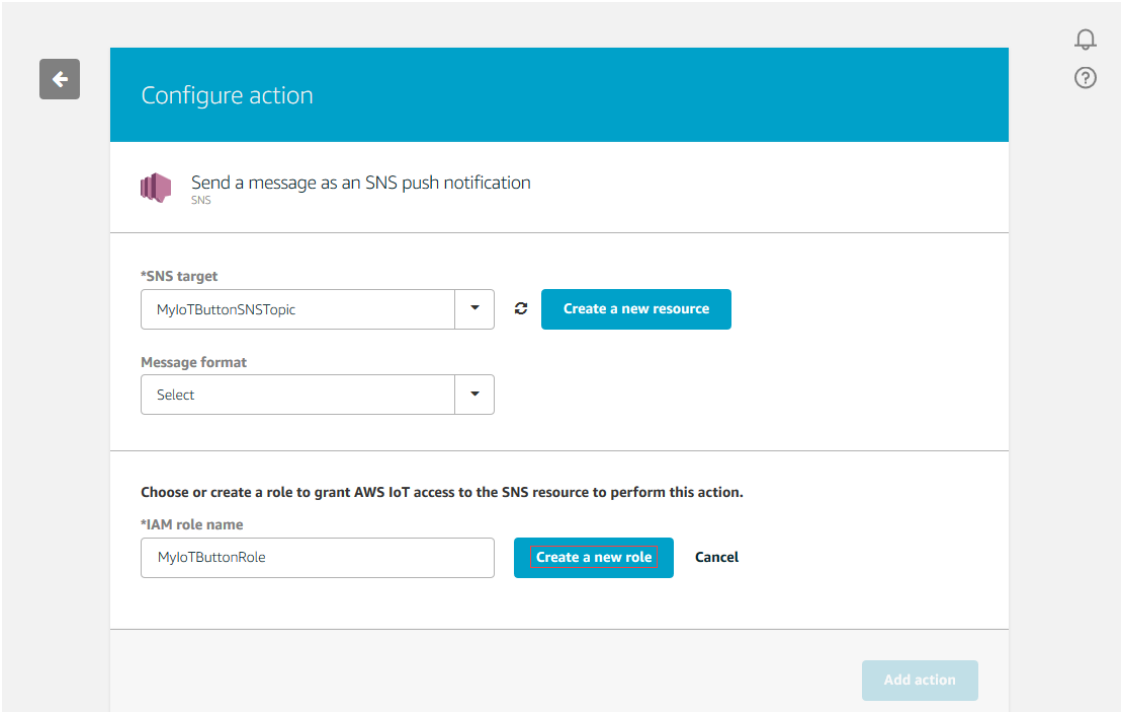
7. On the **Select an action** page, select **Send a message as an SNS push notification**, and then choose **Configure action**.



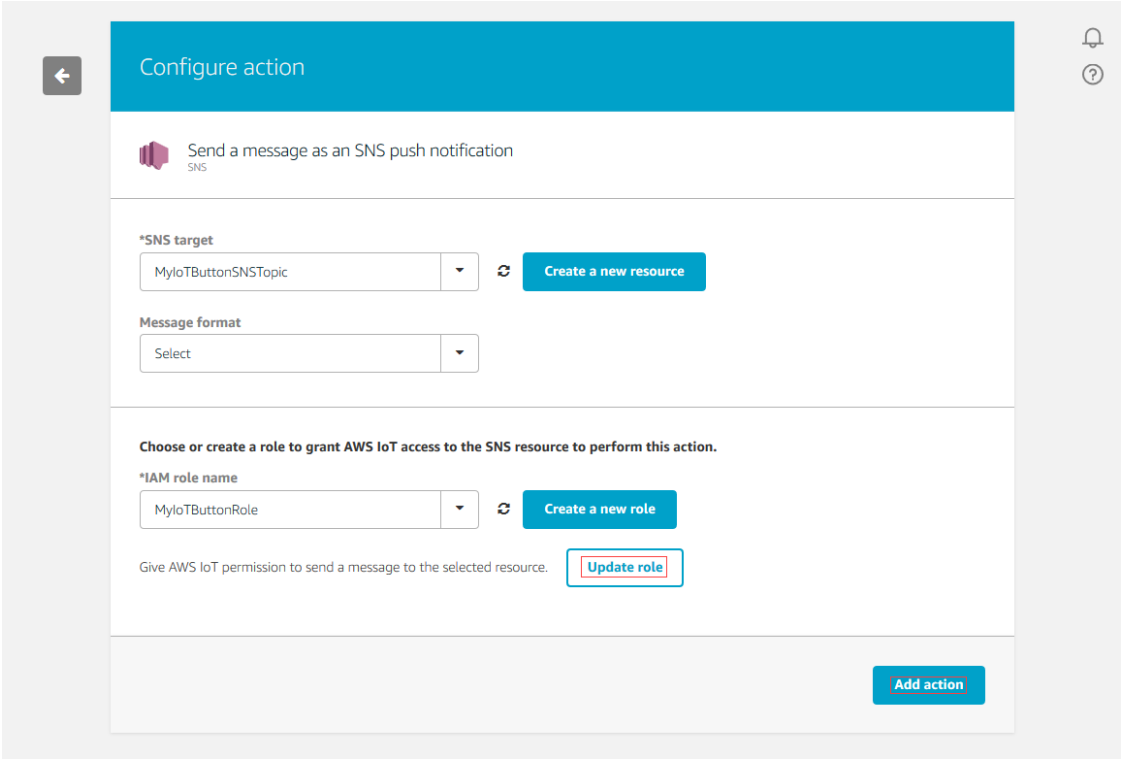
8. On the **Configure action** page, from the **SNS target** drop-down list, choose the Amazon SNS topic you created earlier.



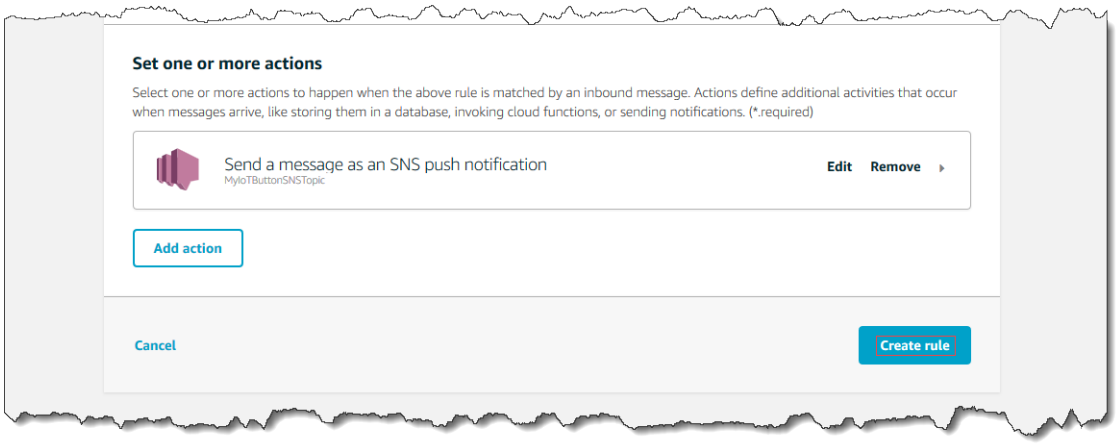
9. Now you need to give AWS IoT permission to publish to the Amazon SNS topic on your behalf when the rule is triggered. Choose **Create a new role**. Enter a name for your new role in the **IAM role name** field. After you have entered the name, choose **Create a new role** again. Select the newly created role from the **IAM role name** drop-down list.



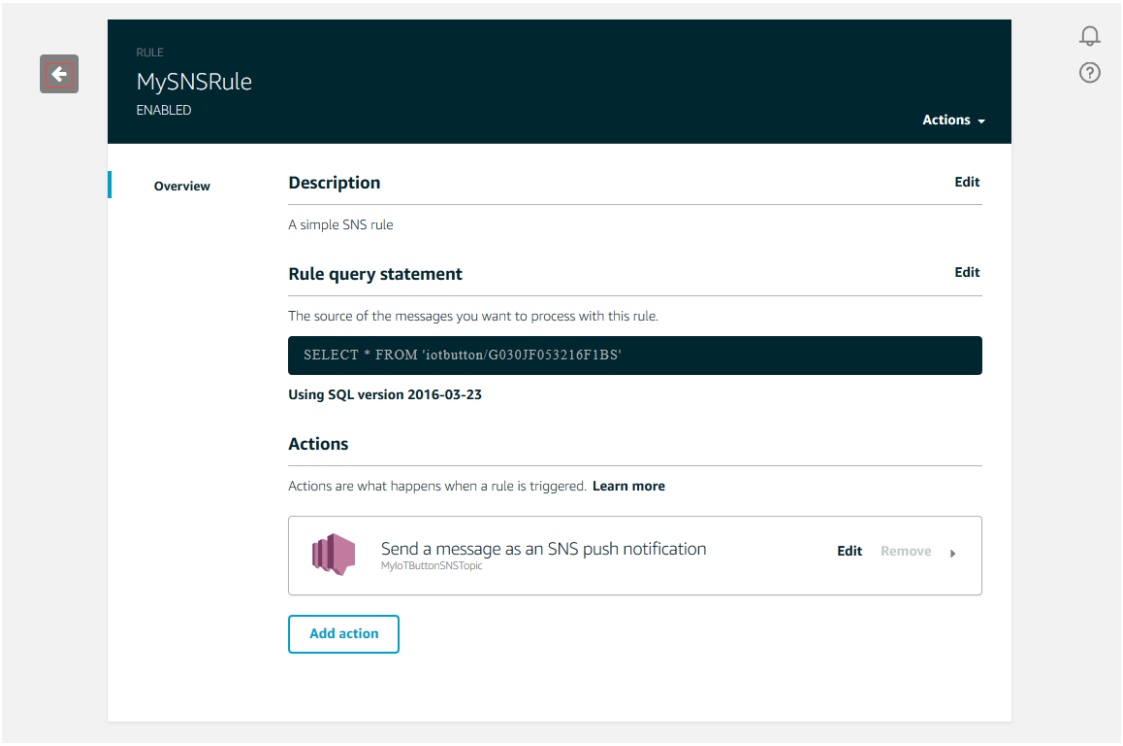
10. Choose **Update role** to apply the permissions to the newly created role, and then choose **Add action**.



11. On the **Create a Rule** page, choose **Create rule**.



12. On the **Overview** page for the rule, choose the left arrow to return to the AWS IoT dashboard.



For more information about creating rules, see AWS IoT Rules .

Test the Amazon SNS Rule

You can test your rule by using an AWS IoT button or the AWS IoT MQTT client.

AWS IoT Button

Press your button. You should receive an SMS text that shows the current battery charge level on your device.

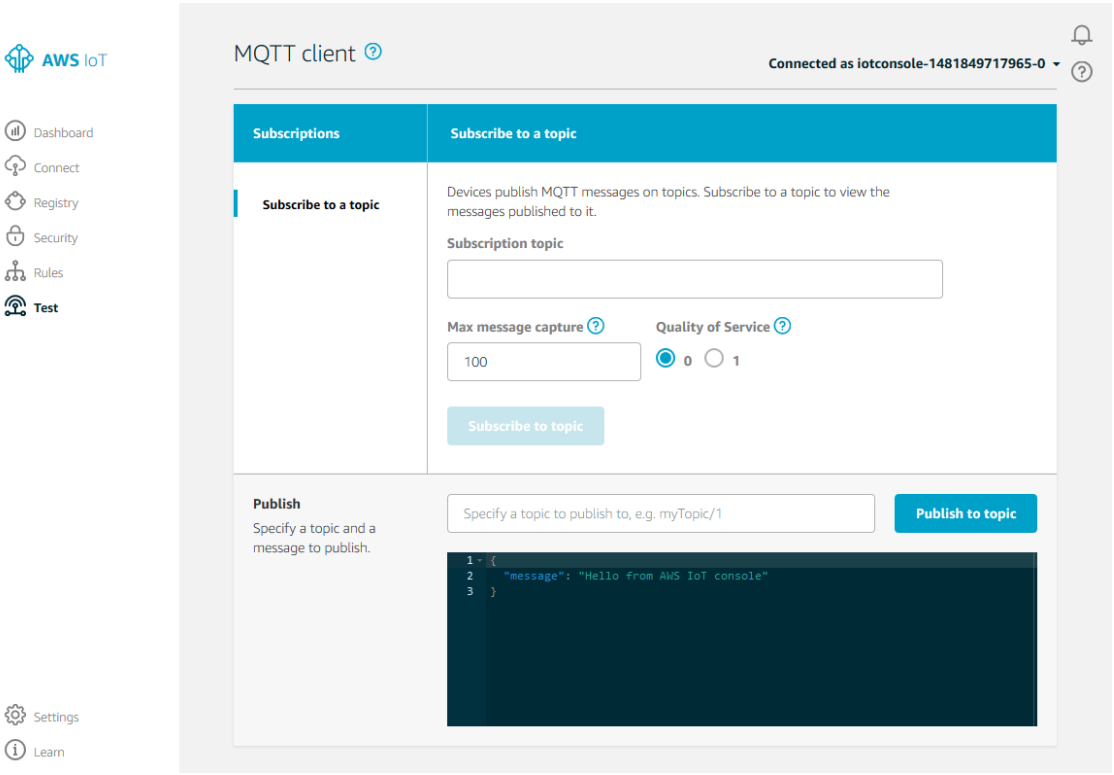
AWS IoT MQTT Client

To test your rule with the AWS IoT MQTT client:

1. In the AWS IoT console, in the left navigation pane, choose **Test**.
2. On the MQTT client page, in the **Publish** section, in **Specify a topic**, type **`my/topic`** or the topic you used in the rule. In the message payload section, type the following JSON:

```
{
    "message": "Hello, from AWS IoT console"
}
```

Note
If you are using a button, type **`iotbutton/`*`your-button-DSN`*** instead of **`my/topic`** in the **Specify a topic** field.



3. Choose **Publish to topic**. You should receive an Amazon SNS message on your cell phone.

Congratulations! You have successfully created and configured a rule that sends data received from a device to an Amazon SNS topic.

Next Steps

For more information about AWS IoT rules, see AWS IoT Rule Tutorials (p. 46) and AWS IoT Rules (p. 140).

AWS IoT Rule Tutorials

This guide includes tutorials that walk you through the creation and testing of AWS IoT rules. If you have not completed the AWS IoT Getting Started Tutorial (p. 19), we recommend you do that first. It shows you how to create an AWS account and connect your device to AWS IoT.

An AWS IoT rule consists of a SQL SELECT statement, a topic filter, and a rule action. Devices send information to AWS IoT by publishing messages to MQTT topics. The SQL SELECT statement allows you to extract data from an incoming MQTT message. The topic filter of an AWS IoT rule specifies one or more MQTT topics. The rule is triggered when an MQTT message is received on a topic that matches the topic filter. Rule actions allow you to take the information extracted from an MQTT message and send it to another AWS service. Rule actions are defined for AWS services like Amazon DynamoDB, AWS Lambda, Amazon SNS, and Amazon S3. By using a Lambda rule, you can call other AWS or third-party web services. For a complete list of rule actions, see AWS IoT Rule Actions (p. 148).

In these tutorials we assume you are using the AWS IoT button and will use `iotbutton/+` as the topic filter in the rules. If you do not have an AWS IoT button, you can buy one here.

The AWS IoT button sends a JSON payload that looks like this:

```
{
    "serialNumber" : "ABCDEFG12345",
    "batteryVoltage" : "2000mV",
    "clickType" : "SINGLE"
}
```

You can emulate the AWS IoT button by using an MQTT client like the AWS IoT MQTT client in the AWS IoT console. To emulate the AWS IoT button, publish a similar message on the `iotbutton/ABCDEFG12345` topic. The number after the / is arbitrary. It will be used as the serial number for the button.

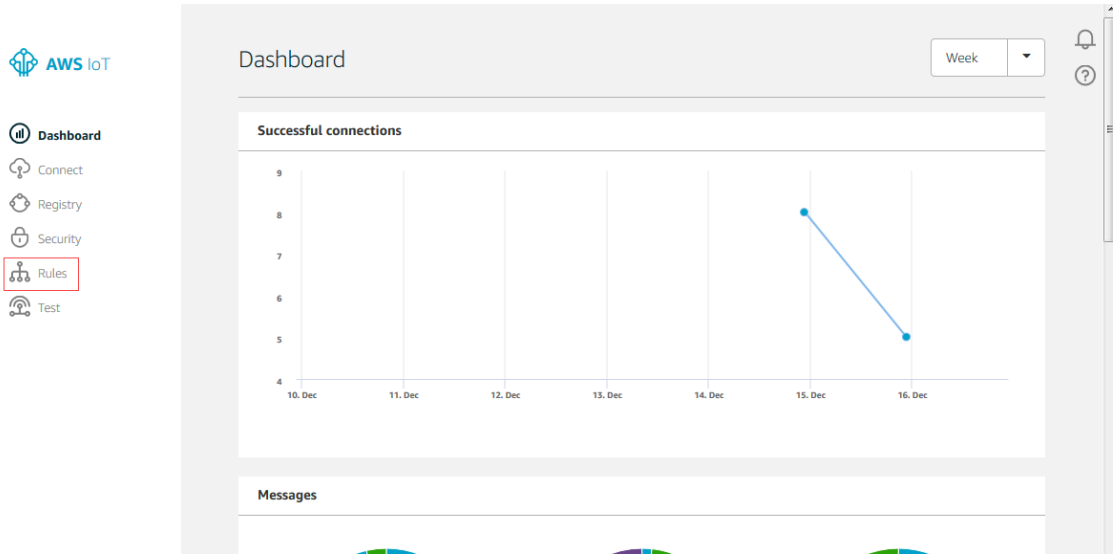
You can use your own device, but you will need to know on which MQTT topic your device publishes so you can specify it as the topic filter in the rule. For more information, see AWS IoT Rules (p. 140).

Creating a DynamoDB Rule

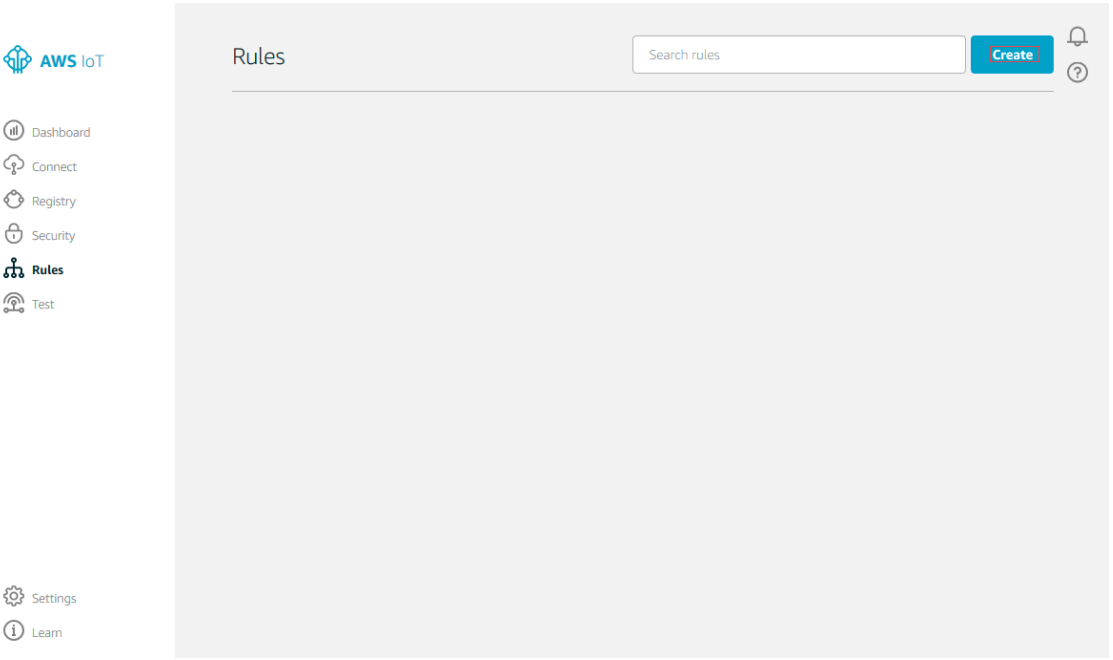
DynamoDB rules allow you to take information from an incoming MQTT message and write it to a DynamoDB table.

To create a DynamoDB rule:

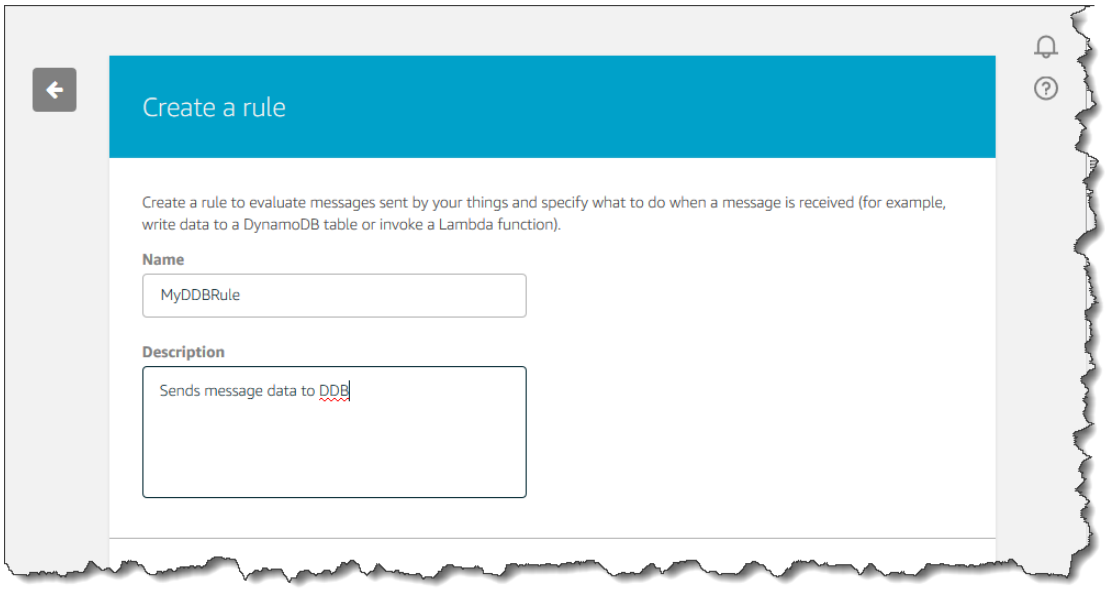
1. In the AWS IoT console, in the left navigation pane, choose **Rules**.



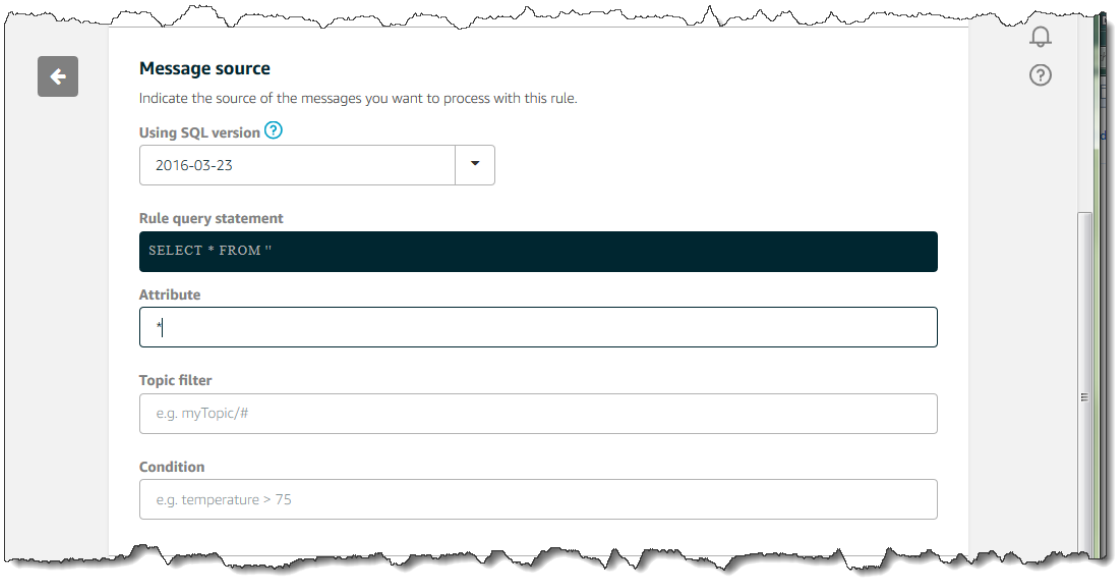
2. On the **Rules** page, choose **Create**.



3. On the **Create a rule** page, in the **Name** field, type a name for your rule. In the **Description** field, type a description for the rule.



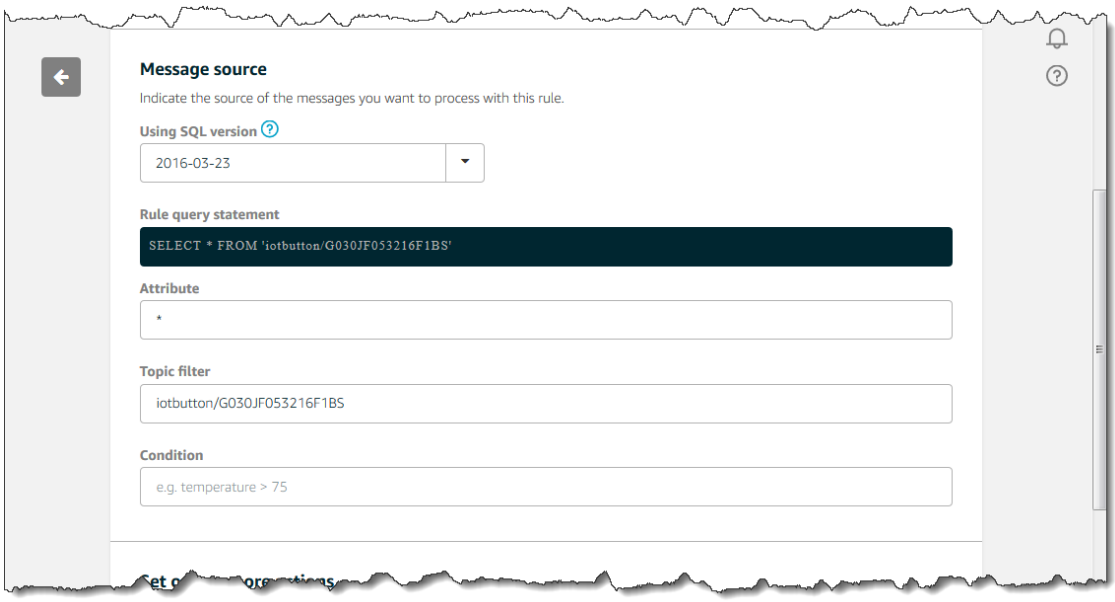
4. Scroll down to **Message source**. Choose the latest version from the **Using SQL version** drop-down list. In the **Attribute** field, type *. This specifies that you want to send the entire MQTT message that triggered the rule.



5. The rules engine uses the topic filter to determine which rules to trigger when an MQTT message is received. In the **Topic filter** field, type **`iotbutton/`*`your-button-DSN`***. If you are not using an AWS IoT button, type **`my/topic`** or the topic used in the rule.

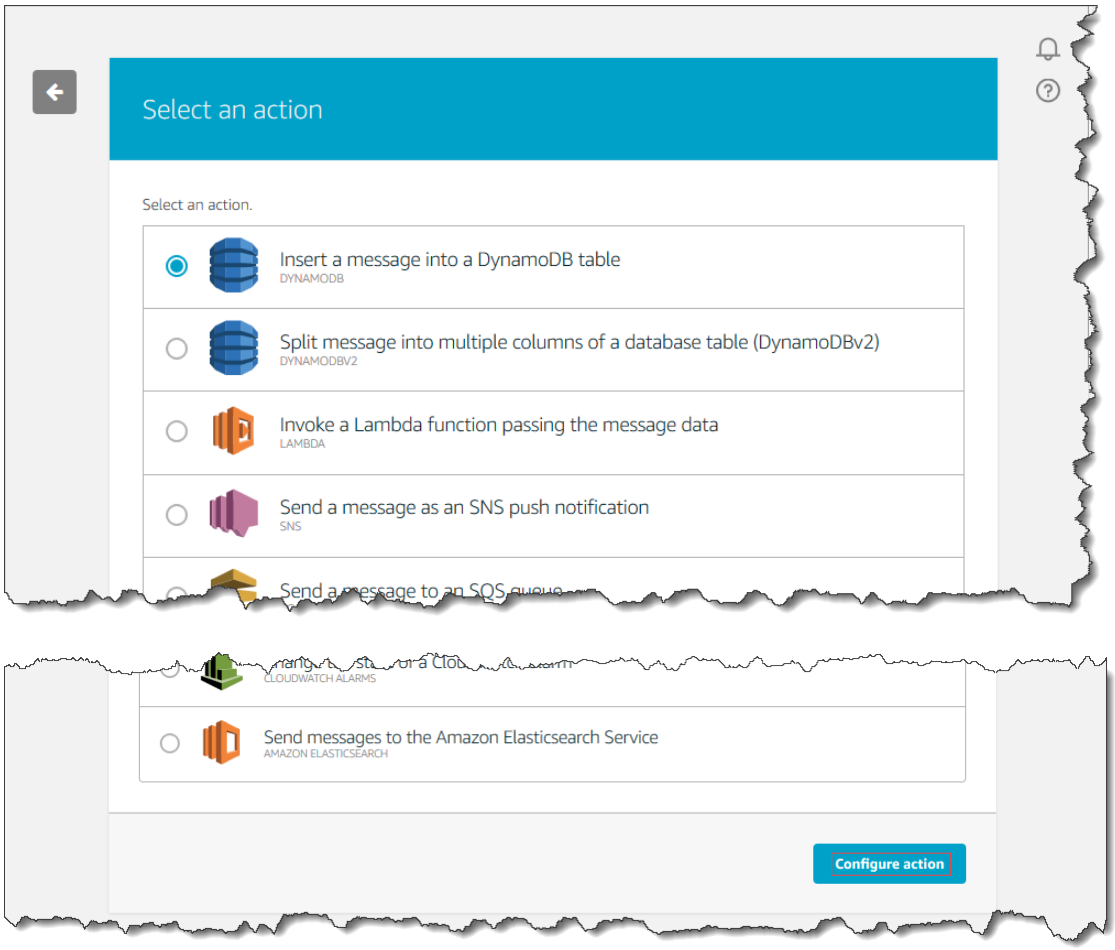
   Note
   You can find the DSN on the bottom of the button.

   Leave **Condition** blank.

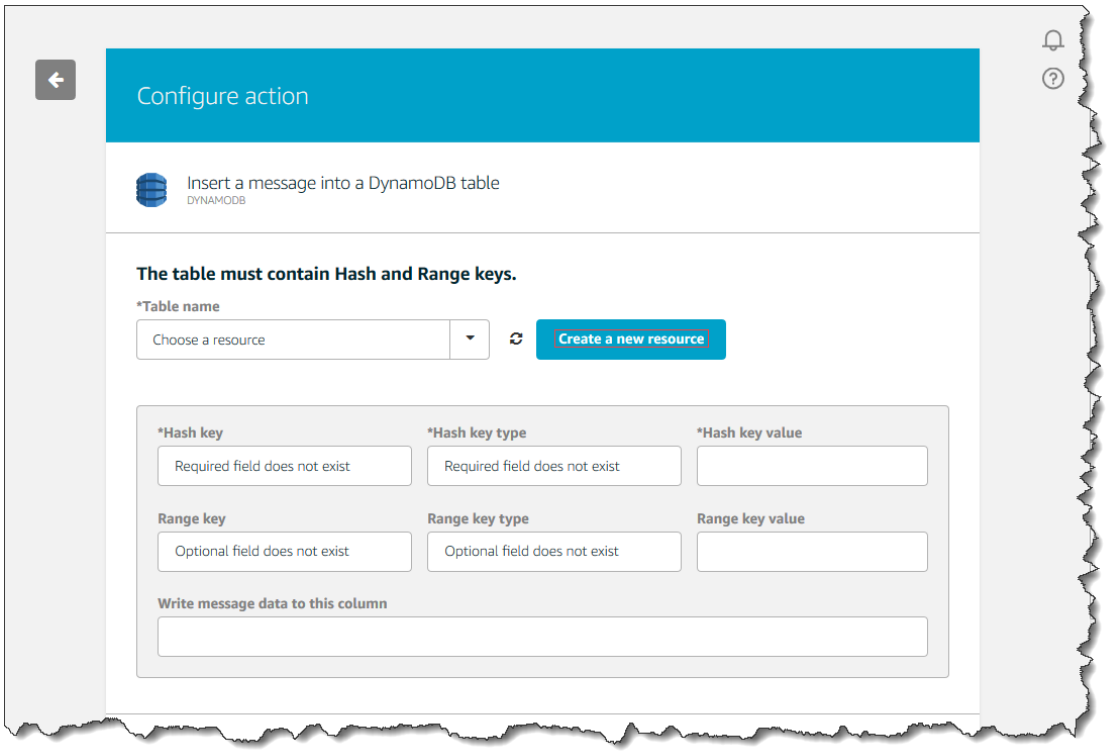


6. In **Set one or more actions**, choose **Add action**.

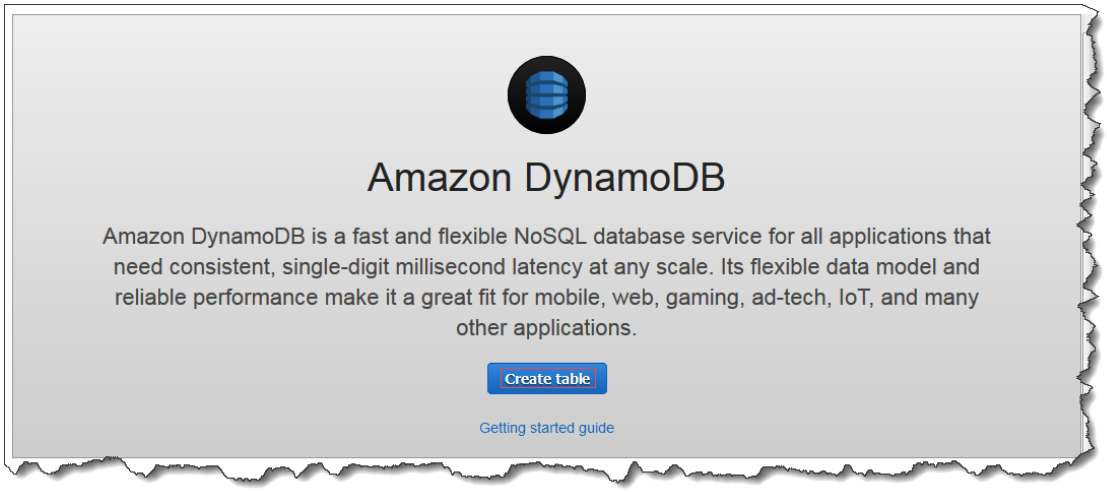
7. On the **Select an action** page, select **Insert a message into a DynamoDB table**, and then choose **Configure action**.



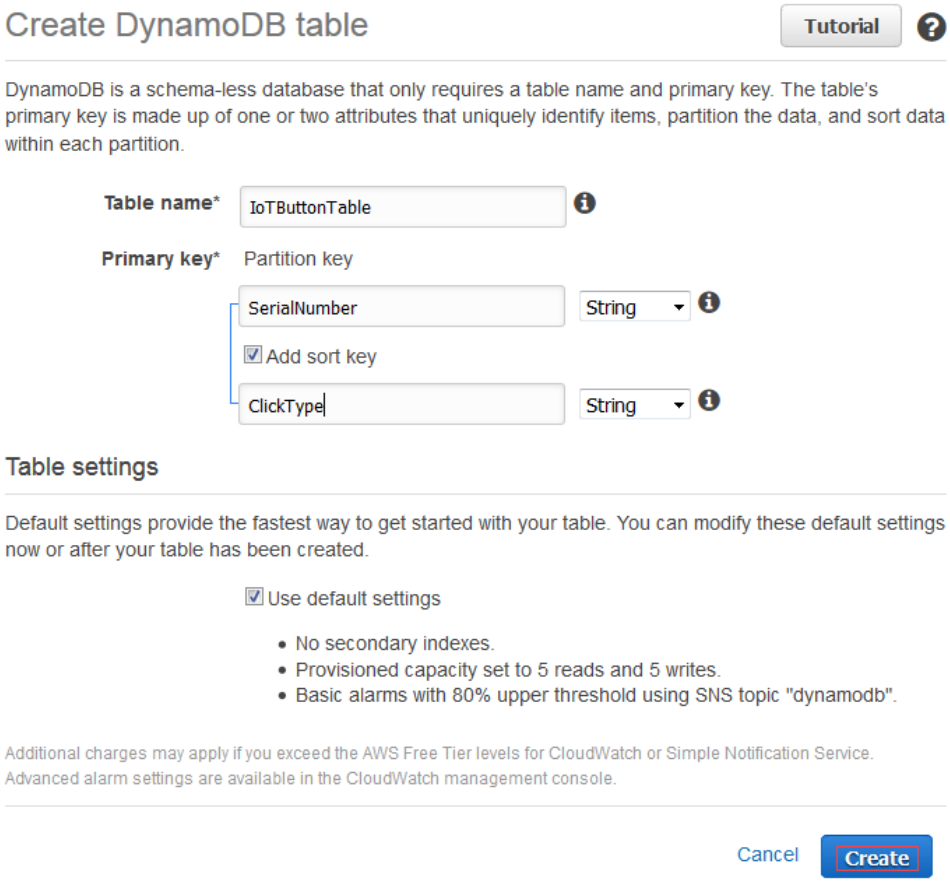
8. On the **Configure action** page, choose **Create a new resource**.



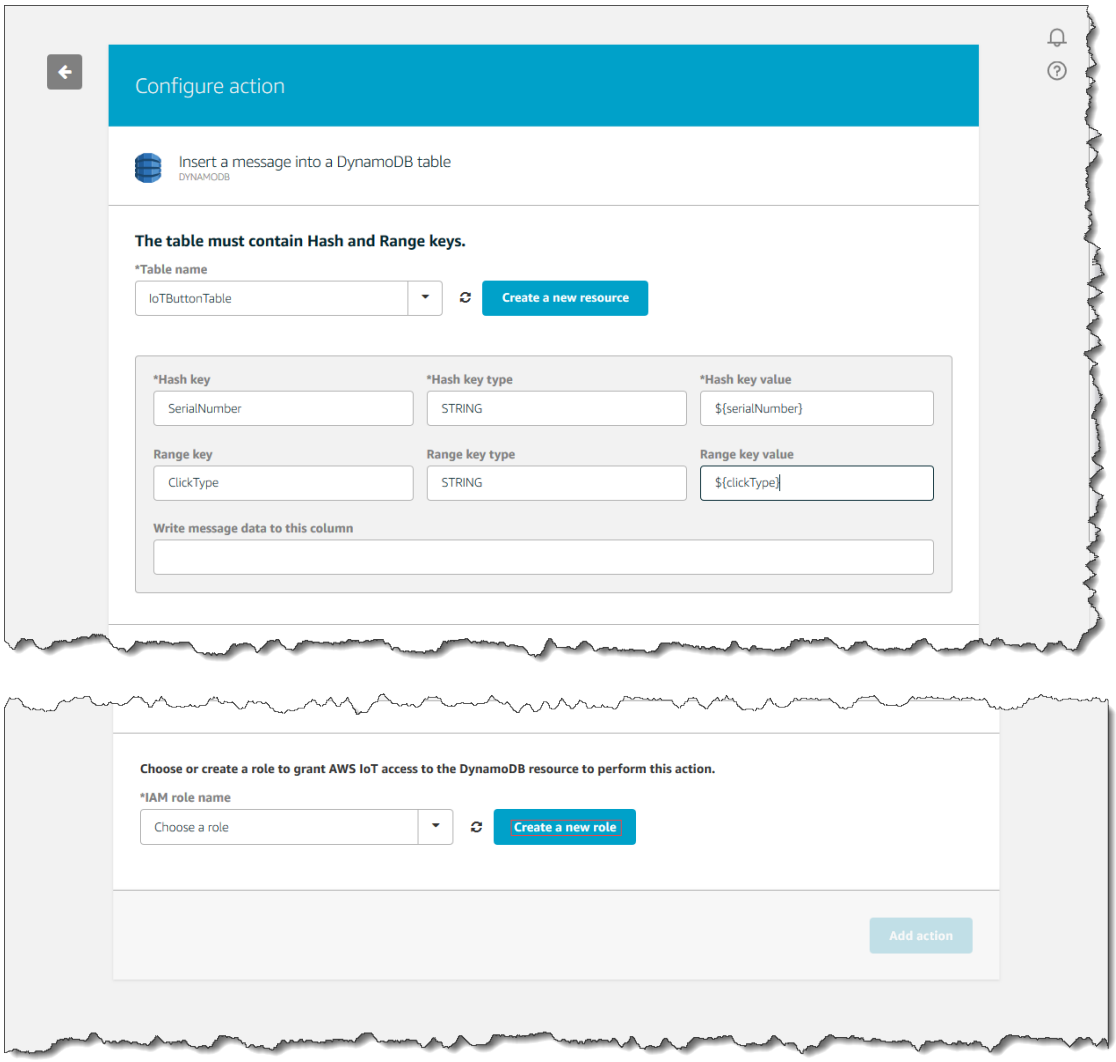
9. On the **Amazon DynamoDB** page, choose **Create table**.



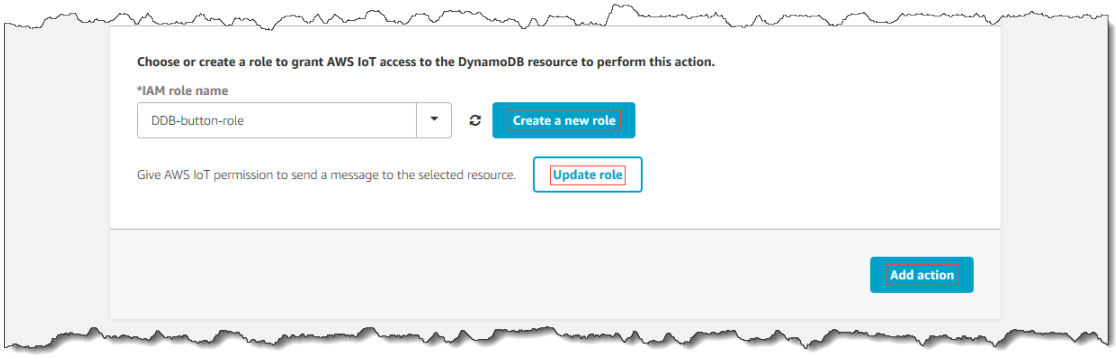
10. On the **Create DynamoDB table** page, type a name in the **Table name** field. In **Partition key**, type **`SerialNumber`**. Select the **Add sort key** check box, then type **`ClickType`** in the **Sort key** field. Select **String** for both the partition and sort keys.



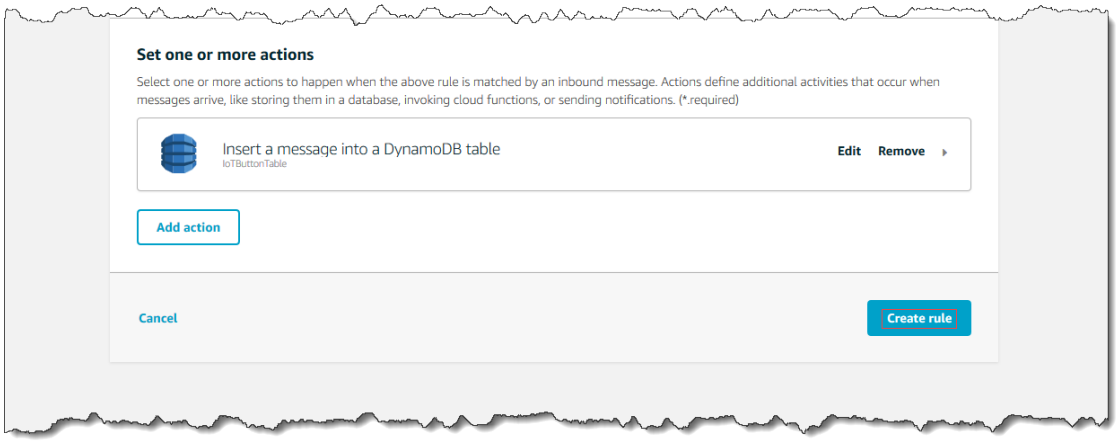
11. Choose **Create**. It will take a few seconds to create your DynamoDB table. Close the browser tab where the Amazon DynamoDB console is open. If you do not close the tab, your DynamoDB table will not be displayed in the **Table name** drop-down list on the AWS IoT **Configure action** page.
12. On the **Configure action** page, choose your new table from the **Table name** drop-down list. In **Hash key value**, type **`${serialNumber}`**. This instructs the rule to take the value of the `serialNumber` attribute from the MQTT message and write it into the **SerialNumber** column in the DynamoDB table. In **Range key value**, type **`${clickType}`**. This writes the value of the `clickType` attribute into the **ClickType** column. Leave **Write message data to this column** blank. By default, the entire message will be written to a column in the table called Payload. Choose **Create a new role**.



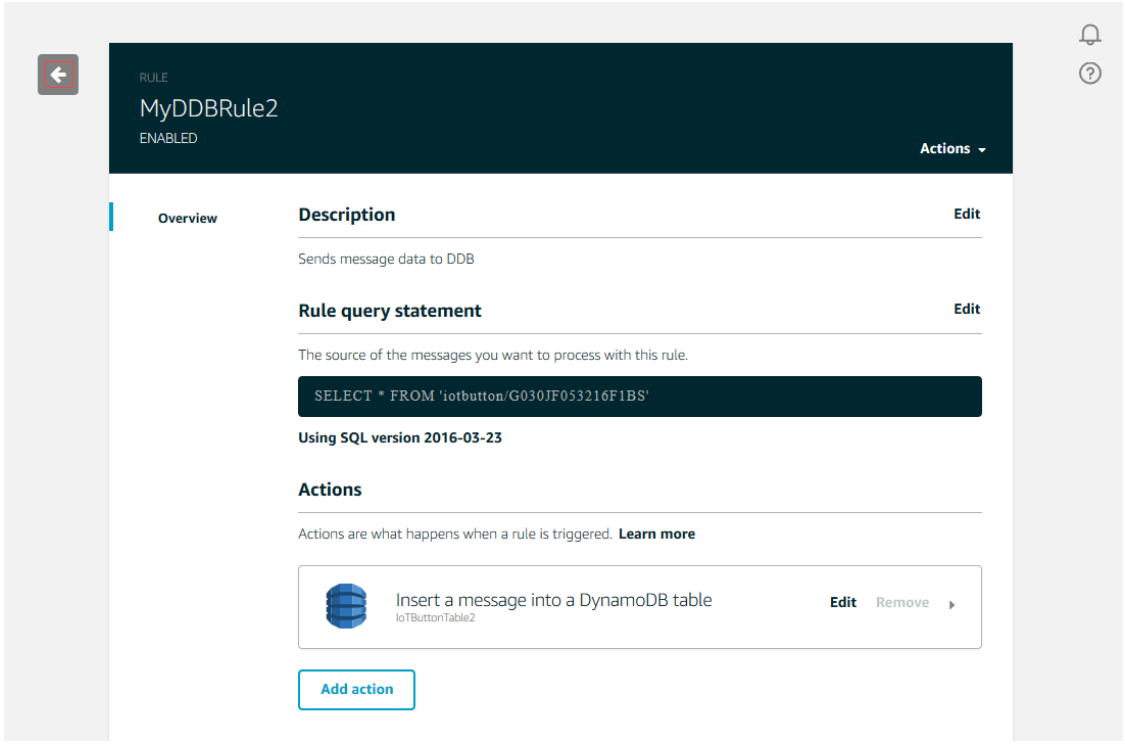
13. Type a unique name in **IAM role name**, and then choose the **Create a new role** button again. Choose the role you just created and choose **Update role**. Then choose **Add action**.



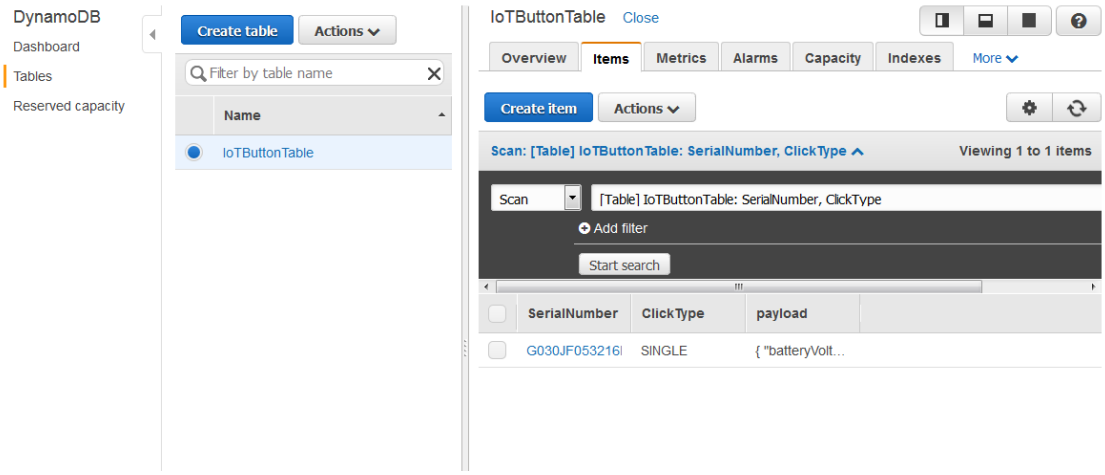
14. Choose **Create rule** to create the rule.



15. A confirmation message shows the rule has been created. Choose the left arrow to return to the **Rules** page.



16. Test the rule by pressing your configured AWS IoT button or using an MQTT client to publish a message on a topic that matches your rule's topic filter. Finally, return to the DynamoDB console and select the table you created to view the entry for your button press or message.



Creating a Lambda Rule

You can define a rule that calls a Lambda function, passing in data from the MQTT message that triggered the rule. This allows you to process the incoming message and then call another AWS or third-party service.

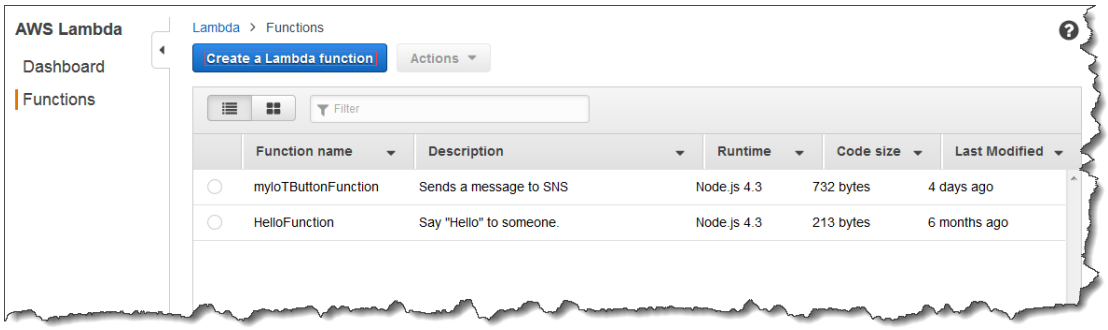
In this tutorial, we assume you have completed the AWS IoT Getting Started Tutorial (p. 19) in which you create and subscribe to an Amazon SNS topic using your cell phone number. You will create a Lambda function that publishes a message to the Amazon SNS topic you created in the AWS IoT Getting Started Tutorial (p. 19). You will also create a Lambda rule that calls the Lambda function, passing in some data from the MQTT message that triggered the rule.

In this tutorial, we also assume you are using an AWS IoT button to trigger the Lambda rule. If you do not have an AWS IoT button, you can buy one here or you can use an MQTT client to send an MQTT message that will trigger the rule.

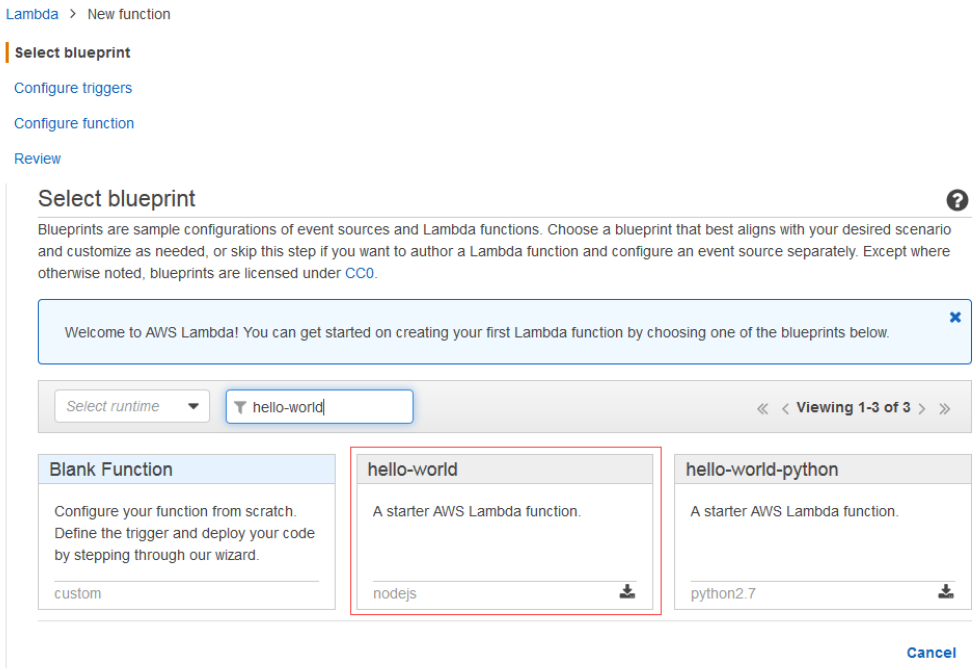
Create the Lambda Function

To create the Lambda function:

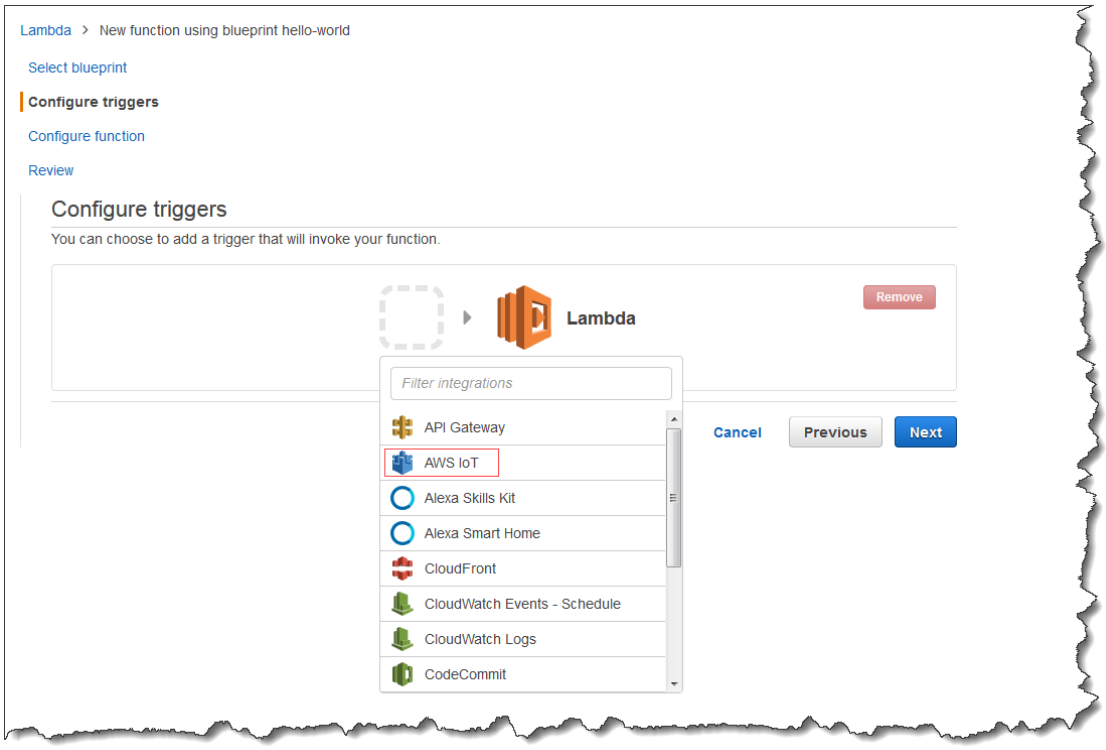
1. In the AWS Lambda console, choose **Get Started Now** or, if you have created a Lambda function before, choose **Create a Lambda function**.



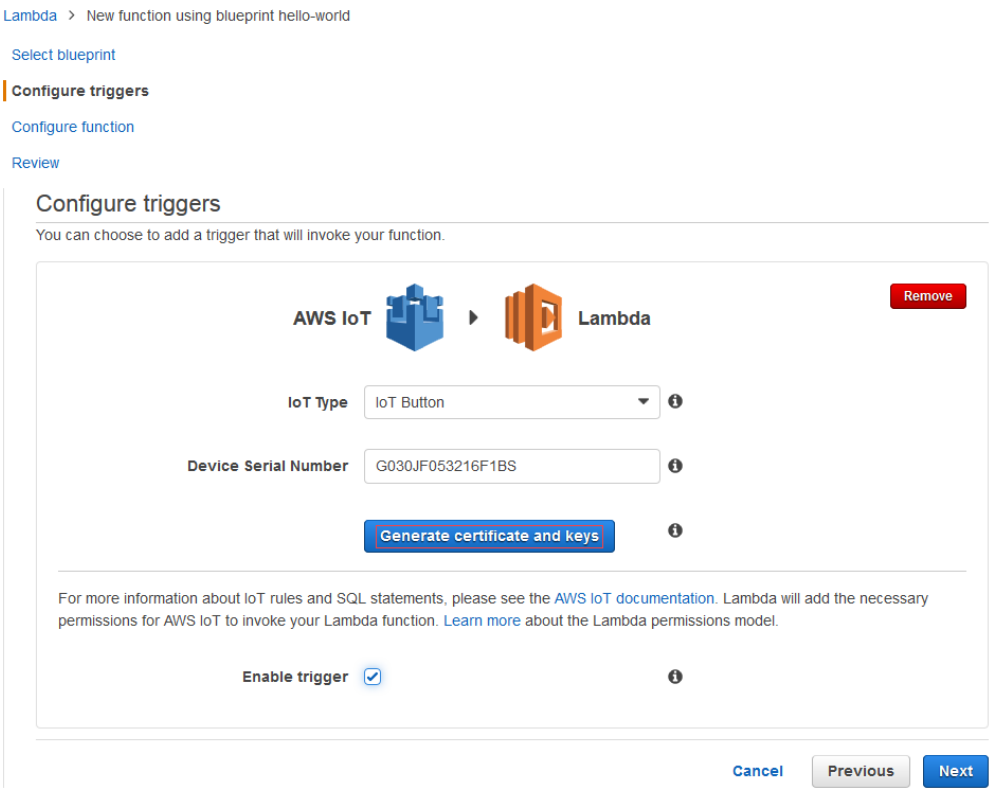
2. On the **Select blueprint** page, in the **Filter** field, type **`hello-world`**, and then choose the **hello-world** blueprint.



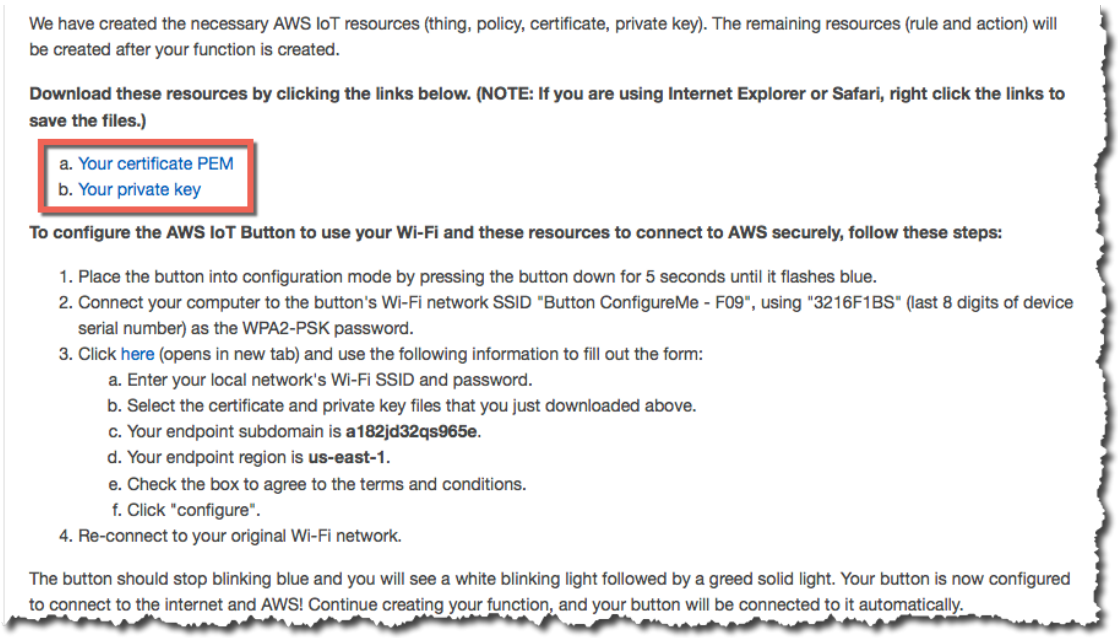
3. On the **Configure triggers** page, select the box to the left of the Lambda icon, and select **AWS IoT** from the drop-down menu.



4. In the **Device Serial Number** field, enter your button's device serial number (DSN). Your DSN is printed on the back of your AWS IoT button. If you have not already generated a certificate and private key for your AWS IoT button, choose **Generate certificate and keys**. Otherwise, skip to step 6.

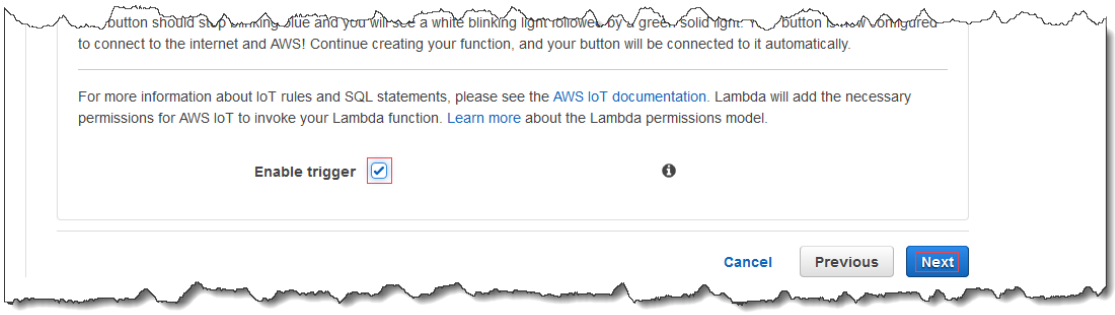


5. Choose the links to download your certificate PEM and private key. Save these files in a secure location on your computer.

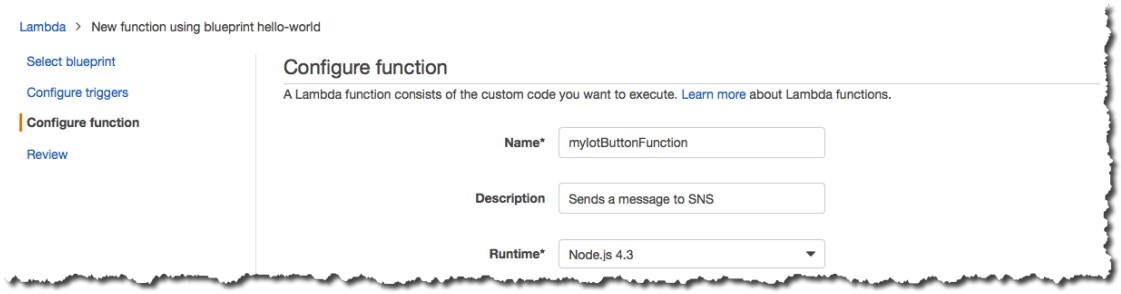


Follow the instructions on the screen to configure your AWS IoT button.

6. Make sure that the **Enable trigger** check box is selected and choose **Next**.



7. On the **Configure function** page, type a name and description for the Lambda function. In **Runtime**, choose **Node.js 4.3**.



8. Scroll down to the **Lambda function code** section of the page. Replace the existing code with the following:

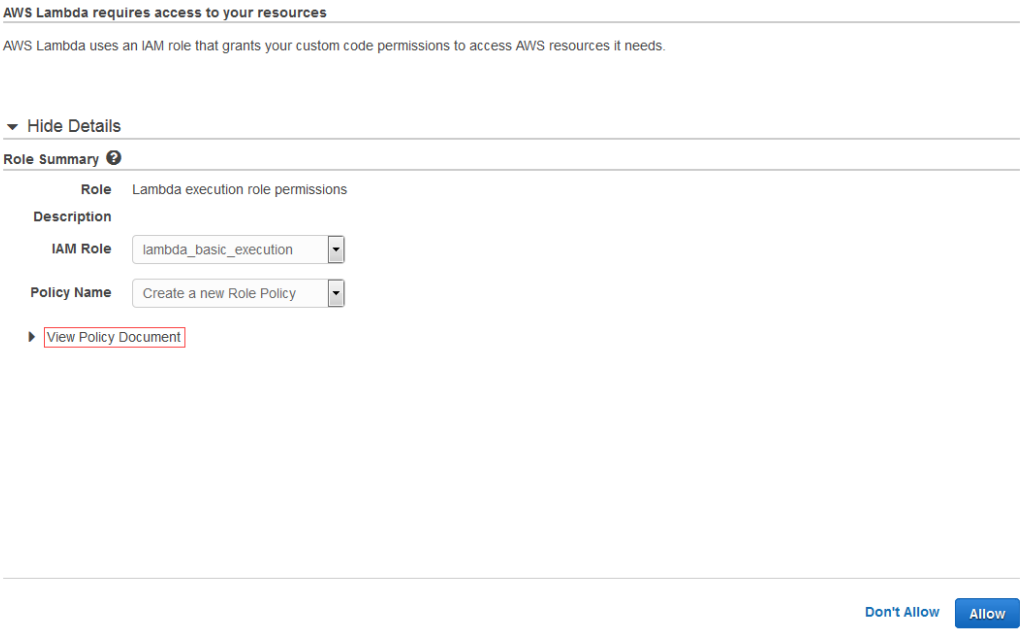
```
console.log('Loading function');
    // Load the AWS SDK
    var AWS = require("aws-sdk");

    // Set up the code to call when the Lambda function is invoked
    exports.handler = (event, context, callback) => {
        // Load the message passed into the Lambda function into a JSON object
        var eventText = JSON.stringify(event, null, 2);

        // Log a message to the console, you can view this text in the Monitoring tab
 in the Lambda console or in the CloudWatch Logs console
        console.log("Received event:", eventText);

        // Create a string extracting the click type and serial number from the message
 sent by the AWS IoT button
        var messageText = "Received  " + event.clickType + " message from button ID: "
 + event.serialNumber;

        // Write the string to the console
        console.log("Message to send: " + messageText);

        // Create an SNS object
        var sns = new AWS.SNS();

        // Populate the parameters for the publish operation
        // - Message : the text of the message to send
        // - TopicArn : the ARN of the Amazon SNS topic to which you want to publish
        var params = {
            Message: messageText,
            TopicArn: "arn:aws:sns:us-east-1:123456789012:MyIoTButtonSNSTopic"
         };
         sns.publish(params, context.done);
    };
```

Note
Replace the value of the TopicArn with the ARN of the Amazon SNS topic you created previously.

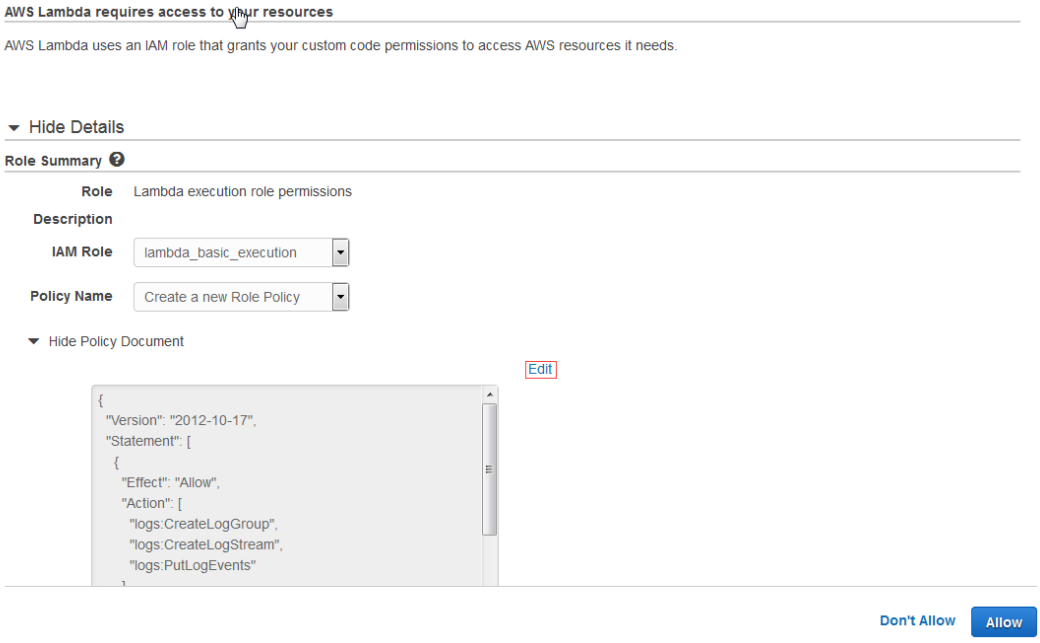
9. Scroll down to the **Lambda function handler and role** section of the page. For **Role**, choose **Create a custom role**. The IAM console will open, allowing you to create an IAM role that Lambda can assume when executing the Lambda function.

   To edit the role's policy to give it permission to publish to your Amazon SNS topic:

   a. Choose **View Policy Document**.



   b. Choose **Edit** to edit the role's policy.



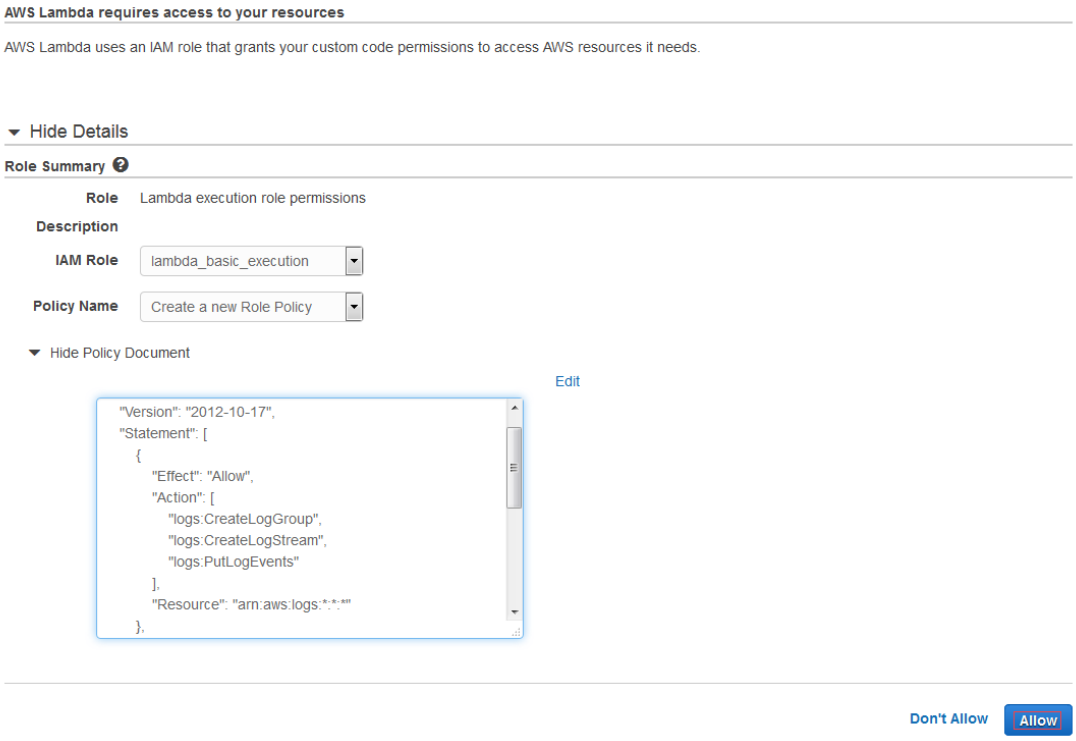
c. Replace the policy document with the following:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": "arn:aws:logs:*:*:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "sns:Publish"
            ],
            "Resource": "arn:aws:sns:us-east-1:123456789012:MyIoTButtonSNSTopic"
        }
    ]
}
```

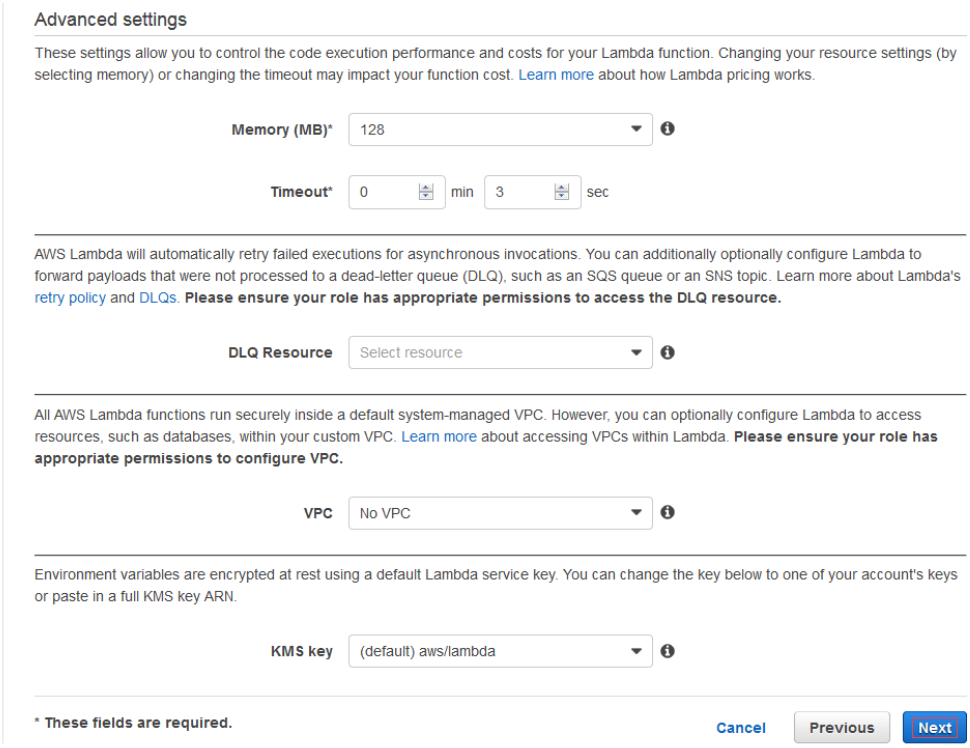
This policy document adds permission to publish to your Amazon SNS topic.

Note
Replace the value of the second `Resource` with the ARN of the Amazon SNS topic you created previously.

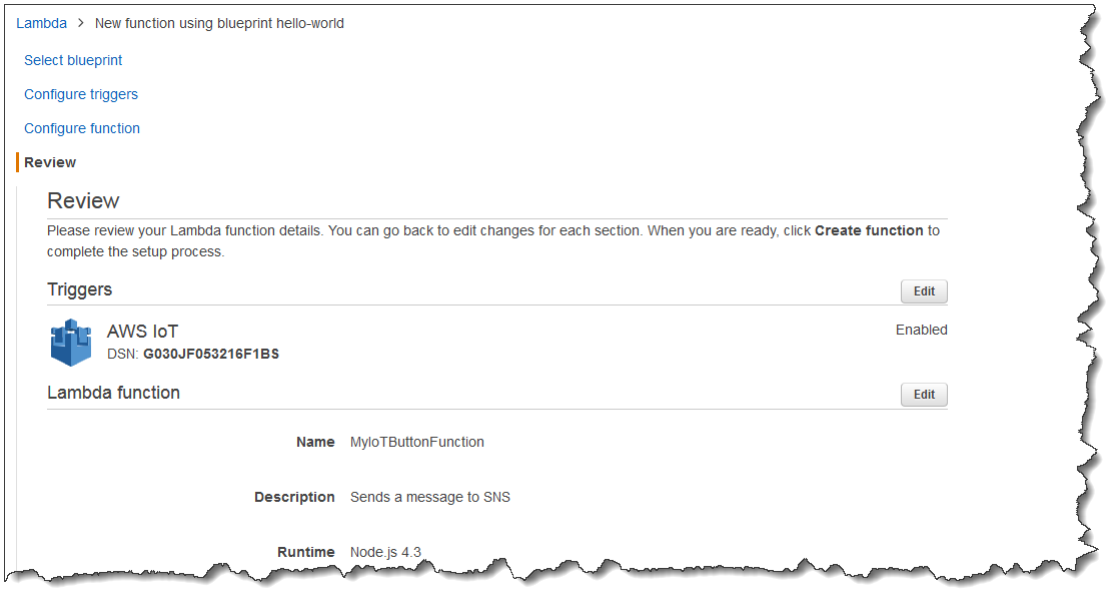
10. Choose **Allow**.



11. Leave the settings on the **Advanced settings** page at their defaults, and choose **Next**.



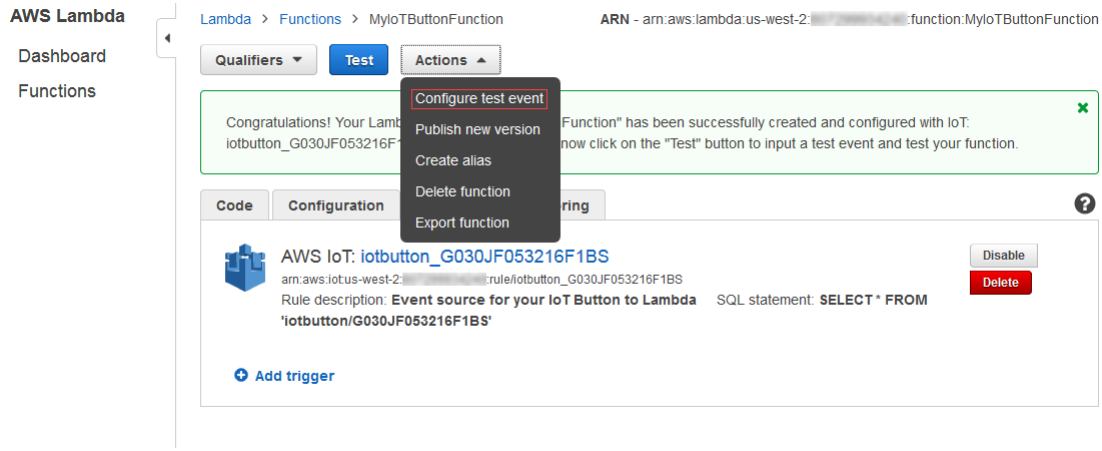
12. On the **Review** page, choose **Create function**.



Test Your Lambda Function

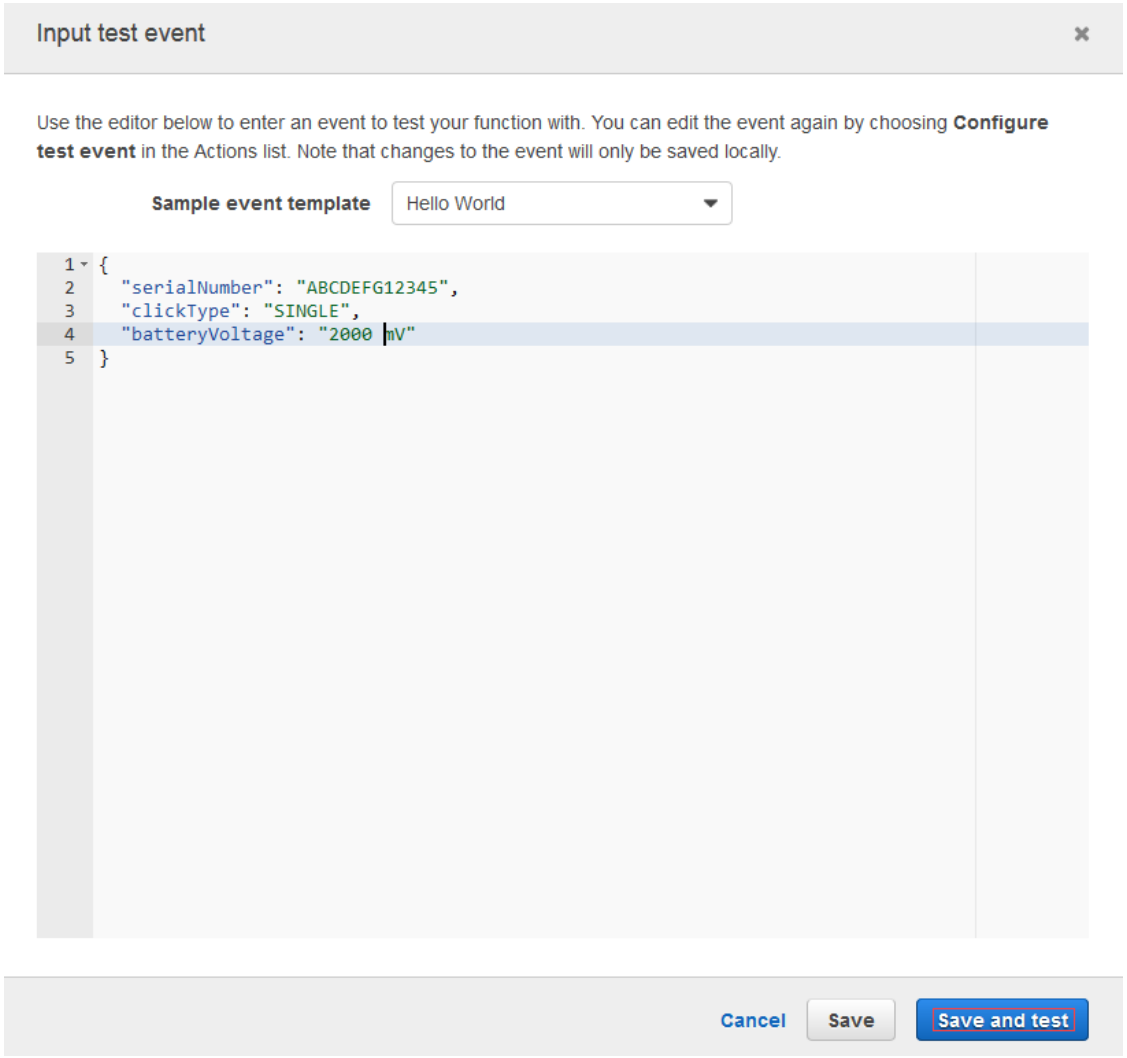
To test the Lambda function:

1. From the **Actions** menu, choose **Configure test event**.

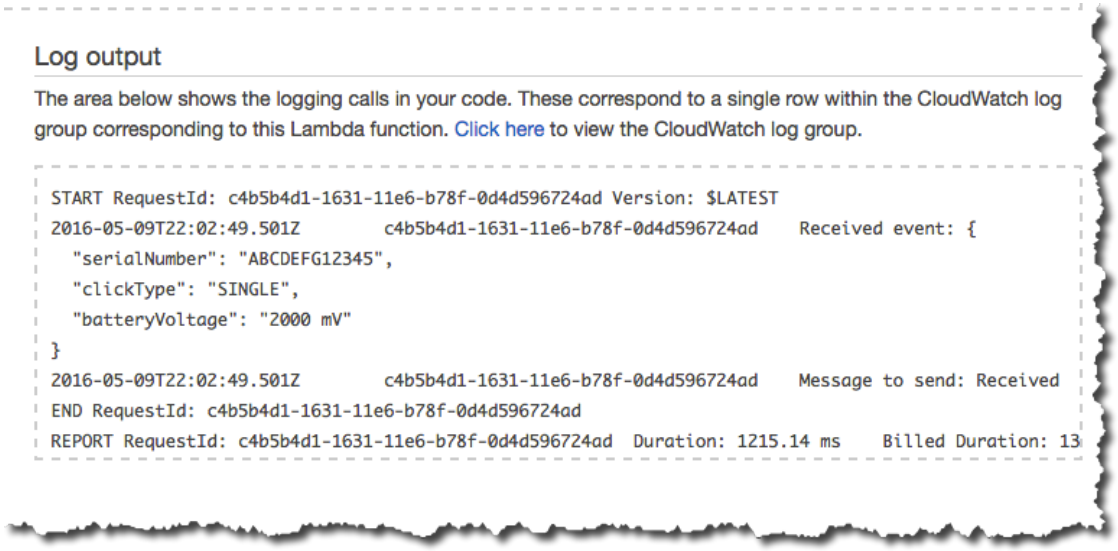


2. Copy and paste the following JSON into the **Input test event** page, and then choose **Save and test**.

```
{
    "serialNumber": "ABCDEFG12345",
    "clickType": "SINGLE",
    "batteryVoltage": "2000 mV"
}
```



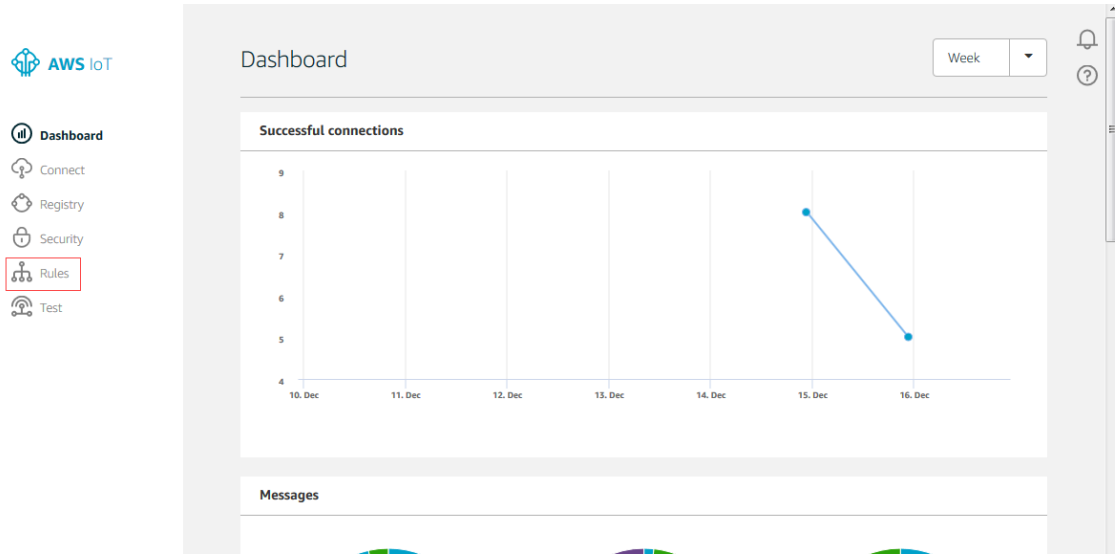
3. In the AWS Lambda console, scroll to the bottom of the page. The **Log output** section displays the output the Lambda function has written to the console.



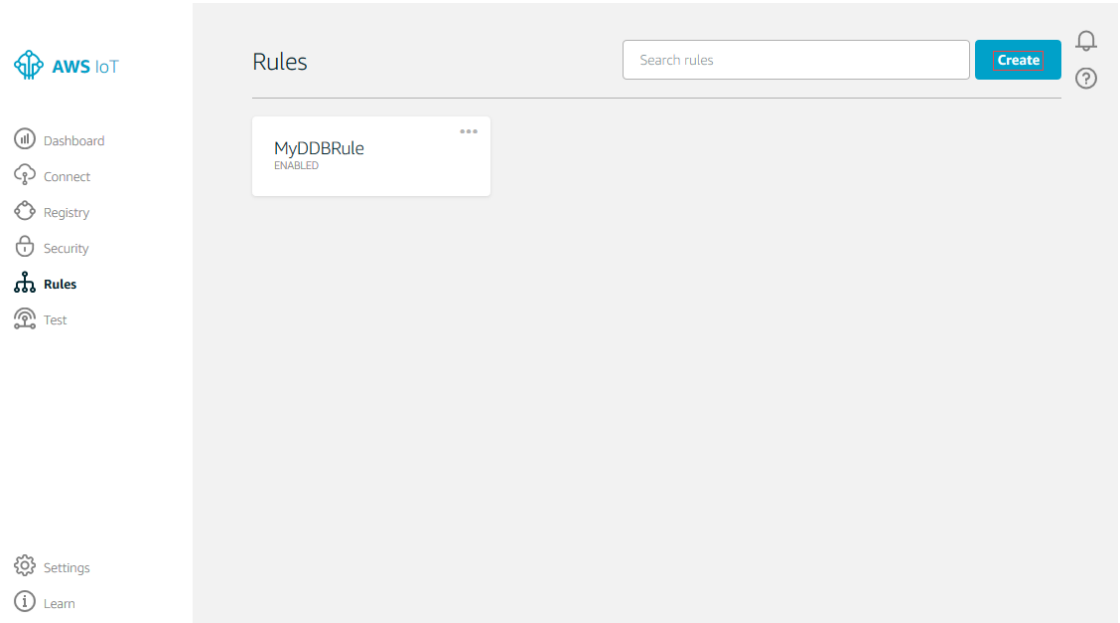
Creating a Lambda Rule

Now that you have created a Lambda function, you can create a rule that invokes the Lambda function.

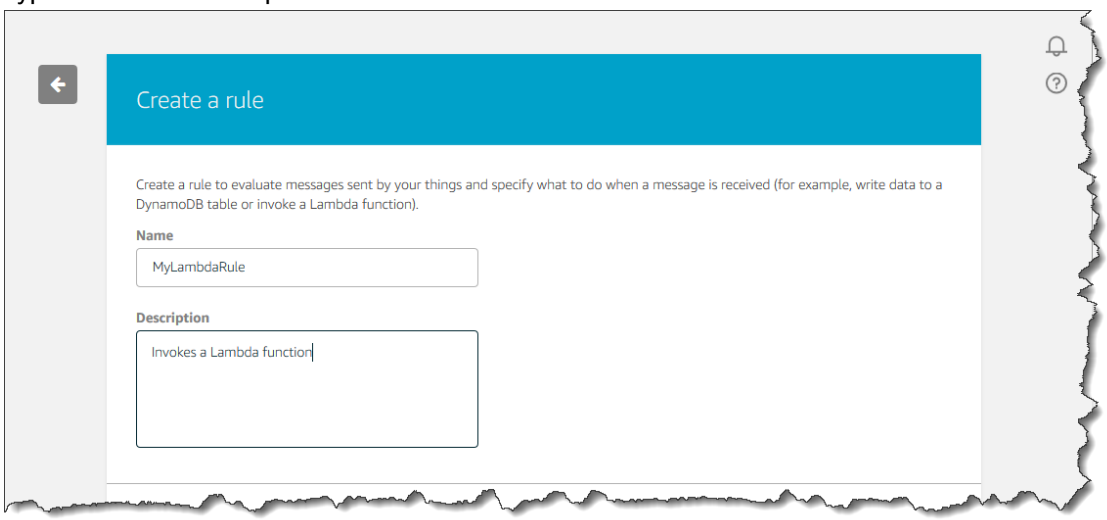
1. In the AWS IoT console, in the left navigation pane, choose **Rules**.



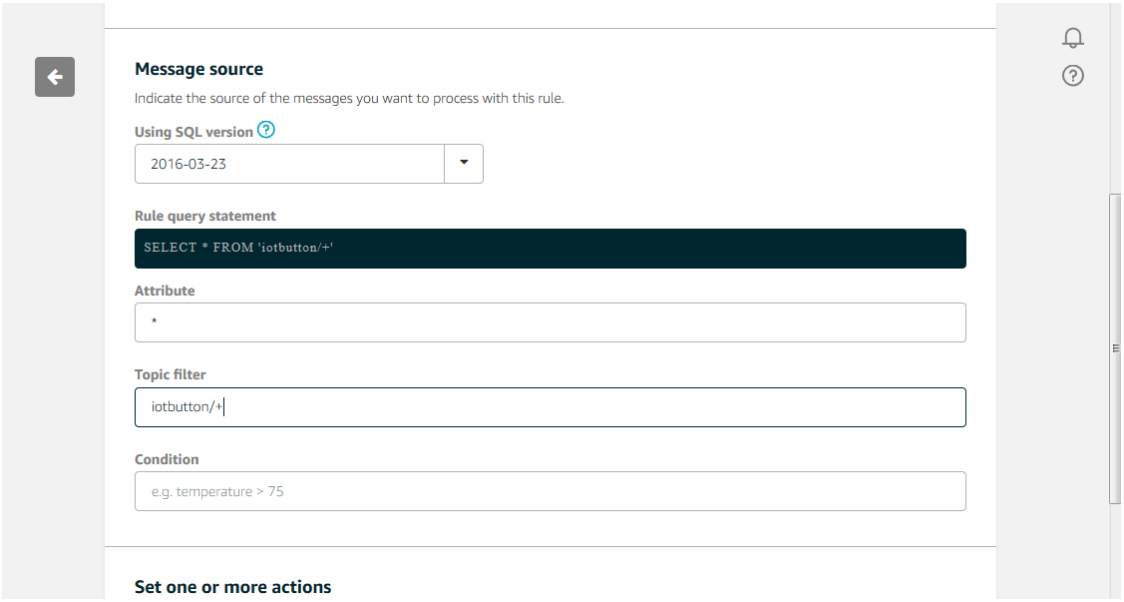
2. On the **Rules** page, choose **Create**.



3. Type a name and description for the rule.

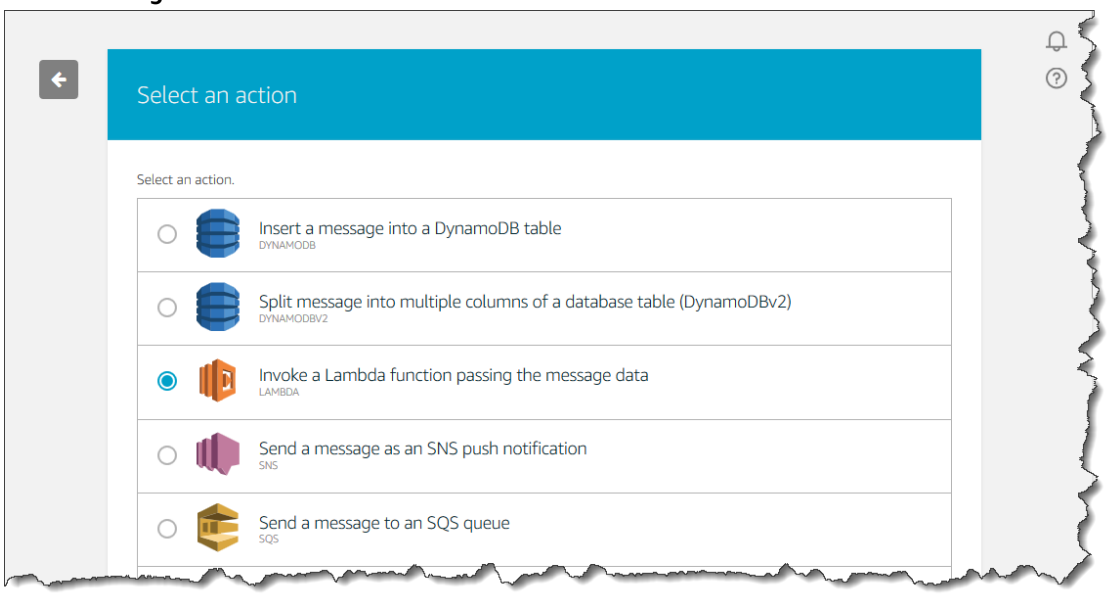


4. Enter the following settings for the rule:

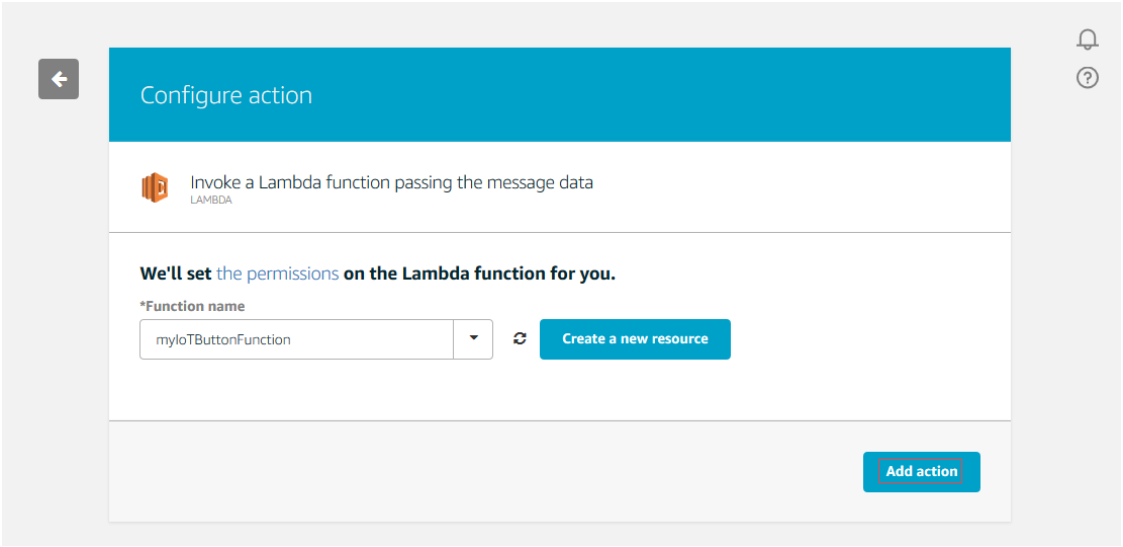


5. In **Set one or more actions**, choose **Add action**.

6. On the **Select an action** page, select **Invoke a Lambda function passing the message data** and then choose **Configure action**.



7. From the **Function name** drop-down list, choose your Lambda function name, then choose **Add action**.



8. Choose **Create rule** to create your Lambda function.

Test Your Lambda Rule

In this tutorial, we assume you have completed the AWS IoT Getting Started Tutorial (p. 19), which covers:

- Configuring an AWS IoT button.
- Creating and subscribing to an Amazon SNS topic with a cell phone number.

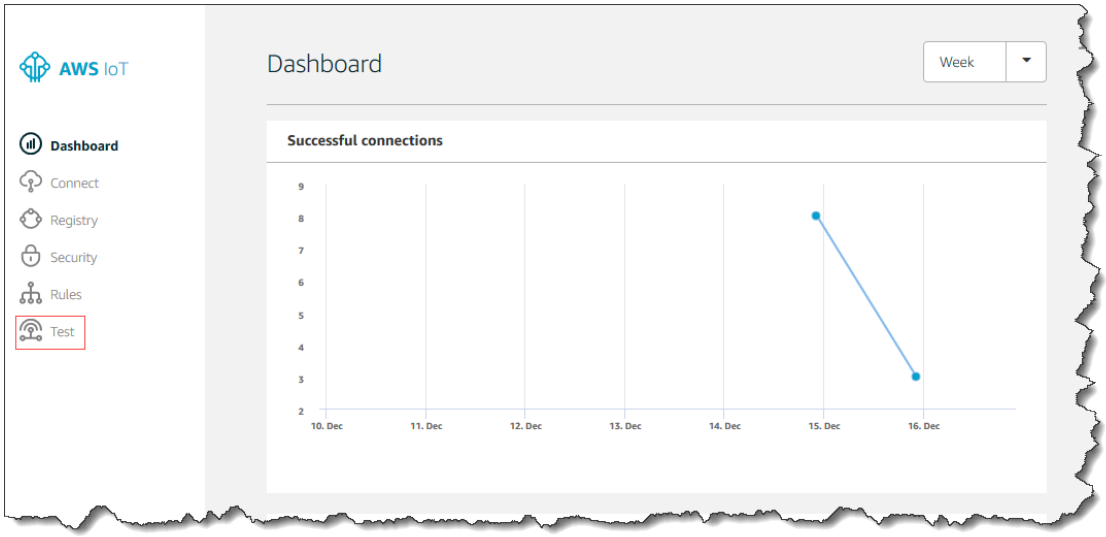
Now that your button is configured and connected to Wi-Fi and you have configured an Amazon SNS topic, you can press the button to test your Lambda rule. You should receive an SMS text message on your phone that contains the serial number of your button, the type of button press (SINGLE or DOUBLE), and the battery voltage.

The message should look like the following:

```
IOT BUTTON> {
    "serialNumber" : "ABCDEFG12345",
    "clickType" : "SINGLE",
    "batteryVoltage" : "2000 mV"
}
```

If you do not have a button, you can buy one here or you can use the AWS IoT MQTT client instead.

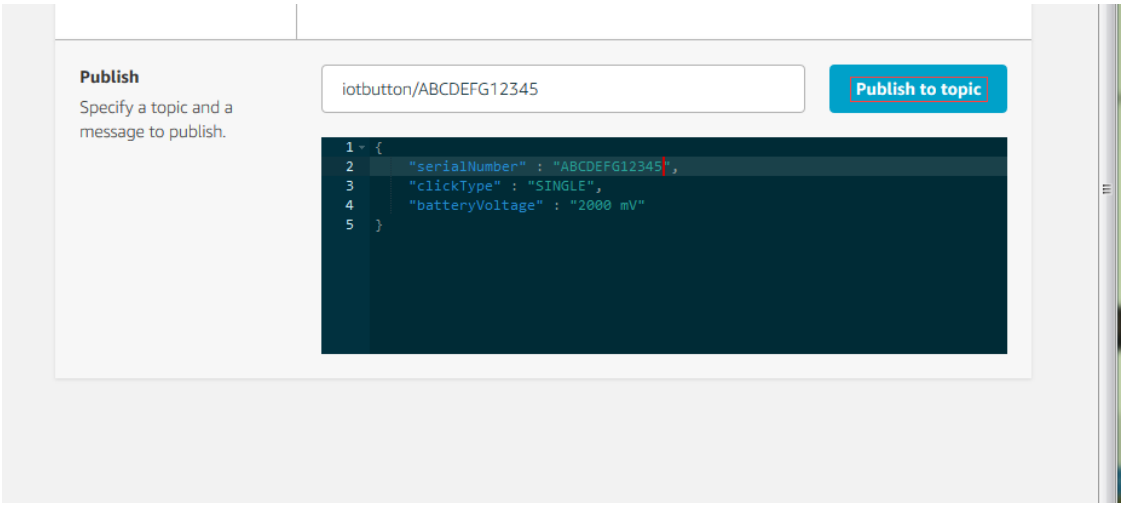
1. In the AWS IoT console, choose **Test**.



2. On the **MQTT client** page, in the **Publish** section, in **Specify a topic**, type **`iotbutton/ABCDEFG12345`**.

   In **Payload**, type the following JSON, and then choose **Publish to topic**.

```
{
    "serialNumber" : "ABCDEFG12345",
    "clickType" : "SINGLE",
    "batteryVoltage" : "2000 mV"
}
```



3. You should receive a message on your cell phone.

AWS IoT SDK Tutorials

The AWS IoT Device SDKs help you to easily and quickly connect your devices to AWS IoT. 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.

This guide provides step-by-step instructions for connecting your Raspberry Pi to the AWS IoT platform and setting it up for use with the AWS IoT Embedded C SDK and Device SDK for Javascript. After following the steps in this guide, you will be able to connect to the AWS IoT platform and run sample apps included with these AWS IoT SDKs.

Contents

Connecting Your Raspberry Pi

Follow these steps to connect your Raspberry Pi to the AWS IoT platform.

Prerequisites

- A fully set up Raspberry Pi board with Internet access

  For information about setting up your Raspberry Pi, see Raspberry Pi Quickstart Guide.
- Chrome or Firefox (Iceweasel) browser

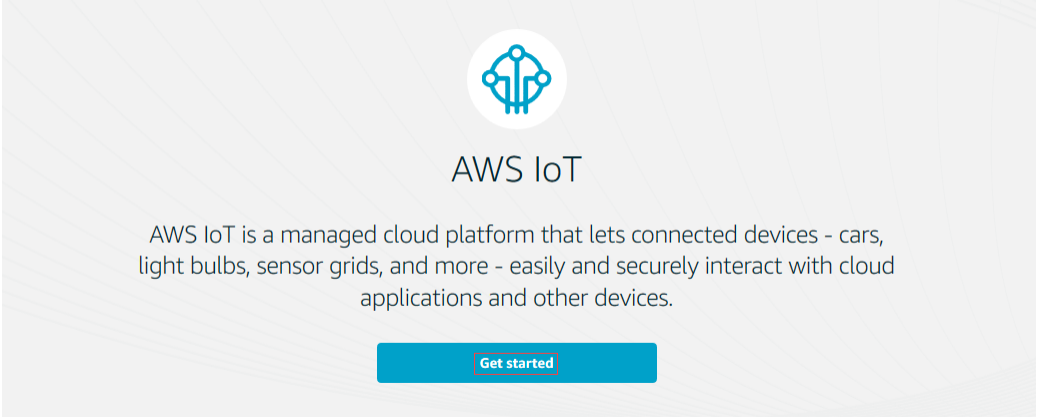
  For information about installing Iceweasel, see the instructions on the Embedded Linux wiki.

In this guide, the following hardware and software are used:

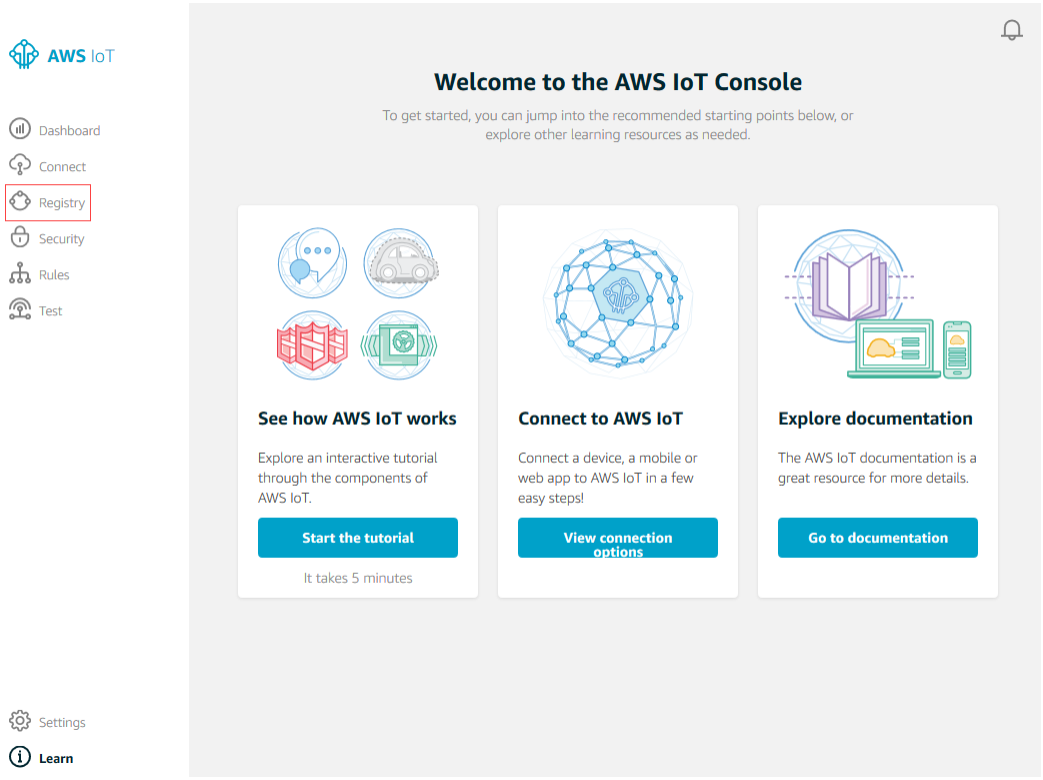
- Raspberry Pi 2 Model B
- Raspbian Wheezy
- Raspbian Jessie
- Iceweasel browser

Sign in to the AWS IoT Console

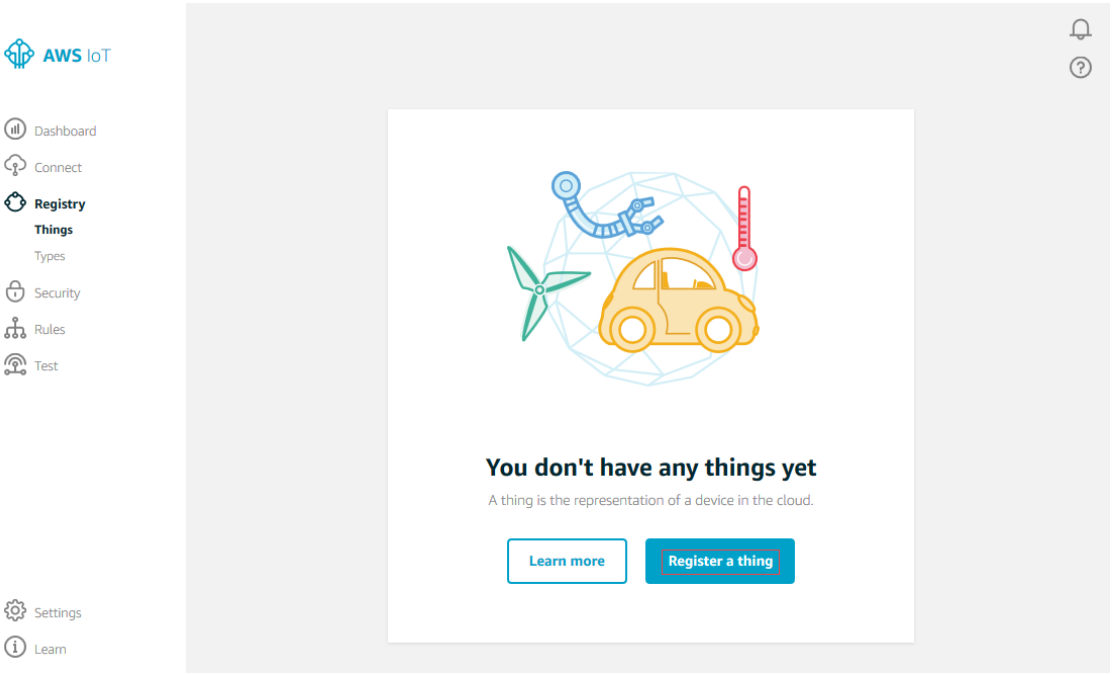
1. Turn on your Raspberry Pi and confirm you have an Internet connection.
2. Sign in to the AWS Management Console and open the AWS IoT console at https://aws.amazon.com/iot. On the **Welcome** page, choose **Get started**.



3. If this is your first time using the AWS IoT console, you will see the **Welcome to the AWS IoT Console** page. In the left navigation pane, choose **Registry** to expand the choices, and then choose **Things**.



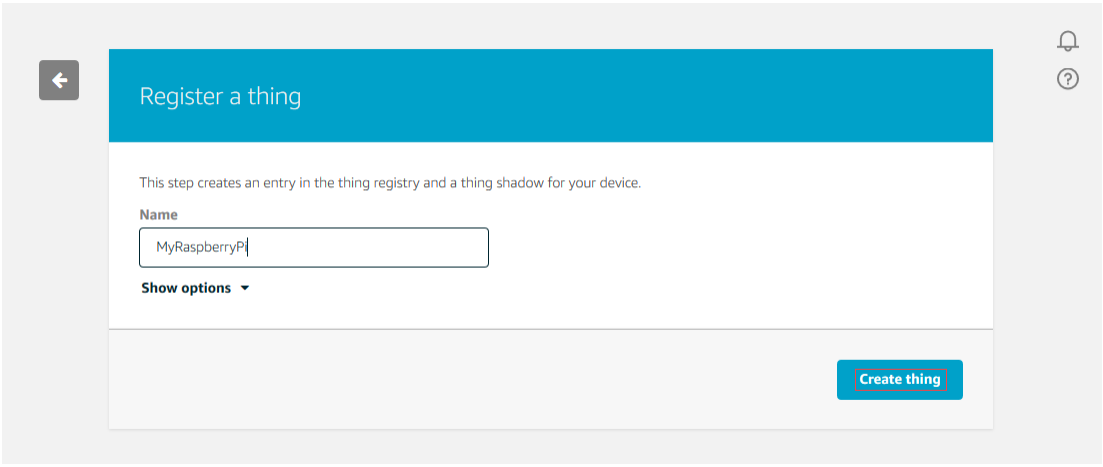
4. On the page that says **You don't have any things yet**, choose **Register a thing**. (If you have created a thing before, choose **Create**.)



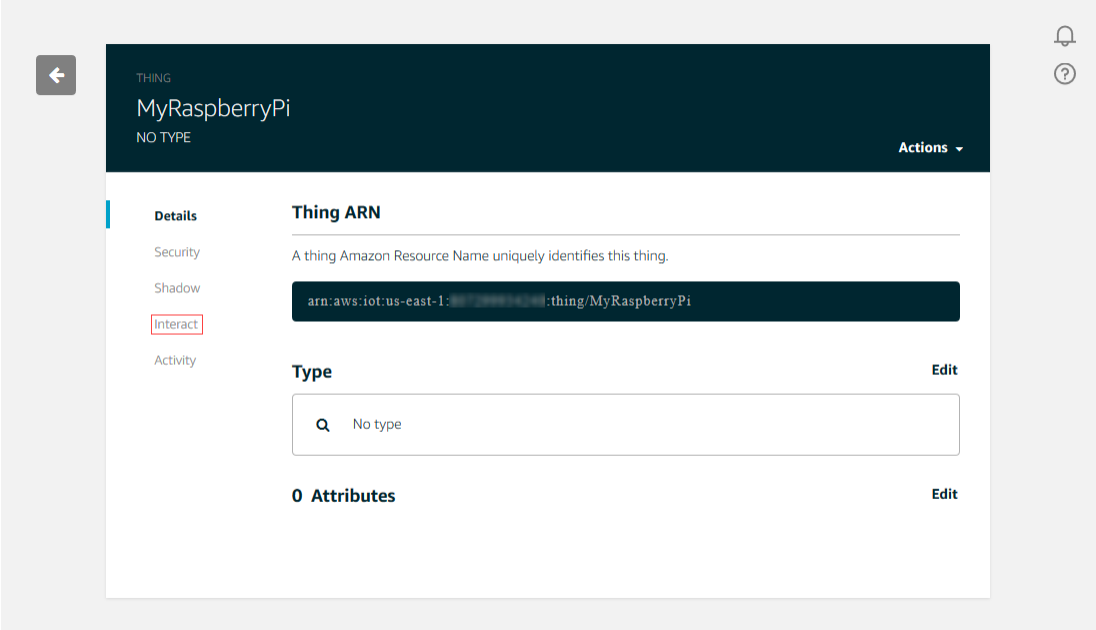
Create and Attach a Thing (Device)

A thing represents a device whose status or data is stored in the AWS cloud. The Thing Shadows service maintains a thing shadow for each device connected to AWS IoT. Thing shadows allow you to access and modify thing state data.

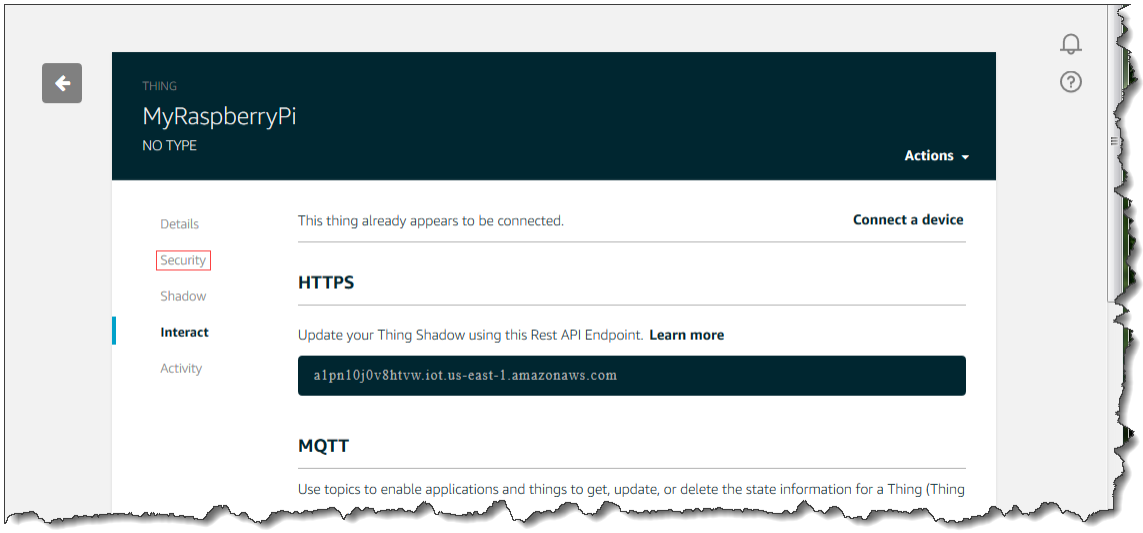
1. Type a name for the thing, and then choose **Create thing**.



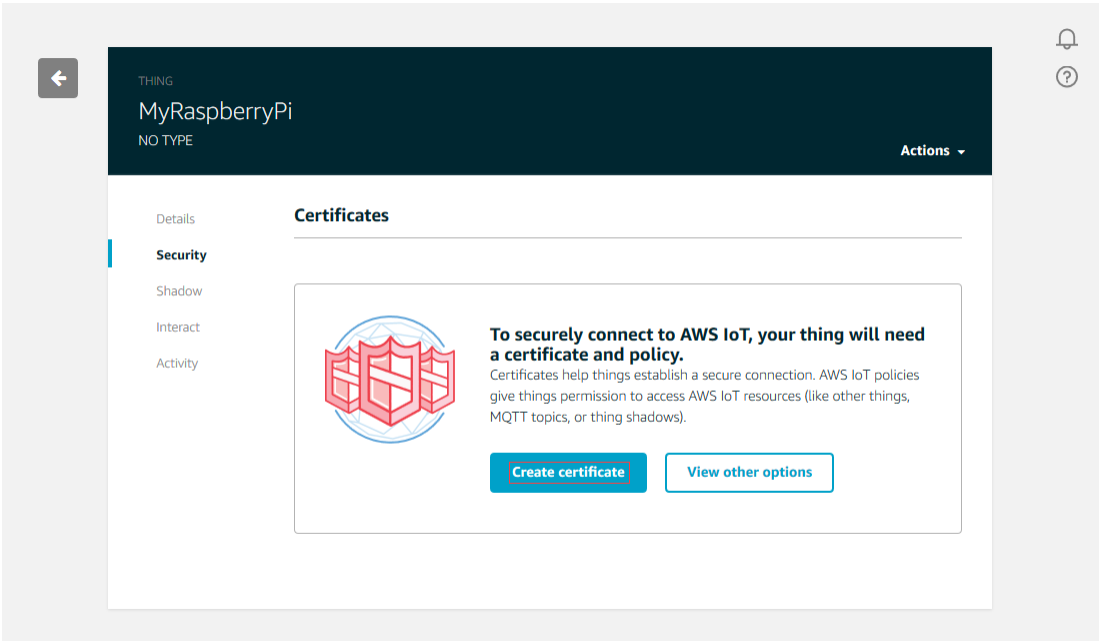
2. On the **Details** page, choose **Interact**.



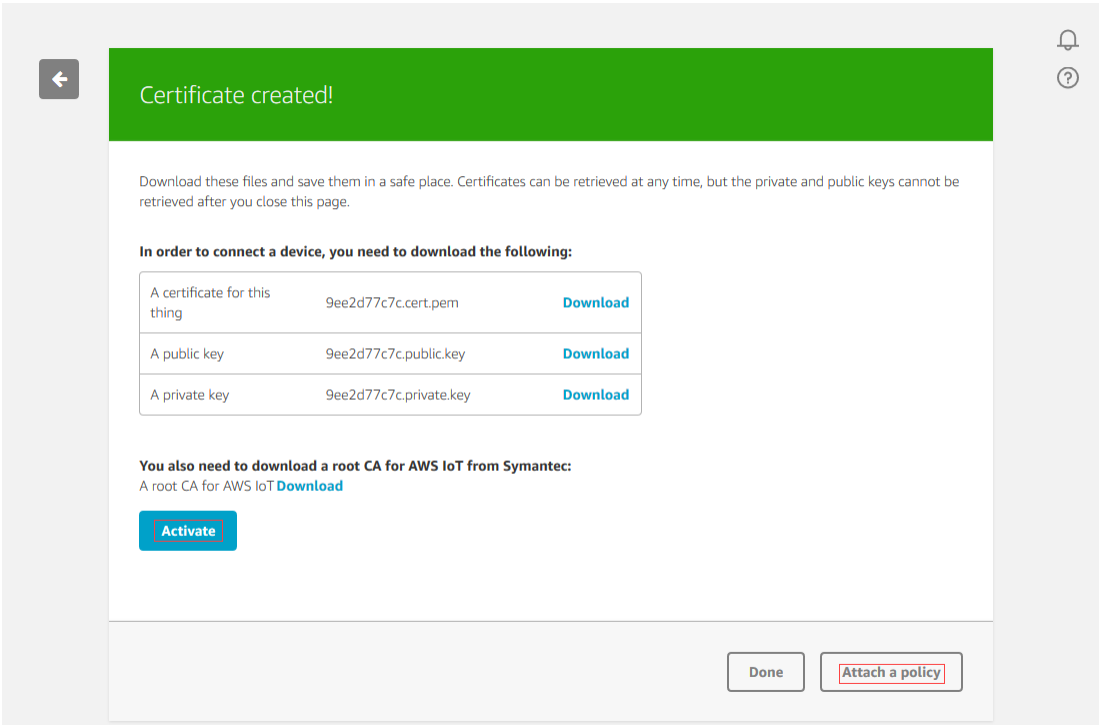
3. Make a note of the REST API endpoint. You will need this value later. Choose **Security**.



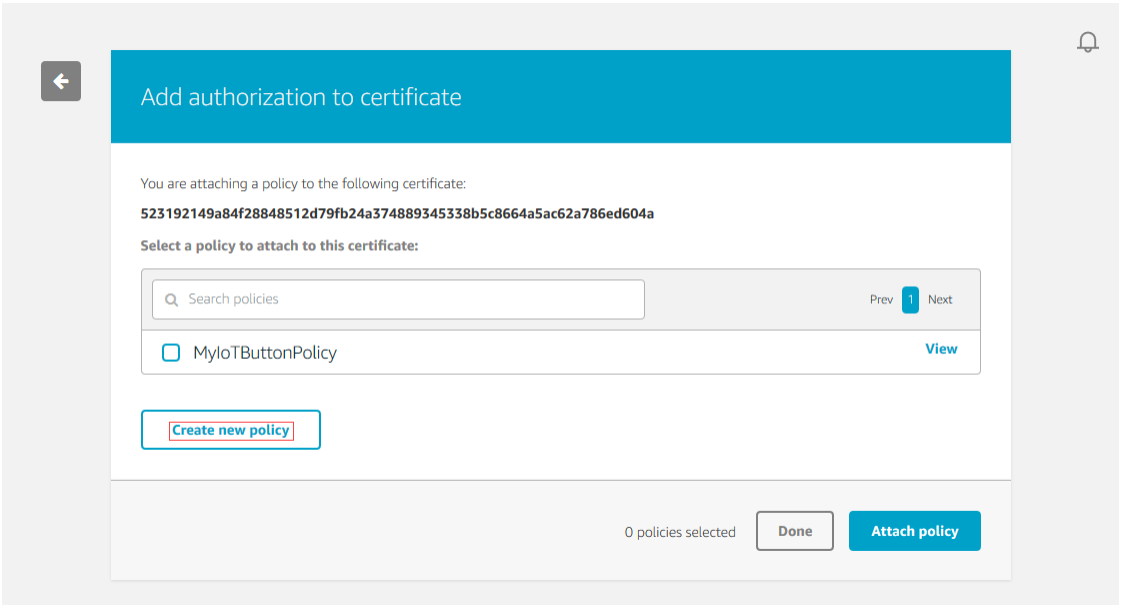
4. Choose **Create certificate**. This will generate an X.509 certificate and key pair.



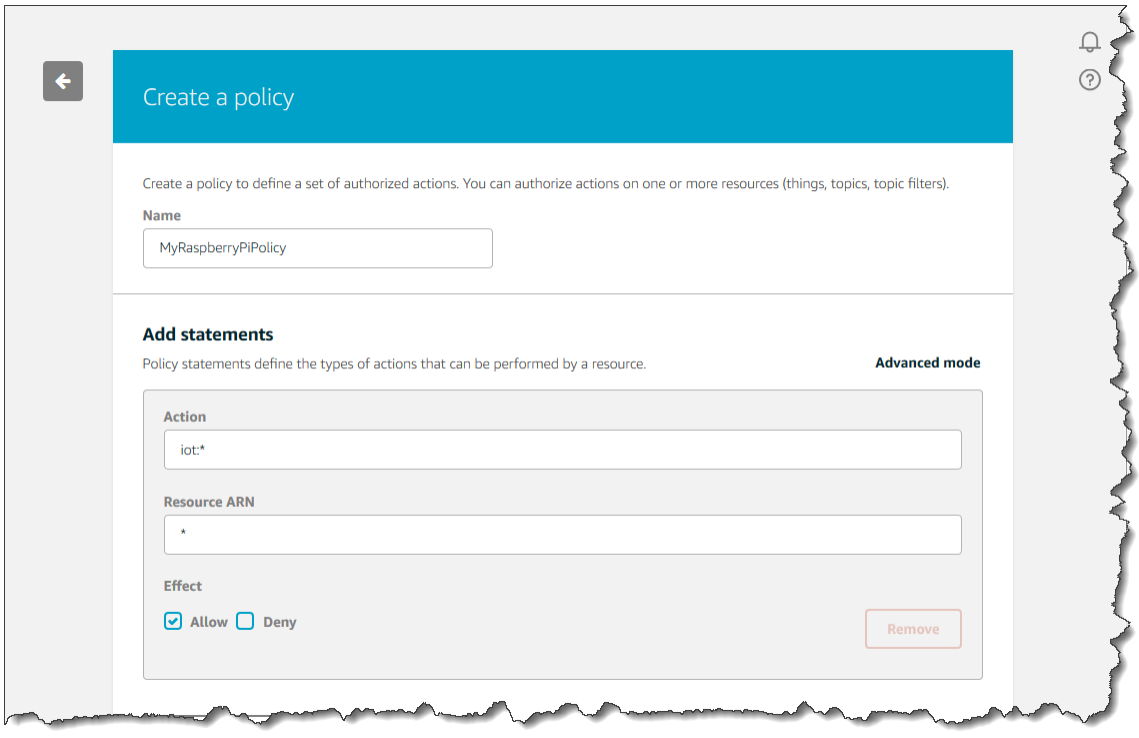
5. Create a working directory called `deviceSDK` where your files will be stored. Choose the links to download your public and private keys, certificate, and root CA and save them in the `deviceSDK` directory. Choose **Activate** to activate the X.509 certificate, then choose **Attach a policy**.



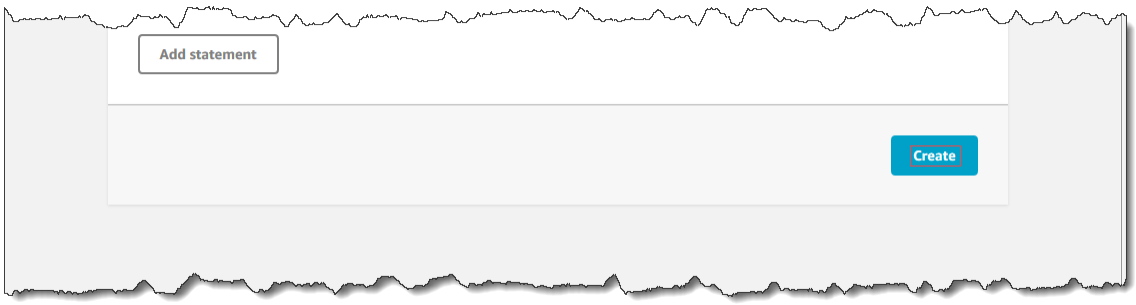
6. Choose **Create new policy**.



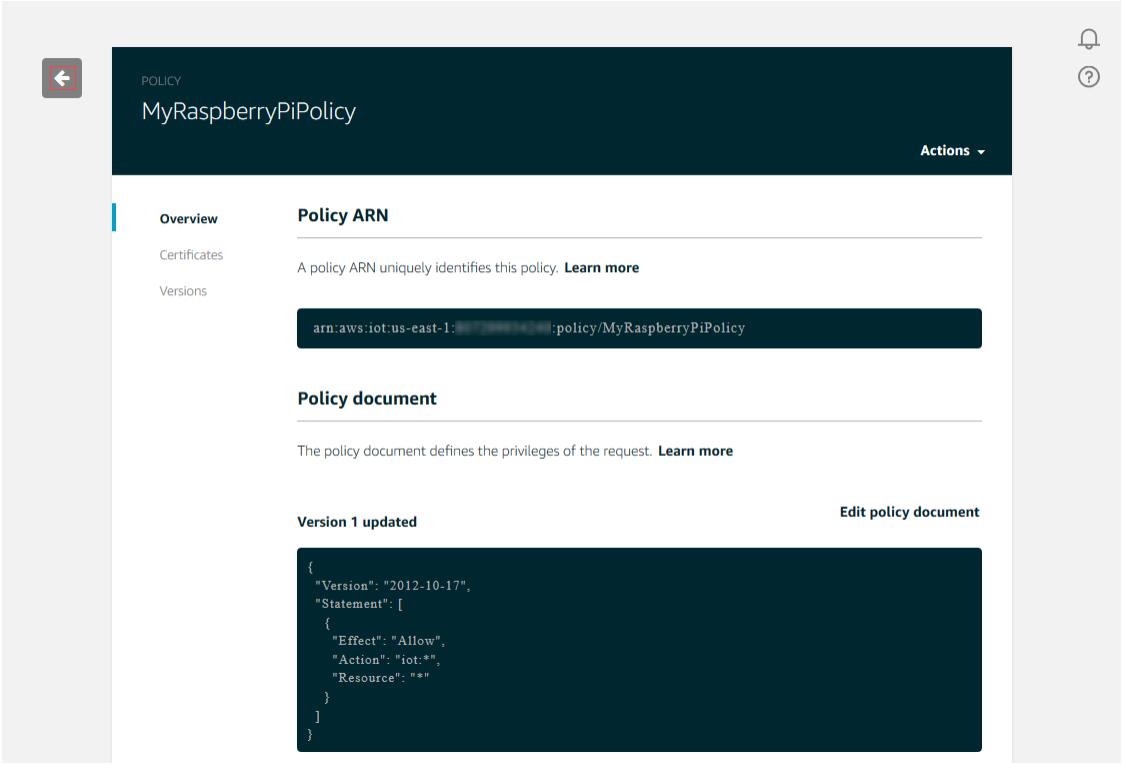
7. On the **Create a policy** page, in the **Name** field, type a name for the policy. In the **Action** field, type `iot:*`. In the **Resource ARN** field, type `*`. Select the **Allow** check box. This allows your Raspberry Pi to publish messages to AWS IoT.



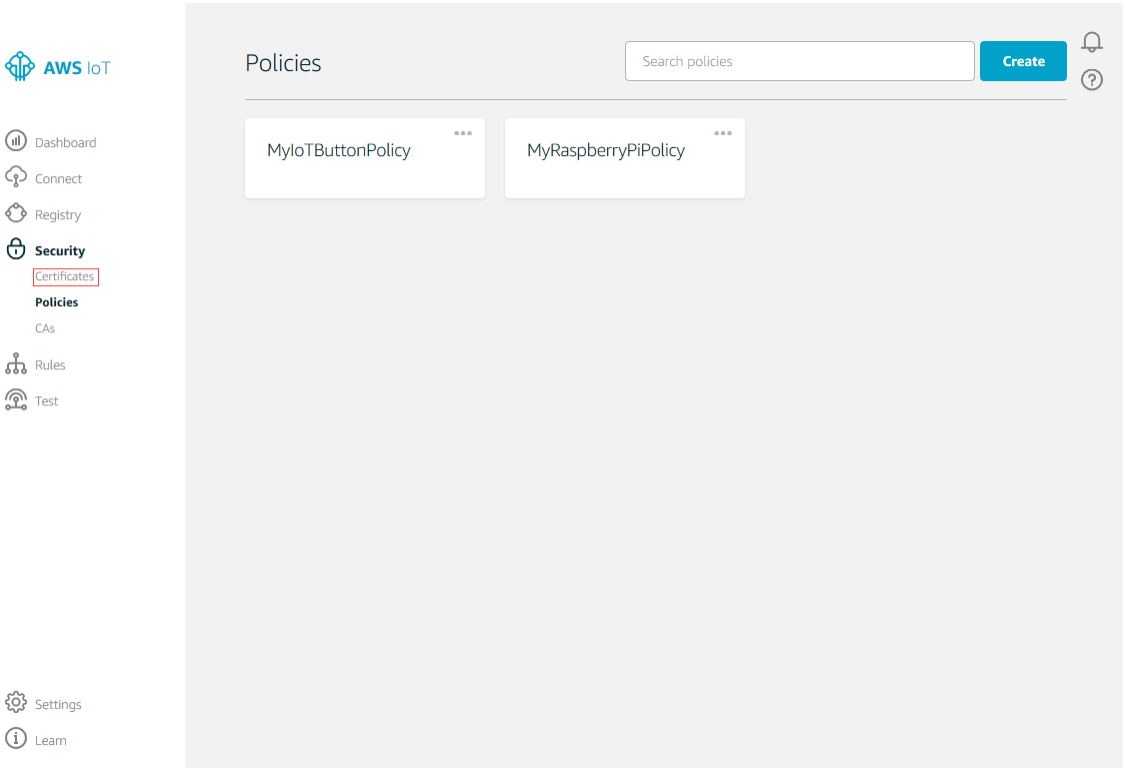
8. Choose **Create**.



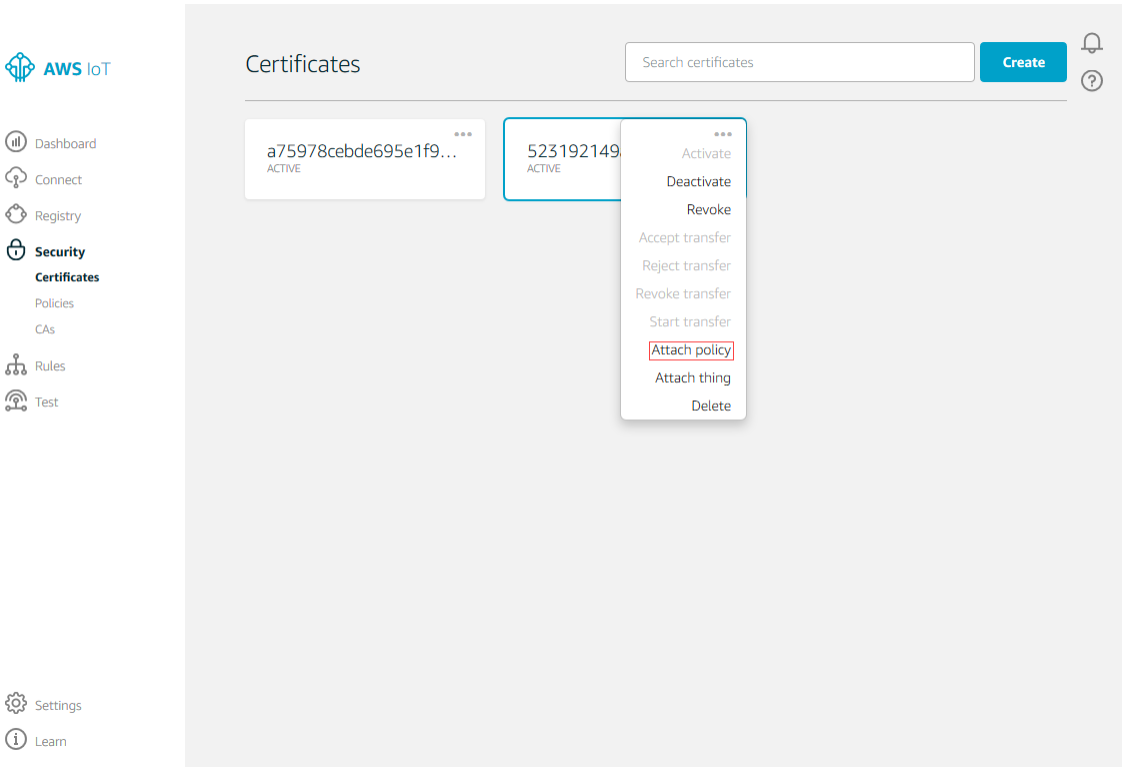
9. Choose the left arrow to return to the **Policies** page.



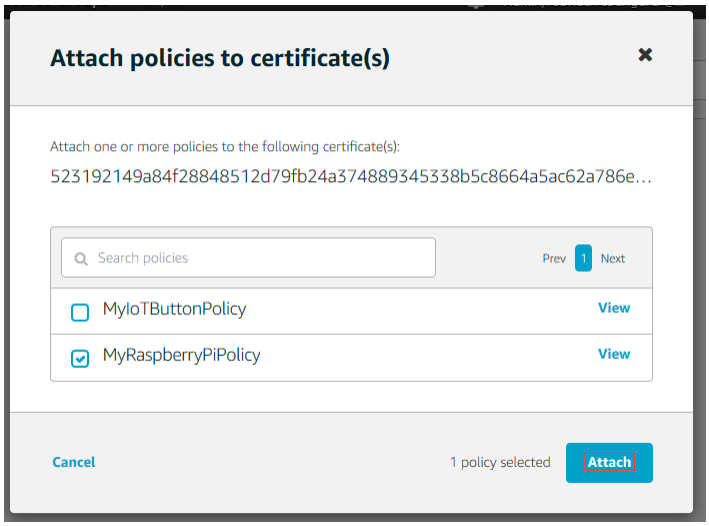
10. In the left navigation pane, under **Security**, choose **Certificates**.



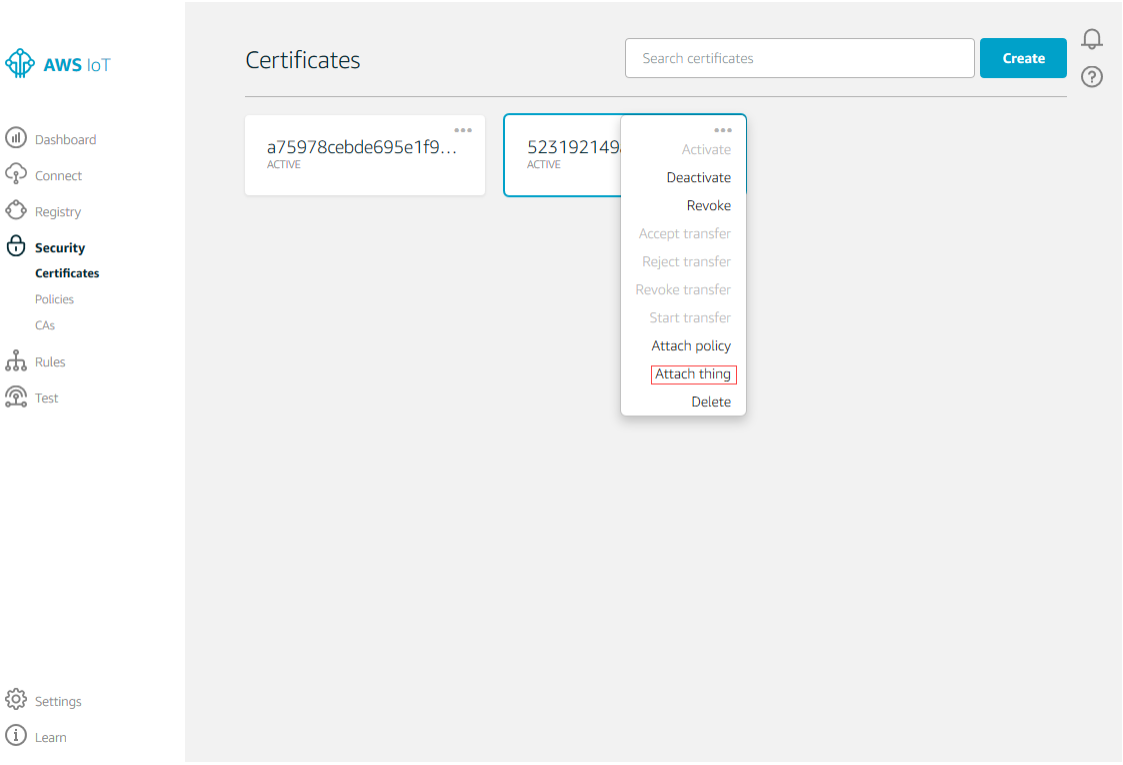
11. In the box for the certificate you created, choose **...** to open a drop-down menu, and then choose **Attach policy**.



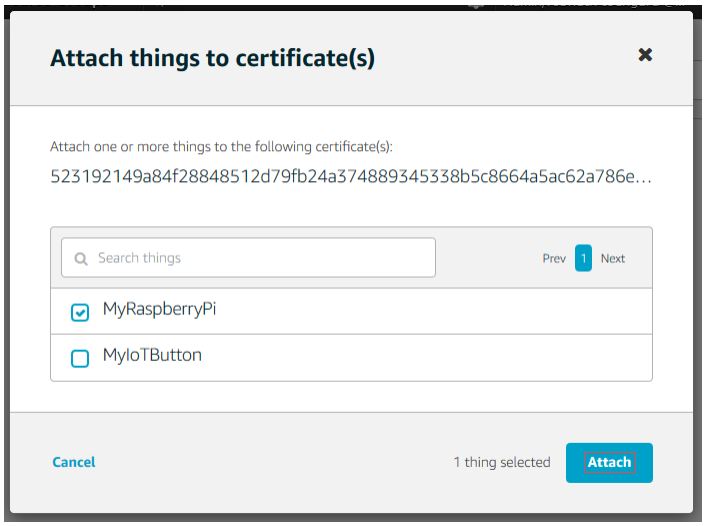
12. In the **Attach policies to certificate(s)** dialog box, select the check box next to the policy you created, and then choose **Attach**.



13. In the box for the certificate you created, choose **...** to open a drop-down menu, and then choose **Attach thing**.



14. In the **Attach things to certificate(s)** dialog box, select the check box next to the thing you created to represent your Raspberry Pi, and then choose **Attach**.



Using the AWS IoT Embedded C SDK

There are two versions of the AWS IoT Embedded C SDK: OpenSSL and mbed TLS. We will use the OpenSSL version.

Set Up the Runtime Environment for the AWS IoT Embedded C SDK

1. Download the AWS IoT Device SDK for C in a tarball (`linux_mqtt_openssl-latest.tar`). Save it in your `deviceSDK` directory.
2. In a terminal window, type the following command to extract the tarball into your `deviceSDK` directory:

   ```
   tar -xvf linux_mqtt_openssl-latest.tar
   ```
3. Before you can use the AWS IoT Embedded C SDK, you must install the OpenSSL library on Raspberry Pi. In a terminal window, run

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

Sample App Configuration

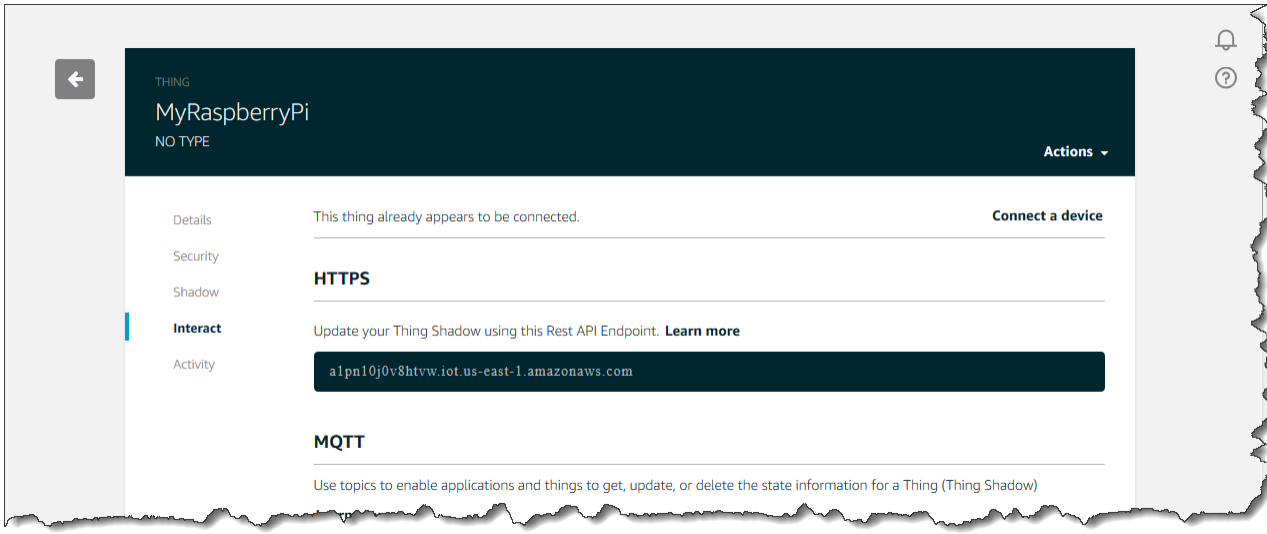
The AWS IoT Embedded C SDK includes sample apps for you to try. For simplicity, we are going to run subscribe_publish_sample.

1. Copy your certificate, private key and root CA certificate into the `deviceSDK/certs` directory.

   If you did not get a copy of the root CA certificate, you can download it here. Copy the root CA text from the browser, paste it into a file, and then copy it into the `deviceSDK/certs` directory.

   Note
   Device and root CA certificates are subject to expiration and/or revocation. If this should occur, you will need to copy new a CA certificate or a new private key and device certificate onto your device.

2. Navigate to the `deviceSDK/sample_apps/subscribe_publish_sample` directory. You will need to configure your personal endpoint, private key, and certificate. The personal endpoint is the REST API endpoint you noted earlier. If you don't remember the endpoint and you have access to a machine with the AWS CLI installed, you can use the `aws iot describe-endpoint` command to find your personal endpoint URL. Or, go to the AWS IoT console. Choose **Registry**, choose **Things**, and then choose the thing that represents your Raspberry Pi. On the **Details** page for the thing, in the left navigation pane, choose **Interact**. Copy everything, including ".com", from **REST API endpoint**.



3. Open the `aws_iot_config.h` file and, in the `//Get from console` section, update the values for the following:

   AWS_IOT_MQTT_HOST

   Your personal endpoint.

   AWS_IOT_MY_THING_NAME

   Your thing name.

   AWS_IOT_ROOT_CA_FILENAME

   Your root CA certificate.

   AWS_IOT_CERTIFICATE_FILENAME

   Your certificate.

   AWS_IOT_PRIVATE_KEY_FILENAME

   Your private key.

   For example:

```
// Get from console
// ================================================
#define AWS_IOT_MQTT_HOST              "a22j5sm6o3yzc5.iot.us-east-1.amazonaws.com"
#define AWS_IOT_MQTT_PORT              8883
#define AWS_IOT_MQTT_CLIENT_ID         "MyRaspberryPi"
#define AWS_IOT_MY_THING_NAME          "MyRaspberryPi"
#define AWS_IOT_ROOT_CA_FILENAME       "root-CA.crt"
#define AWS_IOT_CERTIFICATE_FILENAME   "4bbdc778b9-certificate.pem.crt"
#define AWS_IOT_PRIVATE_KEY_FILENAME   "4bbdc778b9-private.pem.key"
```

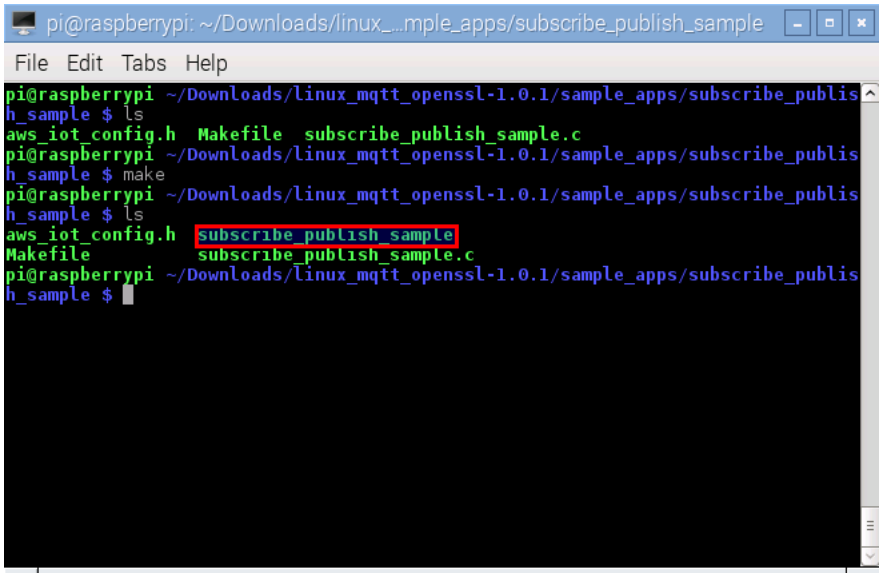
```
// ================================================
```

Run Sample Applications

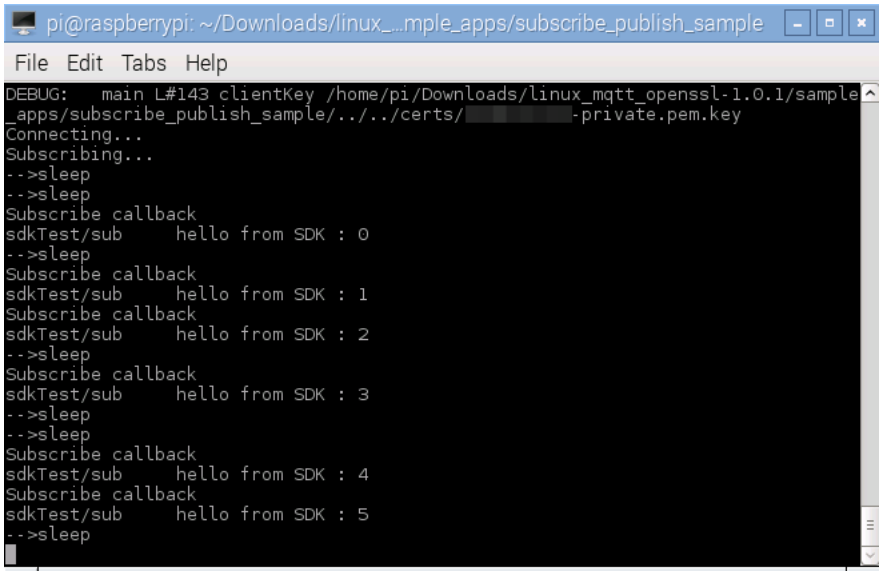
1. Compile the `subscribe_publish_sample_app` using the included makefile.

   ```
   make -f Makefile
   ```

   This will generate an executable file.



2. Now run the subscribe_publish_sample_app. You should see output similar to the following:



Your Raspberry Pi is now connected to AWS IoT using the AWS IoT Device SDK for C.

Using the AWS IoT Device SDK for JavaScript

The easiest way to install the AWS IoT Device SDK for Node.js is to use `npm`. In this section we describe how to install Node and npm.

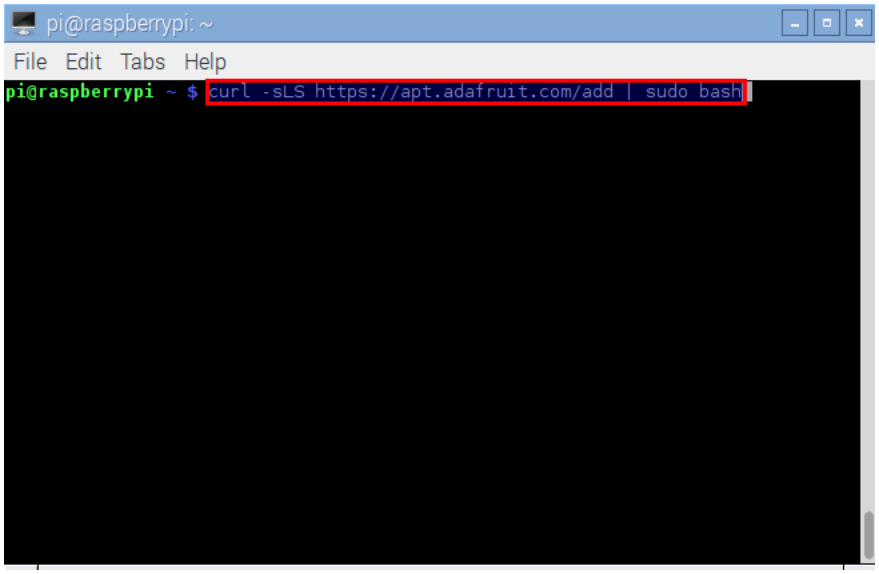
Set Up the Runtime Environment for the AWS IoT Device SDK for JavaScript

To use the AWS IoT Device SDK for JavaScript, you need to install Node and the npm development tool on your Raspberry Pi. These packages are not installed by default.

Note
Before you continue, you might want to configure the keyboard mapping for your Raspberry Pi. For more information, see Configure Raspberry Pi Keyboard Mapping.

1. To add the Node repository, open a terminal and run the following command:

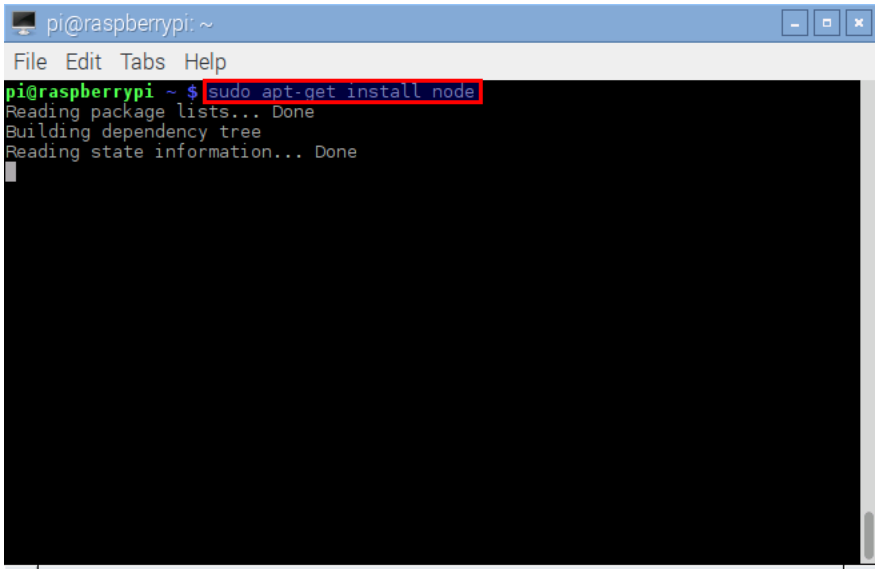
```
curl -sLS https://apt.adafruit.com/add | sudo bash
```



2. To install Node, run

```
sudo apt-get install node
```

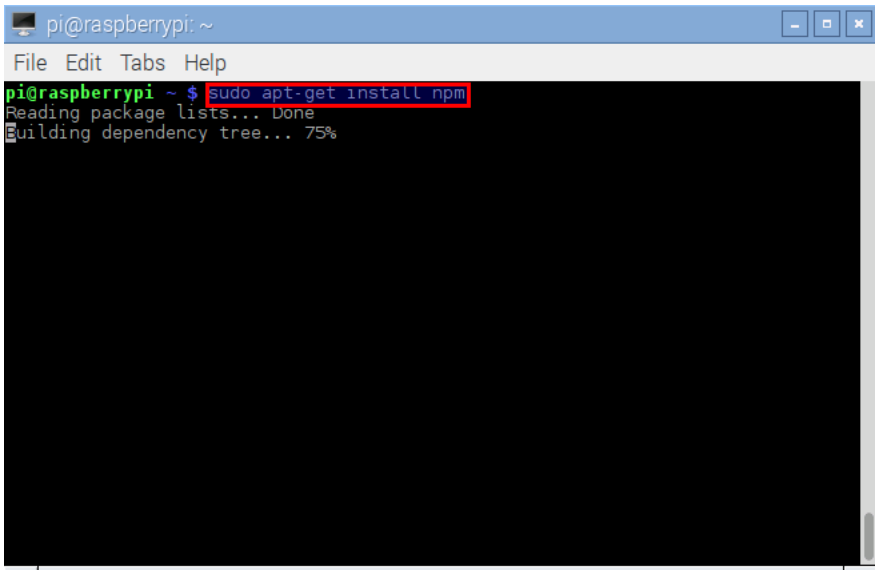
You should see output similar to the following:



3. To install npm, run

```
sudo apt-get install npm
```

You should see output similar to the following:



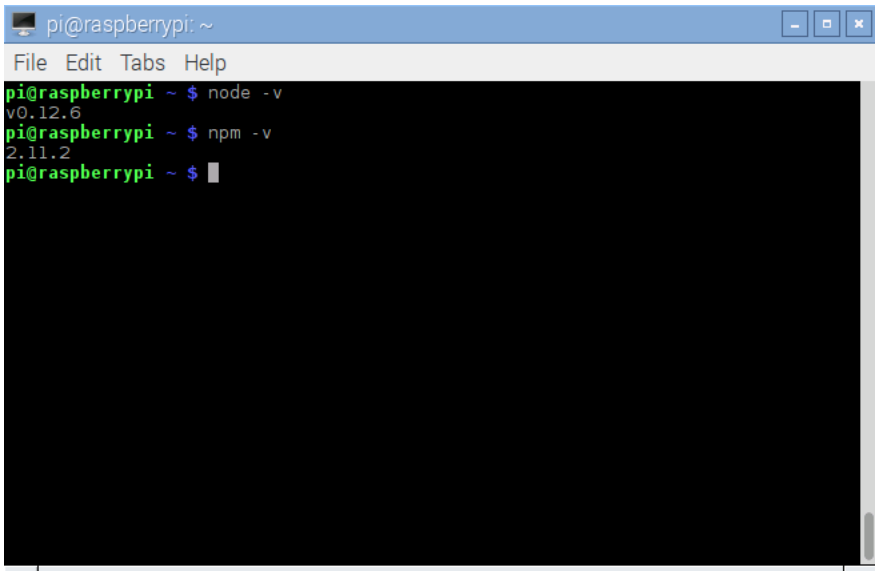
4. To verify the installation of Node and npm, run

```
node -v
```

and

```
npm -v
```

You should see output similar to the following:



Install the AWS IoT Device SDK for JavaScript

To install the AWS IoT Device SDK for JavaScript/Node.js on your Raspberry Pi, open a console window and from your `~/deviceSDK` directory, use npm to install the SDK:

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

After the installation is complete, you should find a `node_modules` directory in your `~/deviceSDK` directory.

Prepare to Run the Sample Applications

The AWS IoT Device SDK for JavaScript includes sample apps for you to try. To run them, you must configure your certificates and private key.

Edit the file ~/deviceSDK/aws-iot-device-sdk/examples/lib/cmdline.js to change the default names for the private key (`privateKey`), certificate (`clientCert`), and CA root certificate (`caCert`) used by the samples. For example:

```
default: {
    region: 'us-east-1',
    clientId: clientIdDefault,
    privateKey: '4bbdc778b9-private.pem.key',
    clientCert: '4bbdc778b9-certificate.pem.crt',
    caCert: 'root-CA.crt,
    testMode: 1,
    reconnectPeriod: 3 * 1000,  /* milliseconds */
    delay: 4 * 1000      /* milliseconds */
};
```

Run the Sample Applications

Run the examples using

```
node examples/<YourDesiredExample>.js -f <certs location>
```

Assuming you are in the directory `~/deviceSDK/node_modules/aws-iot-device-sdk/`, the certificates location should be `~/deviceSDK/certs/`. For more information about how you can use command line options to specify the certificates location and your own host address, see Certificates.

If you want to create a configuration file for use with the command line option `--configuration-file (-F)`, create a file (in JSON format) with the following properties. For example:

```
{
    "host":           "a22j5sm6o3yzc5.iot.us-east-1.amazonaws.com"
    "port":           8883
    "clientId":       "MyRaspberryPi"
    "thingName":      "MyRaspberryPi"
    "caCert":         "root-CA.crt"
    "clientCert":     "4bbdc778b9-certificate.pem.crt"
    "privateKey":     "4bbdc778b9-private.pem.key"
}
```

Your Raspberry Pi is now connected to AWS IoT using the AWS IoT SDK for JavaScript.

Managing Things with AWS IoT

AWS IoT provides a thing registry that helps you manage your 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 thing 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 thing registry and the Thing Shadows service. This provides organization and convenience to your IoT fleet without removing the flexibility of the underlying device certificate model or thing shadows.

You do not need to create a thing in the thing registry to connect it to AWS IoT. Adding your things in the thing registry allows you to manage and search for them more easily.

Managing Things with the Thing Registry

You use the AWS IoT console or the AWS CLI to interact with the registry. The following sections show how to use the CLI to work with the thing registry.

Create a thing

The following command shows how to use the AWS IoT `create-thing` CLI command to create a thing:

```
$ aws iot create-thing --thing-name "MyLightBulb" --attribute-payload "{\"attributes\":
 {\"wattage\":\"75\", \"model\":\"123\"}}"
```

The `create-thing` API will display the name and ARN of your new thing:

```
{
    "thingArn": "arn:aws:iot:us-east-1:803981987763:thing/MyLightBulb",
    "thingName": "MyLightBulb"
}
```

List things

You can use the `list-things` API 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"
        }
    ]
}
```

Search for things

You can use the `describe-thing` API to list information about a thing:

```
$ aws iot describe-thing --thing-name "MyLightBulb"
{
    "version": 3,
    "thingName": "MyLightBulb",
    "defaultClientId": "MyLightBulb",
    "thingTypeName": "StopLight",
    "attributes": {
        "model": "123",
        "wattage": "75"
    }
}
```

You can use the `list-things` API to search for all things associated with a thing type name:

```
$  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": "MySecondLightBulb"
        }
    ]
}
```

You can use the `list-things` API to search for all things that have an attribute with a specific value:

```
$  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"
        }
    ]
}
```

Update a thing

You can use the `update-thing` API to update a thing:

```
$ aws iot update-thing --thing-name "MyLightBulb" --attribute-payload "{\"attributes\":
 {\"wattage\":\"150\", \"model\":\"456\"}}"
```

The `update-thing` command does not produce output. You can use the `describe-thing` API 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 `delete-thing` API to delete a thing:

```
$ aws iot delete-thing --thing-name "MyThing"
```

Attach a principal to a thing

A physical device must have an X.509 certificate in order to communicate with AWS IoT. You can associate the certificate on your device with the thing in the thing registry that represents your device. To attach a certificate to your thing, use the `attach-thing-principal` API:

```
$ aws iot attach-thing-principal --thing-name "MyLightBulb" --principal "arn:aws:iot:us-
east-1:123456789012:cert/a0c01f5835079de0a7514643d68ef8414ab739a1e94ee4162977b02b12842847"
```

The `attach-thing-principal` command does not produce any output.

Detach a principal from a thing

You can use the `detach-thing-principal` API 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 `detach-thing-principal` 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 thing 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 provides better discovery of things.

- Things can have up to 50 attributes.
- Things without a thing type can have up to three attributes.
- A thing can only be associated with 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 `create-thing-type` API 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 `create-thing-type` command returns a response that contains the thing type and its ARN:

```
{
    "thingTypeName": "LightBulb",
    "thingTypeArn": "arn:aws:iot:us-west-2:803981987763:thingtype/LightBulb"
}
```

List thing types

You can use the `list-thing-types` API to list thing types:

```
$ aws iot list-thing-types
```

The `list-thing-types` command returns a list of the thing types defined in your AWS account:

```
{
    "thingTypes": [
        {
            "thingTypeName": "LightBulb",
            "thingTypeProperties": {
                "deprecated": false,
                "creationDate": 1468423800950,
                "searchableAttributes": [
                    "wattage",
                    "model"
                ],
                "thingTypeDescription": "light bulb type"
            }
        }
    ]
}
```

Describe a thing type

You can use the `describe-thing-type` API to get information about a thing type:

```
$ aws iot describe-thing-type --thing-type-name "LightBulb"
```

The `describe-thing-type` API responds with information about the specified type:

```
{
    "thingTypeName": "LightBulb",
    "thingTypeProperties": {
        "deprecated": false,
        "creationDate": 1468423800950,
        "searchableAttributes": [
            "wattage",
            "model"
        ],
        "thingTypeDescription": "light bulb type"
    }
}
```

Associate a thing type with a thing

You can use the `create-thing` API to specify a thing type when you create a thing:

```
$ aws iot create-thing --thing-name "MySecondLightBulb" --thing-type-name "LightBulb" --
attribute-payload "{\"attributes\": {\"wattage\":\"75\", \"model\":\"123\"}}"
```

You can use the `update-thing` API at any time to change the thing type associated with a thing:

```
$ aws iot update-thing --thing-name "MyLightBulb" --thing-type-name "StopLight" --
attribute-payload  "{\"attributes\": {\"wattage\":\"75\", \"model\":\"123\"}}"
```

You can also use the `update-thing` API 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 will be unchanged.

To deprecate a thing type, use the `deprecate-thing-type` API:

```
$ aws iot deprecate-thing-type --thing-type-name "myThingType"
```

You can use the `describe-thing-type` API to see the result:

```
$ aws iot describe-thing --thing-type-name "StopLight":
```

```
{
    "thingTypeName": "StopLight",
    "thingTypeProperties": {
        "deprecated": true,
        "creationDate": 1468425854308,
        "searchableAttributes": [
            "wattage",
            "numOfLights",
            "model"
        ],
        "thingTypeDescription": "traffic light type",
        "deprecationDate": 1468446026349
    }
```

```
}
```

Deprecating a thing type is a reversible operation. You can undo a deprecation by using the `--undo-deprecate` flag with the `deprecate-thing-type` CLI command:

```
$ aws iot deprecate-thing-type --thing-type-name "myThingType" --undo-deprecate
```

You can use the `deprecate-thing-type` CLI command to see the result:

```
$ aws iot deprecate-thing-type --thing-type-name "StopLight":
```

```
{
    "thingTypeName": "StopLight",
    "thingTypeProperties": {
        "deprecated": false,
        "creationDate": 1468425854308,
        "searchableAttributes": [
            "wattage",
            "numOfLights",
            "model"
        ],
        "thingTypeDescription": "traffic light type"
    }
}
```

Delete a thing type

You can delete thing types only after they have been deprecated. To delete a thing type, use the `delete-thing-type` API:

```
$ 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.

Security and Identity for AWS IoT

Each connected device must have a credential to access the message broker or the Thing Shadows service. All traffic to and from AWS IoT must be encrypted over Transport Layer Security (TLS). Device credentials must be kept safe in order to send data securely to the message broker. AWS cloud security mechanisms protect data as it moves between AWS IoT and other devices or AWS services.

- You are responsible for managing device credentials (X.509 certificates, AWS credentials) on your devices and policies in AWS IoT. You are responsible for assigning unique identities to each device and managing the permissions for a device or group of devices.
- Devices connect using your choice of identity (X.509 certificates, IAM users and groups, or Amazon Cognito identities) over a secure connection according to the AWS IoT connection model.
- The AWS IoT message broker authenticates and authorizes all actions in your account. The message broker is responsible for authenticating your devices, securely ingesting device data, and adhering to the access permissions you place on devices using policies.
- The AWS IoT rules engine forwards device data to other devices and other AWS services according to rules you define. It uses AWS access management systems to securely transfer data to its final destination.

Authentication in AWS IoT

AWS IoT supports four types of identity principals for authentication:

- X.509 certificates
- IAM users, groups, and roles
- Amazon Cognito identities
- Federated identities

These identities can be used with mobile applications, web applications, or desktop applications. They can even be used by a user typing AWS IoT 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. CLI commands use IAM.

X.509 Certificates

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.

> **Note**
> We recommend that each device be given a unique certificate to enable fine-grained management including certificate revocation.

> **Note**
> Devices must support rotation and replacement of certificates in order to ensure smooth operation as certificates expire.

AWS IoT supports the following certificate-signing algorithms:

- SHA256WITHRSA
- SHA384WITHRSA
- SHA384WITHRSA
- SHA512WITHRSA
- RSASSAPSS
- DSA_WITH_SHA256
- ECDSA-WITH-SHA256
- ECDSA-WITH-SHA384
- ECDSA-WITH-SHA512

Certificates provide several benefits over other identification and authentication mechanisms. Certificates enable asymmetric keys to be used with devices. This means you can burn private keys into secure storage on a device. This way, sensitive cryptographic material never leaves the device. Certificates provide stronger client authentication over other schemes, such as user name and password or bearer tokens, because the secret key never leaves the device.

AWS IoT authenticates certificates using the TLS protocol's client authentication mode. TLS is available in many programming languages and operating systems and is commonly used for encrypting data. In TLS client authentication, AWS IoT requests a client X.509 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.

To use AWS IoT certificates, clients must support all of the following in their TLS implementation:

- TLS 1.2.
- SHA-256 RSA certificate signature validation.

- One of the cipher suites from the TLS cipher suite support section.

X.509 Certificates and AWS IoT

AWS IoT can use AWS IoT-generated certificates or certificates signed by a CA certificate for device authentication. Certificates generated by AWS IoT do not expire. The expiry date and time for certificates signed by a CA certificate are set when the certificate is created.

> **Note**
> We recommend that each device be given a unique certificate to enable fine-grained management including certificate revocation.

> **Note**
> Devices must support rotation and replacement of certificates in order to ensure smooth operation as certificates expire.

To use a certificate that is not created by AWS IoT, you must register a CA certificate. All device certificates must be signed by the CA certificate you register.

You can use the AWS IoT console or CLI to perform the following operations:

- Create and register an AWS IoT certificate.
- Register a CA certificate.
- Register a device certificate.
- Activate or deactivate a device certificate.
- Revoke a device certificate.
- Transfer a device certificate to another AWS account.
- List all CA certificates registered to your AWS account.
- List all device certificates registered to your AWS account.

For more information about the CLI commands to use to perform these operations, see AWS IoT CLI Reference.

For more information about using the AWS IoT console to create certificates, see Create and Activate a Device Certificate.

Server Authentication

Device certificates allow AWS IoT to authenticate devices. To make sure your device is communicating with AWS IoT and not another server impersonating AWS IoT, copy the VeriSign Class 3 Public Primary G5 root CA certificate onto your device.

> **Note**
> This CA certificate is valid until July 2036, but the CA certificate may need to be replaced before then. You should ensure that you can update the root CA certificate on all of your devices to ensure ongoing connectivity and to keep up to date with security best practices.

Reference the CA root certificate in your device code when you connect to AWS IoT. For more information, see the AWS IoT Device SDKs (p. 235).

> **Note**
> You cannot use your own CA certificate to authenticate the AWS IoT server. You must use the VeriSign Class 3 Public Primary G5 root CA certificate.

Create and Register an AWS IoT Device Certificate

You can use the AWS IoT console or the AWS IoT CLI to create an AWS IoT certificate.

To create a certificate (console)

1. Sign in to the AWS Management Console and open the AWS IoT console at https://console.aws.amazon.com/iot.
2. In the left navigation pane, choose **Security** to expand the choices, and then choose **Certicates**. Choose **Create**.
3. Choose **One-click certificate creation** - **Create certificate**. Alternatively, to generate a certificate with a certificate signing request (CSR), choose **Create with CSR**.
4. Use the links to the public key, private key, and certificate to download each to a secure location.
5. Choose **Activate**.

To create a certificate (CLI)

The AWS IoT CLI provides two commands to create certificates:

- create-keys-and-certificate

  The CreateKeysAndCertificate API creates a private key, public key, and X.509 certificate.
- create-certificate-from-csr

  The CreateCertificateFromCSR API creates a certificate given a CSR.

Use Your Own Certificate

To use your own X.509 certificates, you must register a CA certificate with AWS IoT. The CA certificate can then be used to sign device certificates. You can register up to 10 CA certificates with the same subject field per AWS account per region. This allows you to have more than one CA sign your device certificates.

> **Note**
> Device certificates must be signed by the registered CA certificate. It is common for a CA certificate to be used to create an intermediate CA certificate. If you are using an intermediate certificate to sign your device certificates, you must register the intermediate CA certificate. Use the AWS IoT root CA certificate when you connect to AWS IoT even if you register your own root CA certificate. The AWS IoT root CA certificate is used by a device to verify the identity of the AWS IoT servers.

Contents

If you do not have a CA certificate, you can use OpenSSL tools to create one.

To create a CA certificate

1. Generate a key pair.

```
openssl genrsa -out rootCA.key 2048
```

2. Use the private key from the key pair to generate a CA certificate.

```
openssl req -x509 -new -nodes -key rootCA.key -sha256 -days 1024 -out rootCA.pem
```

Registering Your CA Certificate

To register your CA certificate, you must:

- Get a registration code from AWS IoT.
- Sign a private key verification certificate with your CA certificate.
- Pass your CA certificate and a private key verification certificate to the `register-ca-certificate` CLI command.

  The `Common Name` field in the private key verification certificate must be set to the registration code generated by the `get-registration-code` CLI command. A single registration code is generated per AWS account. You can use the `register-ca-certificate` command or the AWS IoT console to register CA certificates.

To register a CA certificate

1. Get a registration code from AWS IoT. This code will be used 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 verificationCert.key 2048
```

3. Create a CSR for the private key verification certificate. Set the `Common Name` field of the certificate to your registration code.

```
openssl req -new -key verificationCert.key -out verificationCert.csr
```

   You will be prompted for some information, including the `Common Name`, for the certificate.

```
Country Name (2 letter code) [AU]:
State or Province Name (full name) []:
Locality Name (eg, city) []:
Organization Name (eg, company) []:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:XXXXXXXXXXXXMYREGISTRATIONCODEXXXXXX
Email Address []:
```

4. Use the CSR to create a private key verification certificate.

```
openssl x509 -req -in verificationCert.csr -CA rootCA.pem -CAkey rootCA.key -
CAcreateserial -out verificationCert.pem -days 500 -sha256
```

5. Register the CA certificate with AWS IoT. Pass in the CA certificate and the private key verification certificate to the `register-ca-certificate` CLI command.

```
aws iot register-ca-certificate --ca-certificate file://rootCA.pem --verification-cert
 file://verificationCert.pem
```

6. Use the `update-certificate` CLI command to activate the CA certificate .

```
aws iot update-ca-certificate --certificate-id xxxxxxxxxxx --new-status ACTIVE
```

Creating a Device Certificate Using Your CA Certificate

You can use a CA certificate registered with AWS IoT to create a device certificate. The device certificate must be registered with AWS IoT before use.

To create a device certificate

1. Generate a key pair.

```
openssl genrsa -out deviceCert.key 2048
```

2. Create a CSR for the device certificate.

```
openssl req -new -key deviceCert.key -out deviceCert.csr
```

You will be prompted for some information, as shown here.

```
Country Name (2 letter code) [AU]:
State or Province Name (full name) []:
Locality Name (eg, city) []:
Organization Name (eg, company) []:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:
Email Address []:
```

3. Create a device certificate from the CSR.

```
openssl x509 -req -in deviceCert.csr -CA rootCA.pem -CAkey rootCA.key -CAcreateserial -
out deviceCert.pem -days 500 -sha256
```

Note
You must use the CA certificate registered with AWS IoT to create device certificates. If you have more than one CA certificate (with the same subject field and public key) registered in your AWS account, you must specify the CA certificate used to create the device certificate when you register your device certificate.

4. Register a device certificate.

```
aws iot register-certificate --certificate-pem file://deviceCert.pem --ca-certificate-
pem file://caCert.crt
```

5. Use the `update-certificate` CLI command to activate the device certificate .

```
aws iot update-certificate --certificate-id xxxxxxxxxxx --new-status ACTIVE
```

Registering a Device Certificate

You must use the CA certificate registered with AWS IoT to sign device certificates. If you have more than one CA certificate (with the same subject field and public key) registered in your AWS account, you must specify the CA certificate used to sign the device certificate when you register your device certificate. You

can register each device certificate manually, or you can use automatic registration, which allows devices to register their certificate when they connect to AWS IoT for the first time.

Registering Device Certificates Manually

Use the following CLI command to register a device certificate:

```
aws iot register-certificate --certificate-pem file://deviceCert.crt --ca-certificate-pem
 file://caCert.crt
```

Using Automatic/Just-in-Time Registration for Device Certificates

To register device certificates automatically when devices first connect to AWS IoT, you must enable automatic registration for your CA certificate. This will register any device certificate signed by your CA certificate when it connects to AWS IoT.

Enable Automatic Registration

Use the `update-ca-certificate` API to set the `auto-registration-status` of the CA certificate to `ENABLE`:

```
$ aws iot update-ca-certificate --cert-id caCertificateId --new-auto-registration-status
 ENABLE
```

You can also set the `auto-registration-status` to `ENABLE` when you use the `register-ca-certificate` API to register your CA certificate:

```
aws iot register-ca-certificate --ca-certificate file://rootCA.pem --verification-cert
 file://privateKeyVerificationCert.crt --allow-auto-registration
```

When a device first attempts to connect to AWS IoT, as part of the TLS handshake, it must present a registered CA certificate and a device certificate. AWS IoT recognizes the CA certificate as a registered CA certificate and automatically registers the device certificate and sets its status to `PENDING_ACTIVATION`. This means the device certificate was automatically registered and is awaiting activation. A certificate must be in the ACTIVE state before it can be used to connect to AWS IoT. When AWS IoT automatically registers a certificate or when a certificate in PENDING_ACTIVATION status connects, 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 device 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 device 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 device is able to 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.

Deactivate the CA Certificate

When you register a device certificate, AWS will check if the associated CA certificate is `ACTIVE`. If the CA certificate is `INACTIVE`, AWS IoT does not allow the device certificate to be registered. By marking the CA certificate as INACTIVE, you prevent any new device certificates issued by the compromised CA to be registered in your account. You can use the `update-ca-certificate` API to deactivate the CA certificate:

```
$ aws iot update-ca-certificate --cert-id certificateId --new-status INACTIVE
```

Note
Any registered device certificates that were signed by the compromised CA certificate will continue to work until you explicitly revoke them.

Use the `ListCertificatesByCA` API to get a list of all registered device certificates that were signed by the compromised CA. For each device certificate signed by the compromised CA certificate, use the `UpdateCertificate` API to revoke the device certificate to prevent it from being used.

Revoke the Device Certificate

If you detect suspicious activity on a registered device certificate, you can use the `update-certificate` API to revoke it:

```
$ aws iot update-certificate --cert-id certificateId
                             --new-status REVOKED
```

If any error or exception occurs during the auto-registration of the device 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.

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 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.

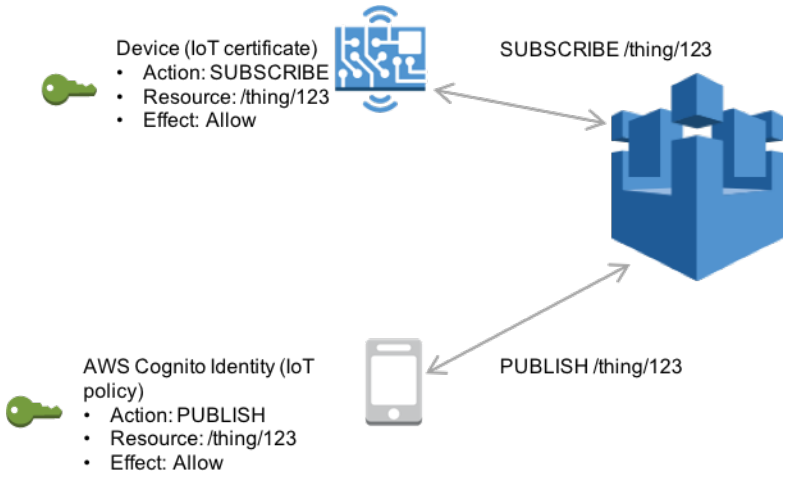
For information, see the IAM User Guide.

Amazon Cognito Identities

Amazon Cognito Identity allows you to use your own identity provider or other popular identity providers, such as Login with Amazon, Facebook, or Google. You exchange a token from your identity

provider for AWS security credentials. The credentials represent an IAM role and can be used with AWS IoT.

AWS IoT extends Amazon Cognito and allows policy attachment to Amazon Cognito identities. You can attach a policy to an Amazon Cognito identity and give fine-grained permissions to an individual user of your AWS IoT application. In this way, you can assign permissions between specific customers and their devices. For more information, see Amazon Cognito Identity.



Authorization

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 CLI commands. The identity can execute AWS IoT operations only if it has a policy that grants it permission.

Both AWS IoT policies and IAM policies are used with AWS IoT 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. The following table shows the identity types, the protocols they use, and the policy types that can be used for authorization.

AWS IoT 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.

The type of policy you use depends on whether you are using control plane or data plane API.

AWS IoT Data Plane API and Policy Types

Protocol and Authentication Mechanism	SDK	Identity Type	Policy Type		
MQTT over mutual authentication (port 8883)	AWS IoT Device SDK	X.509 certificates	AWS IoT policy		

Protocol and Authentication Mechanism	SDK	Identity Type	Policy Type		
MQTT over Websocket (port 443)	AWS Mobile SDK	Amazon Cognito, IAM, or federated identity	AWS IoT policy for Amazon Cognito identities IAM policy for other identities		
HTTP over server authentication (port 443)	AWS CLI	Amazon Cognito, IAM, or federated identity	AWS IoT policy for Amazon Cognito identities IAM policy for other identities		
HTTP over mutual authentication (port 8443)	No SDK support	X.509 certificates	AWS IoT policy		

AWS IoT Control Plane API and Policy Types

Protocol and Authentication Mechanism	SDK	Identity Type	Policy Type		
HTTP over server authentication (port 443)	AWS CLI	Amazon Cognito, IAM, or federated identity	AWS IoT policy for Amazon Cognito identities IAM policy for other identities		

AWS IoT 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 CLI to attach the policy (to a certificate or Amazon Cognito Identity), you use an AWS IoT 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. For example, consider a device connecting to AWS IoT 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 resource for the user. You can also control an application's access to AWS IoT resources.

AWS IoT Policies

AWS IoT policies are JSON documents. They follow the same conventions as IAM policies. AWS IoT 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 defines a set of policy actions that describe the operations and resources to which you can grant or deny access. For example:

- `iot:Connect` represents permission to connect to the AWS IoT message broker.
- `iot:Subscribe` represents permission to subscribe to an MQTT topic or topic filter.
- `iot:GetThingShadow` represents permission to get a thing shadow.

AWS IoT policies allow you to control access to the AWS IoT data plane. The AWS IoT data plane consists of operations that allow you to connect to the AWS IoT message broker, send and receive MQTT messages, and get or update thing shadows. For more information, see AWS IoT Policy Actions (p. 104).

An AWS IoT policy is a JSON document that contains one or more policy statements. Each statement contains an `Effect`, an `Action`, and a `Resource`. The `Effect` specifies whether the action will be allowed or denied. The `Action` specifies the action the policy is allowing or denying. The `Resource` specifies the resource or resources on which the action is allowed or denied. The following policy grants all devices permission to connect to the AWS IoT message broker, but restricts the device to publishing on a specific MQTT topic:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action":["iot:Publish"],
        "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/foo/bar"]
    },
    {
        "Effect": "Allow",
        "Action": ["iot:Connect"],
        "Resource": ["*"]
        }]
}
```

AWS IoT Policy Actions

The following policy actions are defined by AWS IoT:

MQTT Policy Actions

iot:Connect

Represents the permission to connect to the AWS IoT message broker. The `iot:Connect` permission is checked every time a `CONNECT` request is sent to the broker. The message broker does not allow two clients with the same client ID to stay connected at the same time. After the second client connects, the broker detects this case and disconnects one of the clients. The `iot:Connect` permission can be used to ensure only authorized clients can connect using a specific client ID.

iot:Publish

Represents the permission to publish on an MQTT topic. This permission is checked every time a PUBLISH request is sent to the broker. This can be used to allow clients to publish to specific topic patterns.

Note
You must also grant `iot:Connect` permission to grant `iot:Publish` permission.

iot:Receive

Represents the permission to receive a message from AWS IoT The `iot:Receive` permission is checked every time a message is delivered to a client. Because this permission is checked on every delivery, it can be used to revoke permissions to clients that are currently subscribed to a topic.

iot:Subscribe

Represents the permission to subscribe to an MQTT topic or topic filter. This permission is checked every time a SUBSCRIBE request is sent to the broker. This can be used to allow clients to subscribe to topics that match specific topic patterns.

Note
You must also grant `iot:Connect` permission to grant `iot:Subscribe` permission.

Thing Shadow Policy Actions

iot:DeleteThingShadow

Represents the permission to delete a thing shadow. The `iot:DeleteThingShadow` permission is checked every time a request is made to delete the thing shadow document.

iot:GetThingShadow

Represents the permission to retrieve a thing shadow. The `iot:GetThingShadow` permission is checked every time a request is made to retrieve a thing shadow document.

iot:UpdateThingShadow

Represents the permission to update a thing shadow. The `iot:UpdateThingShadow` permission is checked every time a request is made to update the state of a thing shadow document.

Action Resources

To specify a resource for an AWS IoT 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
iot:DeleteThingShadow	A thing ARN - arn:aws:iot:us-east-1:123456789012:thing/thingOne
iot:Connect	A client ID ARN - arn:aws:iot:us-east1:123456789012:client/myClientId
iot:Publish	A topic ARN - arn:aws:iot:us-east-1:123456789012:topic/myTopicName
iot:Subscribe	A topic filter ARN - arn:aws:iot:us-east-1:123456789012:topicFilter/myTopicFilter
iot:Receive	A topic ARN - arn:aws:iot:us-east-1:123456789012:topic/myTopicName
iot:UpdateThingShadow	A thing ARN - arn:aws:iot:us-east-1:123456789012:thing/thingOne
iot:GetThingShadow	A thing ARN - arn:aws:iot:us-east-1:123456789012:thing/thingOne

AWS IoT Policy Variables

AWS IoT defines policy variables that can be used in AWS IoT policies within the resource or condition block. When a policy is evaluated, the policy variables are replaced by actual values. For example, if a device connected to the AWS IoT message broker with a client ID of "100-234-3456", the `iot:ClientId` policy variable would be replaced in the policy document by "100-234-3456". For more information about policy variables, see IAM Policy Variables and Multi-Value Conditions.

Basic Policy Variables

AWS IoT defines the following basic policy variables:

- `iot:ClientId`: The client ID used to connect to the AWS IoT message broker.
- `aws:SourceIp`: The IP address of the client connected to the AWS IoT message broker.

The following AWS IoT policy shows the use of policy variables:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": ["iot:Connect"],
        "Resource": [
            "arn:aws:iot:us-east-1:123451234510:client/${iot:ClientId}"
        ]
    },
    {
        "Effect": "Allow",
        "Action": ["iot:Publish"],
        "Resource": [
            "arn:aws:iot:us-east-1:123451234510:topic/foo/bar/${iot:ClientId}"
        ]
    }]
}
```

In these examples `${iot:ClientId}` will be replaced by the ID of the client connected to the AWS IoT message broker when the policy is evaluated.When you use policy variables like `${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/foo/${iot:ClientId}/bar"
    ]
}
```

A client can connect using `+` as the client ID. This would allow the user to subscribe to any topic matching the topic filter `foo/+/bar`. To protect against such security gaps, use the `iot:Connect` policy action to control which client IDs are able to connect. For example, this policy will allow only 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"
        ]
    }]
}
```

X.509 Certificate Policy Variables

X.509 certificate policy variables allow you to write AWS IoT policies that grant permissions based on X.509 certificate attributes. The following sections describe how you can use these certificate policy variables.

Issuer Attributes

The following AWS IoT 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 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. When evaluated in a policy, `iot:Certificate.Subject.Country` is replaced by the first country name. You can request a specific attribute value using a zero-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 attribute 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 will be made and authorization will fail. You can use the `.List` suffix on the policy variable name to specify all values of the attribute. The following example policy allows any client to connect to AWS IoT, but restricts publishing rights to those clients with certificates whose `Certificate.Subject.Organization` attribute is set to `"Example Corp"` or `"AnyCompany"`. This is done through the use of a `"Condition"` attribute that specifies a condition for the preceding action. The condition in this case is that the `Certificate.Subject.Organization` attribute of the certificate must include one of the listed values.

```
{
    "Version":"2012-10-17",
    "Statement":[
        {
            "Effect":"Allow",
            "Action":[
                "iot:Connect"
            ],
            "Resource":[
                "*"
            ]
        },
        {
            "Effect":"Allow",
            "Action":[
                "iot:Publish"
            ],
            "Resource":[
                "*"
            ],
            "Condition":{
                "ForAllValues:StringEquals":{
                    "iot:Certificate.Subject.Organization.List":[
                        "Example Corp",
                        "AnyCompany"
                    ]
                }
            }
        }
    ]
}
```

Issuer Alternate Name Attributes

The following AWS IoT 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 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 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 will not be replaced by the certificate attribute value, leaving the `${policy-variable}` text in the policy document. This might cause authorization failure.

Array fields

: Certificate attributes that contain arrays are limited to five items. Additional items will be 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 will not be replaced by the certificate attribute value, leaving the `${policy-variable}` in the policy document. This might cause authorization failure.

Thing Policy Variables

Thing policy variables allow you to write AWS IoT policies that 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. Thing policy variables are also replaced when a certificate or authenticated Amazon Cognito identity is attached to a thing. You can use the AttachThingPrincipal API to attach certificates and authenticated Amazon Cognito identities to a thing.

The following thing policy variables are available:

- `iot:Connection.Thing.ThingName`
- `iot:Connection.Thing.ThingTypeName`
- `iot:Connection.Thing.Attributes[`*`attributeName`*`]`
- `iot:Connection.Thing.IsAttached`

iot:Connection.Thing.ThingName

This resolves to the name of 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.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. The thing type name is obtained by a call to the `DescribeThing` API. 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 will be 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

This resolves to `true` if the thing for which the policy is being evaluated has a certificate or Amazon Cognito identity attached.

Example Policies

AWS IoT policies are specified in a JSON document. These are the components of an AWS IoT policy:

Version

Must be set to `"2012-10-17"`.

Effect

Must be set to `"Allow"` or `"Deny"`.

Action

Must be set to "iot:*operation-name*" where *operation-name* is one of the following:

`"iot:Publish"`: MQTT publish.

`"iot:Subscribe"`: MQTT subscribe.

`"iot:UpdateThingShadow"`: Update a thing shadow.

`"iot:GetThingShadow"`:Retrieve a thing shadow.

`"iot:DeleteThingShadow`:Delete a thing shadow.

Resource

Must be set to one of the following:

Client - arn:aws:iot:*region*:*account-id*:client/*client-id*

Topic ARN - arn:aws:iot:*region*:*account-id*:topic/*topic-name*

Topic filter ARN - arn:aws:iot:*region*:*account-id*:topicfilter/*topic-filter*

Connect Policy Examples

The following policy allows a set of client IDs to connect:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/clientid1",
                "arn:aws:iot:us-east-1:123456789012:client/clientid2",
                "arn:aws:iot:us-east-1:123456789012:client/clientid3"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish",
                "iot:Subscribe",
                "iot:Receive"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

The following policy prevents a set of client IDs from connecting:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/clientid1",
                "arn:aws:iot:us-east-1:123456789012:client/clientid2"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

The following policy allows the certificate holder using any client ID to subscribe to topic filter `foo/*`:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
```

```
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topicfilter/foo/*"
            ]
        }
    ]
}
```

Publish/Subscribe Policy Examples

The policy you use will depend on how you are connecting to AWS IoT. You can connect to AWS IoT using an MQTT client, HTTP, or WebSocket. When you connect with an MQTT client, you will be authenticating with an X.509 certificate. When you connect over HTTP or the WebSocket protocol, you will be authenticating with Signature Version 4 and Amazon Cognito.

Policies for MQTT Clients

When you specify topic filters in AWS IoT policies for MQTT clients, MQTT wildcard characters "+" and "#" will be treated as literal characters. Their use might result in unexpected behavior. For example, the following policy will allow a client to subscribe to the topic filter `foo/+/bar` only:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topicfilter/foo/+/bar"
            ]
        }
    ]
}
```

Note
Attempts to subscribe to topic filters that match the pattern `foo/+/bar` like `foo/baz/bar` or `foo/goo/bar` will fail and cause the client to disconnect.

You can use "*" as a wildcard in the resource attribute of the policy. For example, the following policy allows the certificate holder to publish to all topics and subscribe to all topic filters in the AWS account:

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

The following policy allows the certificate holder to publish to all topics in the AWS account:

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

You can also use the "*" wildcard at the end of a topic filter. For example, the following policy allows the certificate holder to subscribe to a topic or topic filter matching the pattern `foo/bar/*`:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topicfilter/foo/bar/*"
            ]
        }
    ]
}
```

The following policy allows the certificate holder to publish to the `foo/bar` and `foo/baz` topics:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/foo/bar",
                "arn:aws:iot:us-east-1:123456789012:topic/foo/baz"
            ]
        }
    ]
}
```

The following policy prevents the certificate holder from publishing to the `foo/bar` topic:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "*"
            ]
        },
        {
            "Effect": "Deny",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/foo/bar"
            ]
        }
    ]
}
```

The following policy allows the certificate holder to publish on topic `foo` and prevents the certificate holder from publishing to topic `bar`:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
```

```
                "*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/foo"
            ]
        },
        {
            "Effect": "Deny",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/bar"
            ]
        }
    ]
}
```

The following policy allows the certificate holder to subscribe to topic filter `foo/bar`:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topicfilter/foo/bar"
            ]
        }
    ]
}
```

The following policy allows the certificate holder to publish on the `arn:aws:iot:us-east-1:123456789012:topic/iotmonitor/provisioning/8050373158915119971` topic and allows the certificate holder to subscribe to the topic filter `arn:aws:iot:us-east-1:123456789012:topicfilter/iotmonitor/provisioning/8050373158915119971`:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
```

```
            "Resource": [
                "*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish",
                "iot:Receive"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/iotmonitor/
provisioning/8050373158915119971"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topicfilter/iotmonitor/
provisioning/8050373158915119971"
            ]
        }
    ]
}
```

Policies for HTTP and WebSocket Clients

For the following operations, AWS IoT uses AWS IoT policies attached to Amazon Cognito identities (through the `AttachPrincipalPolicy` API) to scope down the permissions attached to the Amazon Cognito identity pool with authenticated identities. That means an Amazon Cognito identity needs permission from the IAM role policy attached to the pool and the AWS IoT policy attached to the Amazon Cognito identity through the AWS IoT `AttachPrincipalPolicy` API.

- `iot:Connect`
- `iot:Publish`
- `iot:Subscribe`
- `iot:Receive`
- `iot:GetThingShadow`
- `iot:UpdateThingShadow`
- `iot:DeleteThingShadow`

> **Note**
> For other AWS IoT operations or for unauthenticated identities, AWS IoT 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 attaching to the Amazon Cognito pool role.

To allow unauthenticated Amazon Cognito identities to publish messages over HTTP on any topic, attach the following policy to the Amazon Cognito identity pool role:

```
{
    "Version": "2012-10-17",
    "Statement": [
    {
        "Effect": "Allow",
        "Action": [
```

```
            "iot:Connect",
            "iot:Publish",
            "iot:Subscribe",
            "iot:Receive",
            "iot:GetThingShadow",
            "iot:UpdateThingShadow",
            "iot:DeleteThingShadow"
        ],
        "Resource": ["*"]
    }]
}
```

To allow unauthenticated Amazon Cognito identities to publish MQTT messages over HTTP on any topic in your account, attach the following policy to the Amazon Cognito identity pool role:

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

Note
This example is for illustration only. Unless your service absolutely requires it, we recommend the use of a more restrictive policy, one that does not allow unauthenticated Amazon Cognito identities to publish on any topic.

To allow unauthenticated Amazon Cognito identities to publish MQTT messages over HTTP on `topic1` in your account, attach the following policy to your Amazon Cognito identity pool role:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": ["iot:Publish"],
        "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/topic1"]
    }]
}
```

For an authenticated Amazon Cognito identity to publish MQTT messages over HTTP on `topic1` in your AWS account, you must specify two policies, as outlined here. The first policy must be attached to an Amazon Cognito identity pool role. It allows identities from that pool to make a publish call. The second policy must be attached to an Amazon Cognito user using the AWS IoT AttachPrincipalPolicy API. It allows the specified Amazon Cognito user access to the `topic1` topic.

Amazon Cognito identity pool policy:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": [ "iot:Publish"],
        "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/topic1"]
    }]
}
```

Amazon Cognito user policy:

```
{
```

```
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": ["iot:Publish"],
        "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/topic1"]
    }]
}
```

Similarly, the following example policy allows the Amazon Cognito user to publish MQTT messages over HTTP on the `topic1` and `topic2` topics. Two policies are required. The first policy gives the Amazon Cognito identity pool role the ability to make the publish call. The second policy gives the Amazon Cognito user access to the `topic1` and `topic2` topics.

Amazon Cognito identity pool policy:

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

Amazon Cognito user policy:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": ["iot:Publish"],
        "Resource": [
            "arn:aws:iot:us-east-1:123456789012:topic/topic1",
            "arn:aws:iot:us-east-1:123456789012:topic/topic2"
        ]
    }]
}
```

The following policies allow multiple Amazon Cognito users to publish to a topic. Two policies per Amazon Cognito identity are required. The first policy gives the Amazon Cognito identity pool role the ability to make the publish call. The second and third policies give the Amazon Cognito users access to the topics `topic1` and `topic2`, respectively.

Amazon Cognito identity pool policy:

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

Amazon Cognito user1 policy:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": ["iot:Publish"],
```

```
        "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/topic1"]
    }]
}
```

Amazon Cognito user2 policy:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": ["iot:Publish"],
        "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/topic2"]
    }]
}
```

Receive Policy Examples

The following policy prevents the certificate holder using any client ID from receiving messages from a topic:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "iot:Receive"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/foo/restricted"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:*"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

The following policy allows the certificate holder using any client ID to subscribe and receive messages on one topic:

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

```
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topicfilter/foo/bar"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Receive"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/foo/bar"
            ]
        }
    ]
}
```

Certificate Policy Examples

The following policy allows a device to publish on a topic whose name is equal to the `certificateId` of the certificate with which the device authenticated 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": ["*"]
    }]
}
```

The following policy allows a device to publish on a topic whose name is equal to the subject's common name field of the certificate with which the device authenticated itself:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action":["iot:Publish"],
        "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/
${iot:Certificate.Issuer.CommonName}"]
    },
    {
        "Effect": "Allow",
        "Action": ["iot:Connect"],
        "Resource": ["*"]
    }]
}
```

The following policy allows a device to publish on a topic that 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": ["*"]
```

```
    },
    {
        "Effect": "Allow",
        "Action": ["iot:Publish"],
        "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/admin/*"],
        "Condition": {
            "StringEquals": {
                "iot:Certificate.Subject.CommonName.2": "Administrator"
            }
        }
    }]
}
```

The following policy allows a device to publish on a topic that is prefixed with "admin/" when the certificate used to authenticate the device has any one of its `Subject.Common` fields set to "Administrator":

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": ["iot:Connect"],
        "Resource": ["*"]
    },
    {
        "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 thing to publish on a specific topic that contains the thing type name and thing name:

```
{
    "Version":"2012-10-17",
    "Statement":[{
        "Effect":"Allow",
        "Action":["iot:Publish"],
        "Resource":[
            "arn:aws:iot:us-east-1:123456789012:topic/
${iot:Connection.Thing.ThingTypeName}/${iot:Connection.Thing.ThingName}"
        ]
    }]
}
```

The following policy allows the device to connect if the certificate used to authenticate with AWS IoT 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"]
            }
        }
    }]
}
```

The following policy allows a device to publish on a set of topics ("/foo/bar" and "/foo/baz") if:

- The thing associated with the device has an attribute called "Manufacturer" with a value of "foo", "bar", or "baz".
- The thing associated with the device exists in the thing registry and is attached to the certificate used to connect to AWS IoT.

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": ["iot:Publish"],
        "Resource": [
            "arn:aws:iot:us-east-1:123456789012:topic/foo/bar",
            "arn:aws:iot:us-east-1:123456789012:topic/foo/baz"
        ],
        "Condition": {
            "ForAnyValue:StringLike": {
                "iot:Connection.Thing.Attributes[Manufacturer]": [
                    "foo",
                    "bar",
                    "baz"
                ]
            }
        }
    }]
}
```

The following policy allows a device to publish to a topic if:

- The topic is composed of the thing type name, a '/', and the thing name.
- The thing exists in the thing registry.
- The thing is attached to the certificate used to connect to AWS IoT.

```
{
    "Version":"2012-10-17",
    "Statement":[{
        "Effect":"Allow",
        "Action":["iot:Publish"],
        "Resource":[
            "arn:aws:iot:us-east-1:123456789012:topic/
${iot:Connection.Thing.ThingTypeName}/${iot:Connection.Thing.ThingName}"
         ]
    }]
}
```

The following policy allows a device to publish only on its own thing shadow topic, if the thing exists in the thing registry.

```
{
```

```
    "Version": "2012-10-17",
    "Statement": [{
       "Effect": "Allow",
        "Action": ["iot:Publish"],
        "Resource": [
            "arn:aws:iot:us-east-1:123456789012:topic/$aws/things/
${iot:Connection.Thing.ThingName}/shadow/update"
        ]
    }]
}
```

IAM IoT Policies

AWS Identity and Access Management defines a policy action for each operation defined by AWS IoT, including control plane and data plane APIs.

AWS IoT API Permissions

The following table lists the AWS IoT API, the IAM permissions required, and the resource the API manipulates.

API	Required Permission (Policy Actions)	Resources
AttachPrincipalPolicy	iot:AttachPrincipalPolicy	arn:aws:iot:*region*:*account-id*:cert/*cert-id*
AttachThingPrincipal	iot:AttachThingPrincipal	arn:aws:iot:*region*:*account-id*:cert/*cert-id*
CancelCertificateTransfer	iot:CancelCertificateTransfer	arn:aws:iot:*region*:*account-id*:cert/*cert-id* **Note** The AWS account specified in the ARN must be the account to which the certificate is being transferred.
CreateCertificateFromCsr	iot:CreateCertificateFromCsr	*
CreateKeysAndCertificate	iot:CreateKeysAndCertificate	*
CreatePolicy	iot:CreatePolicy	*
CreatePolicyVersion	iot:CreatePolicyVersion	arn:aws:iot:*region*:*account-id*:policy/*policy-name* **Note** This must be an AWS IoT policy, not an IAM policy.
CreateThing	iot:CreateThing	arn:aws:iot:*region*:*account-id*:thing/*thing-name*.
CreateThingType	iot:CreateThingType	arn:aws:iot:*region*:*account-id*:thingtype/*thing-type-name*
CreateTopicRule	iot:CreateTopicRule	arn:aws:iot:*region*:*account-id*:rule/*rule-name*
DeleteCACertificate	iot:DeleteCACertificate	arn:aws:iot:*region*:*account-id*:cacert/*cert-id*
DeleteCertificate	iot:DeleteCertificate	arn:aws:iot:*region*:*account-id*:cert/*cert-id*
DeletePolicy	iot:DeletePolicy	arn:aws:iot:*region*:*account-id*:policy/*policy-name*
DeletePolicyVersion	iot:DeletePolicyVersion	arn:aws:iot:*region*:*account-id*:policy/*policy-name*
DeleteRegistrationCode	iot:DeleteRegistrationCode	*

API	Required Permission (Policy Actions)	Resources
DeleteThing	iot:DeleteThing	arn:aws:iot:*region*:*account-id*:thing/*thing-name*
DeleteThingType	iot:DeleteThingType	arn:aws:iot:*region*:*account-id*:thingtype/*thing-type-name*
DeleteTopicRule	iot:DeleteTopicRule	arn:aws:iot:*region*:*account-id*:rule/*rule-name*
DeprecateThingType	iot:DeprecateThingType	arn:aws:iot:*region*:*account-id*:thingtype/*thing-type-name*
DescribeCaCertificate	iot:DescribeCaCertificate	arn:aws:iot:*region*:*account-id*:cacert/*cert-id*
DescribeCertificate	iot:DescribeCertificate	arn:aws:iot:*region*:*account-id*:cert/*cert-id*
DescribeEndpoint	iot:DescribeEndpoint	*
DescribeThing	iot:DescribeThing	arn:aws:iot:*region*:*account-id*:thing/*thing-name*
DescribeThingType	iot:DescribeThingType	arn:aws:iot:*region*:*account-id*:thingtype/*thing-type-name*
DetachPrincipalPolicy	iot:DetachPrincipalPolicy	arn:aws:iot:*region*:*account-id*:cert/*cert-id*
DetachThingPrincipal	iot:DetachThingPrincipal	arn:aws:iot:*region*:*account-id*:cert/*cert-id*
DisableTopicRule	iot:DisableTopicRule	arn:aws:iot:*region*:*account-id*:rule/*rule-name*
EnableTopicRule	iot:EnableTopicRule	arn:aws:iot:*region*:*account-id*:rule/*rule-name*
GetLoggingOptions	iot:GetLoggingOptions	*
GetPolicy	iot:GetPolicy	arn:aws:iot:*region*:*account-id*:policy/*policy-name*
GetPolicyVersion	iot:GetPolicyVersion	arn:aws:iot:*region*:*account-id*:policy/*policy-name*
GetRegistrationCode	iot:GetRegistrationCode	*
GetTopicRule	iot:GetTopicRule	arn:aws:iot:*region*:*account-id*:rule/*rule-name*
ListCaCertificates	iot:ListCaCertificates	*
ListCertificates	iot:ListCertificates	*
iot:ListCertificatesByCa	iot:ListCertificatesByCa	*
ListOutgoingCertificates	iot:ListOutgoingCertificates	*
ListPolicies	iot:ListPolicies	*
ListPolicyPrincipals	iot:ListPolicyPrincipals	The ARN of the policy: arn:aws:iot:*region*:*account-id*:policy/*policy-name*
ListPolicyVersions	iot:ListPolicyVersions	The ARN of the policy: arn:aws:iot:*region*:*account-id*:policy/*policy-name*
ListPrincipalPolicies	iot:ListPrincipalPolicies	The ARN of the certificate: arn:aws:iot:*region*:*account-id*:cert/*cert-id*
ListPrincipalThings	iot:ListPrincipalThings	The ARN of the certificate: arn:aws:iot:*region*:*account-id*:cert/*cert-id*

API	Required Permission (Policy Actions)	Resources
ListThingPrincipals	iot:ListThingPrincipals	The ARN of the AWS IoT thing: arn:aws:iot:*region*:*account-id*:thing/*thing-name*
ListThings	iot:ListThings	*
ListThingTypes	iot:ListThingTypes	*
ListTopicRules	iot:ListTopicRules	*
RegisterCACertificate	iot:RegisterCACertificate	*
RegisterCertificate	iot:RegisterCertificate	*
RejectCertificateTransfer	iot:RejectCertificateTransfer	arn:aws:iot:*region*:*account-id*:cert/*cert-id*
ReplaceTopicRule	iot:ReplaceTopicRule	arn:aws:iot:*region*:*account-id*:rule/*rule-name*
SetDefaultPolicyVersion	iot:SetDefaultPolicyVersion	arn:aws:iot:*region*:*account-id*:policy/*policy-name*
SetLoggingOptions	iot:SetLoggingOptions	arn:aws:iot:*region*:*account-id*:role/*role-name*
TransferCertificate	iot:TransferCertificate	arn:aws:iot:*region*:*account-id*:cert/*cert-id*
UpdateCACertificate	iot:UpdateCACertificate	arn:aws:iot:*region*:*account-id*:cacert/*cert-id*
UpdateCertificate	iot:UpdateCertificate	arn:aws:iot:*region*:*account-id*:cert/*cert-id*
UpdateThing	iot:UpdateThing	arn:aws:iot:*region*:*account-id*:thing/*thing-name*

IAM Policy Templates

AWS IoT provides a set of IAM policy templates you can either use as-is or as a starting point for creating custom IAM policies. These templates 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
AWSIotLogging	Allows the associated identity to configure CloudWatch logging. This policy is attached to your CloudWatch logging role.
AWSIoTConfigAccess	Allows the associated identity access to all AWS IoT configuration operations.
AWSIoTConfigReadOnlyAccess	Allows the associated identity to call 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.
AWSIoTFullAccess	Allows the associated identity full access to all AWS IoT configuration and data operations.

Policy Template	Description
AWSIoTRuleActions	Allows the associated identity access to all AWS services supported in AWS IoT rule actions.

Cross Account Access

AWS IoT 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 an IAM policy just like you would for other users and certificates in your AWS account. For example, the following policy grants permissions to connect and publish to the `/foo/bar` topic.

```
{
    "Version": "2012-10-17",
    "Statement": [
    {
        "Effect": "Allow",
        "Action": [
            "iot:Connect"
         ],
         "Resource": [
             "*"
         ]
     },
     {
         "Effect": "Allow",
         "Action": [
             "iot:Publish"
         ],
         "Resource": [
             "arn:aws:iot:us-east-1:123456789012:topic/foo/bar"
         ]
     }]
}
```

Next, follow the steps in Creating a Role for an IAM User. Enter the AWS 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-1:111111111111:user/MyUser"
        },
        "Action": "sts:AssumeRole"
    }]
}
```

Transport Security

The AWS IoT message broker and Thing Shadows service encrypt all communication with TLS. TLS is used to ensure the confidentiality of the application protocols (MQTT, HTTP) supported by AWS IoT. TLS is available in a number of programming languages and operating systems.

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.

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

Message Broker for AWS IoT

The AWS IoT message broker is a publish/subscribe broker service that enables the sending and receiving of messages to and from AWS IoT. When communicating with AWS IoT, a client sends a message addressed to a topic like `Sensor/temp/room1`. The message broker, in turn, sends the message to all clients that have registered to receive messages for that topic. The act of sending the message is referred to as *publishing*. The act of registering to receive messages for a topic filter is referred to as *subscribing*.

The topic namespace is isolated for each AWS account and region pair. For example, the `Sensor/temp/room1` topic for an AWS account is independent from the `Sensor/temp/room1` topic for another AWS account. This is true of regions, too. The `Sensor/temp/room1` topic in the same AWS account in us-east-1 is independent from the same topic in us-west-2. AWS IoT does not support sending and receiving messages across AWS accounts and regions.

The message broker maintains a list of all client sessions and the subscriptions for each session. When a message is published on a topic, the broker checks for sessions with subscriptions that map to the topic. The broker then forwards the publish message to all sessions that have a currently connected client.

Protocols

The message broker supports the use of the MQTT protocol to publish and subscribe and the HTTPS protocol to publish. Both protocols are supported through IP version 4 and IP version 6. The message broker also supports MQTT over the WebSocket protocol.

Protocol/Port Mappings

The following table shows each protocol supported by AWS IoT, the authentication method, and port used for each protocol.

Protocol, Authentication, and Port Mappings

Protocol	Authentication	Port
MQTT	Client Certificate	8883
HTTP	Client Certificate	8443
HTTP	SigV4	443

Protocol	Authentication	Port
MQTT + WebSocket	SigV4	443

MQTT

MQTT is a widely adopted lightweight messaging protocol designed for constrained devices. For more information, see MQTT.

Although the AWS IoT message broker implementation is based on MQTT version 3.1.1, it deviates from the specification as follows:

- In AWS IoT, subscribing to a topic with Quality of Service (QoS) 0 means a message will be 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.
- AWS IoT does not support publishing and subscribing with QoS 2. The AWS IoT message broker does not send a PUBACK or SUBACK when QoS 2 is requested.
- The QoS levels for publishing and subscribing to a topic have no relation to each other. One client can subscribe to a topic using QoS 1 while another client can publish to the same topic using QoS 0.
- 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. The value of this flag might be incorrect if two MQTT clients connect with the same client ID simultaneously.
- 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.
- The MQTT specification provides a provision for the publisher to request that the broker retain the last message sent to a topic and send it to all future topic subscribers. AWS IoT does not support retained messages. If a request is made to retain messages, the connection is disconnected.
- 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 are not allowed to be connected concurrently to the message broker. When a client connects to the message broker using a client ID that another client is using, a CONNACK message will be sent to both clients and the currently connected client will be disconnected.
- The message broker does not support persistent sessions (clean session set to 0). All sessions are assumed to be clean sessions and messages are not stored across sessions. If an MQTT client sends a message with the clean session attribute set to false, the client will be disconnected.
- On rare occasions, the message broker might resend the same logical PUBLISH message with a different packet ID.
- The message broker does not guarantee the order in which messages and ACK are received.

HTTP

The message broker supports clients connecting with the HTTP protocol using a REST API. Clients can publish by sending a POST message to `<AWS IoT Endpoint>/topics/<url_encoded_topic_name>?qos=1"`.

For example, you can use curl to emulate a button press. If you followed the tutorial in Getting Started with AWS IoT (p. 19), rather than using the AWS IoT MQTT client to publish a message as you did in AWS IoT MQTT Client (p. 44), use something like the following command:

```
curl --tlsv1.2 --cacert root-CA.crt --cert 4b7828d2e5-certificate.pem.crt --key 4b7828d2e5-
private.pem.key -X POST -d "{ \"serialNumber\": \"G030JF053216F1BS\", \"clickType
```

```
\": \"SINGLE\", \"batteryVoltage\": \"2000mV\" }" "https://a1pn10j0v8htvw.iot.us-
east-1.amazonaws.com:8443/topics/iotbutton/virtualButton?qos=1"
```

--tlsv1.2

Use TLSv1.2 (SSL). curl must be installed with OpenSSL and you must use version 1.2 of TLS.

--cacert <filename>

The filename of the CA certificate to verify the peer.

--cert <filename>

The client certificate filename.

--key <filename>

The private key filename.

-X POST

The type of request, in this case, POST.

-d <data>

The HTTP POST data you want to publish. In this case, we emulate the data sent by a single button press.

"https://..."

The URL. In this case the REST API endpoint for the thing. (To find the endpoint for a thing, from the AWS IoT console choose **Registry** to expand your choices. Choose **Things**, choose the thing, and then choose **Interact**.) After the endpoint add the port (:8443) followed by the topic and, finally, specify the quality of service in a query string (?qos=1).

MQTT Over the WebSocket Protocol

AWS IoT supports MQTT over the WebSocket protocol to enable browser-based and remote applications to send and receive data from AWS IoT-connected devices using AWS credentials. AWS credentials are specified using AWS Signature Version 4. WebSocket support is available on TCP port 443, which allows messages to pass through most firewalls and web proxies.

A WebSocket connection is initiated on a client by sending an HTTP GET request. The URL you use is of the following form:

```
wss://<endpoint>.iot.<region>.amazonaws.com/mqtt
```

wss

Specifies the WebSocket protocol.

endpoint

Your AWS account-specific AWS IoT endpoint. You can use the AWS IoT CLI describe-endpoint command to find this endpoint.

region

The AWS region of your AWS account.

mqtt

Specifies you will be sending MQTT messages over the WebSocket protocol.

When the server responds, the client sends an upgrade request to indicate to the server it will communicate using the WebSocket protocol. After the server acknowledges the upgrade request, all communication is performed using the WebSocket protocol. The WebSocket implementation you use acts as a transport protocol. The data you send over the WebSocket protocol are MQTT messages.

Using the WebSocket Protocol in a Web Application

The WebSocket implementation provided by most web browsers does not allow the modification of HTTP headers, so you must add the Signature Version 4 information to the query string. For more information, see Adding Signing Information to the Query String.

The following JavaScript defines some utility functions used in generating a Signature Version 4 request.

```
  /**
   * utilities to do sigv4
   * @class SigV4Utils
   */
  function SigV4Utils() {}

SigV4Utils.getSignatureKey = function (key, date, region, service) {
    var kDate = AWS.util.crypto.hmac('AWS4' + key, date, 'buffer');
    var kRegion = AWS.util.crypto.hmac(kDate, region, 'buffer');
    var kService = AWS.util.crypto.hmac(kRegion, service, 'buffer');
    var kCredentials = AWS.util.crypto.hmac(kService, 'aws4_request', 'buffer');
    return kCredentials;
};

SigV4Utils.getSignedUrl = function(host, region, credentials) {
    var datetime = AWS.util.date.iso8601(new Date()).replace(/[:\-]|\.\d{3}/g, '');
    var date = datetime.substr(0, 8);

    var method = 'GET';
    var protocol = 'wss';
    var uri = '/mqtt';
    var service = 'iotdevicegateway';
    var algorithm = 'AWS4-HMAC-SHA256';

    var credentialScope = date + '/' + region + '/' + service + '/' + 'aws4_request';
    var canonicalQuerystring = 'X-Amz-Algorithm=' + algorithm;
    canonicalQuerystring += '&X-Amz-Credential=' +
 encodeURIComponent(credentials.accessKeyId + '/' + credentialScope);
    canonicalQuerystring += '&X-Amz-Date=' + datetime;
    canonicalQuerystring += '&X-Amz-SignedHeaders=host';

    var canonicalHeaders = 'host:' + host + '\n';
    var payloadHash = AWS.util.crypto.sha256('', 'hex')
    var canonicalRequest = method + '\n' + uri + '\n' + canonicalQuerystring + '\n' +
 canonicalHeaders + '\nhost\n' + payloadHash;

    var stringToSign = algorithm + '\n' + datetime + '\n' + credentialScope + '\n' +
 AWS.util.crypto.sha256(canonicalRequest, 'hex');
    var signingKey = SigV4Utils.getSignatureKey(credentials.secretAccessKey, date, region,
 service);
    var signature = AWS.util.crypto.hmac(signingKey, stringToSign, 'hex');

    canonicalQuerystring += '&X-Amz-Signature=' + signature;
    if (credentials.sessionToken) {
        canonicalQuerystring += '&X-Amz-Security-Token=' +
 encodeURIComponent(credentials.sessionToken);
    }

    var requestUrl = protocol + '://' + host + uri + '?' + canonicalQuerystring;
    return requestUrl;
};
```

To create a Signature Version 4 request

1. Create a canonical request for Signature Version 4.

   The following JavaScript code creates a canonical request:

```
var datetime = AWS.util.date.iso8601(new Date()).replace(/[:\-]|\.\d{3}/g, '');
var date = datetime.substr(0, 8);

var method = 'GET';
var protocol = 'wss';
var uri = '/mqtt';
var service = 'iotdevicegateway';
var algorithm = 'AWS4-HMAC-SHA256';

var credentialScope = date + '/' + region + '/' + service + '/' + 'aws4_request';
var canonicalQuerystring = 'X-Amz-Algorithm=' + algorithm;
canonicalQuerystring += '&X-Amz-Credential=' +
 encodeURIComponent(credentials.accessKeyId + '/' + credentialScope);
canonicalQuerystring += '&X-Amz-Date=' + datetime;
canonicalQuerystring += '&X-Amz-SignedHeaders=host';

var canonicalHeaders = 'host:' + host + '\n';
var payloadHash = AWS.util.crypto.sha256('', 'hex')
var canonicalRequest = method + '\n' + uri + '\n' + canonicalQuerystring + '\n' +
 canonicalHeaders + '\nhost\n' + payloadHash;
```

2. Create a string to sign, generate a signing key, and sign the string.

   Take the canonical URL you created in the previous step and assemble it into a string to sign. You do this by creating a string composed of the hashing algorithm, the date, the credential scope, and the SHA of the canonical request. Next, generate the signing key and sign the string, as shown in the following JavaScript code.

```
var stringToSign = algorithm + '\n' + datetime + '\n' + credentialScope + '\n' +
 AWS.util.crypto.sha256(canonicalRequest, 'hex');
var signingKey = SigV4Utils.getSignatureKey(credentials.secretAccessKey, date, region,
 service);
var signature = AWS.util.crypto.hmac(signingKey, stringToSign, 'hex');
```

3. Add the signing information to the request.

   The following JavaScript code shows how to add the signing information to the query string.

```
canonicalQuerystring += '&X-Amz-Signature=' + signature;
```

4. If you have session credentials (from an STS server, AssumeRole, or Amazon Cognito), append the session token to the end of the URL string after signing:

```
canonicalQuerystring += '&X-Amz-Security-Token=' +
 encodeURIComponent(credentials.sessionToken);
```

5. Prepend the protocol, host, and URI to the canonicalQuerystring:

```
var requestUrl = protocol + '://' + host + uri + '?' + canonicalQuerystring;
```

6. Open the WebSocket.

   The following JavaScript code shows how to create a Paho MQTT client and call CONNECT to AWS IoT. The `endpoint` argument is your AWS account-specific endpoint. The `clientId` is a text identifier that is unique among all clients simultaneously connected in your AWS account.

```
var client = new Paho.MQTT.Client(requestUrl, clientId);
var connectOptions = {
    onSuccess: function(){
        // connect succeeded
    },
    useSSL: true,
    timeout: 3,
    mqttVersion: 4,
    onFailure: function() {
        // connect failed
    }
};
client.connect(connectOptions);
```

Using the WebSocket Protocol in a Mobile Application

We recommend using one of the AWS IoT Device SDKs to connect your device to AWS IoT when making a WebSocket connection. The following AWS IoT Device SDKs support WebSocket-based MQTT connections to AWS IoT:

- Node.js
- iOS
- Android

For a reference implementation for connecting a web application to AWS IoT using MQTT over the WebSocket protocol, see AWS Labs WebSocket sample.

If you are using a programming or scripting language that is not currently supported, any existing WebSocket library can be used as long as the initial WebSocket upgrade request (HTTP POST) is signed using AWS Signature Version 4. Some MQTT clients, such as Eclipse Paho for JavaScript, support the WebSocket protocol natively.

Topics

The message broker uses topics to route messages from publishing clients to subscribing clients. The forward slash (/) is used to separate topic hierarchy. The following table lists the wildcards that can be used in the topic filter when you subscribe.

Topic Wildcards

Wildcard	Description
#	Must be the last character in the topic to which you are subscribing. Works as a wildcard by matching the current tree and all subtrees. For example, a subscription to `Sensor/#` will

Wildcard	Description
	receive messages published to `Sensor/`, `Sensor/temp`, `Sensor/temp/room1`, but not the messages published to `Sensor`.
+	Matches exactly one item in the topic hierarchy. For example, a subscription to `Sensor/+/room1` will receive messages published to `Sensor/temp/room1`, `Sensor/moisture/room1`, and so on.

Reserved Topics

Any topics beginning with $ are considered reserved and are not supported for publishing and subscribing except for those topics listed below. Any other attemps to publish or subscribe on topics beginning with $ will result in a terminated connection.

Topic	Allowed Operations	Description
$aws/events/presence/connected/*clientId*	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. 137).
$aws/events/presence/disconnected/*clientId*	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. 137).
$aws/events/subscriptions/subscribed/*clientId*	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. 138).
$aws/events/subscriptions/unsubscribed/*clientId*	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. 138).
$aws/things/*thingName*/shadow/delete	Publish/Subscribe	A thing or an application publishes to this topic to delete a thing shadow. For more information see http://docs.aws.amazon.com/iot/

Topic	Allowed Operations	Description
		latest/developerguide//thing-shadow-mqtt.html#delete-pub-sub-topic.
$aws/things/*thingName*/shadow/delete/accepted	Subscribe	The Thing Shadows service sends messages to this topic when a thing shadow is deleted. For more information see http://docs.aws.amazon.com/iot/latest/developerguide//thing-shadow-mqtt.html#delete-accepted-pub-sub-topic.
$aws/things/*thingName*/shadow/delete/rejected	Subscribe	The Thing Shadows service sends messages to this topic when a request to delete a thing shadow is rejected. For more information see http://docs.aws.amazon.com/iot/latest/developerguide//thing-shadow-mqtt.html#delete-rejected-pub-sub-topic.
$aws/things/*thingName*/shadow/get	Publish/Subscribe	An application or a thing publishes an empty message to this topic to get a thing shadow. For more information see http://docs.aws.amazon.com/iot/latest/developerguide//thing-shadow-mqtt.html.
$aws/things/*thingName*/shadow/get/accepted	Subscribe	The Thing Shadows service sends messages to this topic when a request for a thing shadow is made successfully. For more information see http://docs.aws.amazon.com/iot/latest/developerguide//thing-shadow-mqtt.html#get-accepted-pub-sub-topic.

Topic	Allowed Operations	Description
$aws/things/*thingName*/shadow/get/rejected	Subscribe	The Thing Shadows service sends messages to this topic when a request for a thing shadow is rejected. For more information see http://docs.aws.amazon.com/iot/latest/developerguide//thing-shadow-mqtt.html#get-rejected-pub-sub-topic.
$aws/things/*thingName*/shadow/update	Publish/Subscribe	A thing or application publishes to this topic to update a thing shadow. For more information see http://docs.aws.amazon.com/iot/latest/developerguide//thing-shadow-mqtt.html#update-pub-sub-topic.
$aws/things/*thingName*/shadow/update/accepted	Subscribe	The Thing Shadows service sends messages to this topic when an update is successfully made to a thing shadow. For more information see http://docs.aws.amazon.com/iot/latest/developerguide//thing-shadow-mqtt.html#update-accepted-pub-sub-topic.
$aws/things/*thingName*/shadow/update/rejected	Subscribe	The Thing Shadows service sends messages to this topic when an update to a thing shadow is rejected. For more information see http://docs.aws.amazon.com/iot/latest/developerguide//thing-shadow-mqtt.html#update-rejected-pub-sub-topic.

Topic	Allowed Operations	Description
$aws/things/*thingName*/shadow/update/delta	Subscribe	The Thing Shadows service sends messages to this topic when a difference is detected between the reported and desired sections of a thing shadow. For more information see http://docs.aws.amazon.com/iot/latest/developerguide//thing-shadow-mqtt.html#update-delta-pub-sub-topic.
$aws/things/*thingName*/shadow/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 http://docs.aws.amazon.com/iot/latest/developerguide//thing-shadow-mqtt.html#update-documents-pub-sub-topic.

Lifecycle Events

AWS IoT publishes lifecycle events on the MQTT topics discussed in the following sections. These messages allow you to be notified of lifecycle events from the message broker.

Note
Lifecycle messages might be sent out of order and you might receive duplicate messages.

Policy Required for Receiving Lifecycle Events

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:topicfilter/$aws/events/*"
        ]
    }]
}
```

Connect/Disconnect Events

AWS IoT publishes a message to the following MQTT topics when a client connects or disconnects:

```
$aws/events/presence/connected/clientId
```

or

```
$aws/events/presence/disconnected/clientId
```

Where `clientId` is the MQTT client ID that connects to or disconnects from the AWS IoT message broker.

The message published to this topic has the following structure:

```
{
    "clientId": "a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6",
    "timestamp": 1460065214626,
    "eventType": "connected",
    "sessionIdentifier": "00000000-0000-0000-0000-000000000000",
    "principalIdentifier": "000000000000/ABCDEFGHIJKLMNOPQRSTU:some-user/
ABCDEFGHIJKLMNOPQRSTU:some-user"
}
```

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 + will not receive lifecycle events.

eventType

The type of event. Valid values are `connected` or `disconnected`.

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.

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": "00000000-0000-0000-0000-000000000000",
    "principalIdentifier": "000000000000/ABCDEFGHIJKLMNOPQRSTU:some-user/
ABCDEFGHIJKLMNOPQRSTU: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 + will 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.

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.

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.

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 Elasticsearch 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.

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

- Deleting a Rule (p. 148)
- AWS IoT Rule Actions (p. 148)
- AWS IoT SQL Reference (p. 158)

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 called iot-role-trust.json:

```
{
    "Version":"2012-10-17",
    "Statement":[{
        "Effect": "Allow",
        "Principal": {
            "Service": "iot.amazonaws.com"
        },
        "Action": "sts:AssumeRole"
    }]
}
```

   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 will look 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 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 iot-policy.json file:

```
aws iam create-policy --policy-name my-iot-policy --policy-document file://my-iot-policy-document.json
```

For more information about how to grant access to AWS services in policies for AWS IoT, see Creating an AWS IoT Rule (p. 143).

The output of the create-policy command will contain the ARN of the policy. You will 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 that 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 creating 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.

Optional description

A textual description of the rule.

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. 158).

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. If this property is not set, the default, `2015-10-08`, will be used.

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.

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 will be 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 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 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 ES:

```
{
   "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 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"
        }
    }]
}
```

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 my-rule
```

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 a 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. 143). The `awsIotSqlVersion` property allows you to specify which version of the AWS IoT SQL rules engine you want to use. When a new version is deployed, you can continue to use an older version or change your rule to use the new version. Your current rules will continue to use the version with which they were created.

The following JSON example shows 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"
        }
     }]
}
```

Current supported versions are:

- `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. The use of this version 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.
- Inter-object query support.
- Support to output an array as a top-level object.
- Adds the encode (*value*, *encodingScheme*) function, which can be applied on both JSON and non-JSON format data.

Inter-Object Queries

This feature allows you to query for an attribute in a JSON object. For example, given 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 }
    ]
}
```

And the following rule:

```
SELECT (SELECT v FROM e WHERE n = 'temperature') as temperature FROM 'my/topic'
```

The rule will generate the following output:

```
{"temperature": [{"v":22.5}]}
```

Using the same MQTT message, given a slightly more complicated rule such as:

```
SELECT get((SELECT v FROM e WHERE n = 'temperature'),1).v as temperature FROM 'topic'
```

The rule will generate the following output:

```
{"temperature":22.5}
```

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 will generate the following output:

```
[1,2,3,4]
```

Encode Function

Encodes the payload, which potentially might be non-JSON data, into its string representation based on the specified encoding scheme.

Troubleshooting a Rule

If you are having an issue with your rules, you should enable CloudWatch Logs. By analyzing your logs, you can determine whether the issue is authorization or whether, for example, a WHERE clause condition did not match. For more information about using Amazon CloudWatch Logs, see Setting Up CloudWatchLogs.

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 my-rule
```

AWS IoT Rule Actions

AWS IoT rule actions are used to specify what to do when a rule is triggered. You can define actions to write data to a DynamoDB database or an Amazon Kinesis stream or to invoke a Lambda function, and more. The following actions are supported:

- `cloudwatchAlarm` to change a CloudWatch alarm.
- `cloudwatchMetric` to capture a CloudWatch metric.
- `dynamoDB` to write data to a DynamoDB database.
- `dynamoDBv2` to write data to a DynamoDB database.
- `elasticsearch` to write data to a Amazon Elasticsearch Service domain.
- `firehose` to write data to an Amazon Kinesis Firehose stream.
- `kinesis` to write data to a Amazon Kinesis stream.
- `lambda` to invoke a Lambda function.
- `s3` to write data to a Amazon S3 bucket.
- `sns` to write data as a push notification.
- `sqs` to write data to an SQS queue.

- `republish` to republish the message on another MQTT topic.

Note
The AWS IoT rules engine does not currently retry delivery for messages that fail to be published to another service.

The following sections discuss each action in detail.

CloudWatch Alarm Action

The CloudWatch alarm action allows you to change CloudWatch alarm state. You can specify the state change reason and value in this call. When creating an AWS IoT rule with a CloudWatch alarm action, you must specify the following information:

roleArn

The IAM role that allows access to the CloudWatch alarm.

alarmName

The CloudWatch alarm name.

stateReason

Reason for the alarm change.

stateValue

The value of the alarm state. Acceptable values are `OK`, `ALARM`, `INSUFFICIENT_DATA`.

Note
Ensure the role associated with the rule has a policy granting the `cloudwatch:SetAlarmState` permission.

The following JSON example shows how to define a CloudWatch alarm action in an AWS IoT rule:

```
{
    "rule": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "actions": [{
            "cloudwatchAlarm": {
                "roleArn": "arn:aws:iam::123456789012:role/aws_iot_cw",
                "alarmName": "IotAlarm",
                "stateReason": "Temperature stabilized.",
                "stateValue": "OK"
            }
        }]
    }
}
```

For more information, see CloudWatch Alarms.

CloudWatch Metric Action

The CloudWatch metric action allows you to capture a CloudWatch metric. You can specify the metric namespace, name, value, unit, and timestamp. When creating an AWS IoT rule with a CloudWatch metric action, you must specify the following information:

roleArn

The IAM role that allows access to the CloudWatch metric.

metricNamespace

CloudWatch metric namespace name.

metricName

The CloudWatch metric name.

metricValue

The CloudWatch metric value.

metricUnit

The metric unit supported by CloudWatch.

metricTimestamp

An optional Unix timestamp.

Note
Ensure the role associated with the rule has a policy granting the `cloudwatch:PutMetricData` permission.

The following JSON example shows how to define a CloudWatch metric action in an AWS IoT rule:

```
{
    "rule": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "actions": [{
            "cloudwatchMetric": {
                "roleArn": "arn:aws:iam::123456789012:role/aws_iot_cw",
                "metricNamespace": "IotNamespace",
                "metricName": "IotMetric",
                "metricValue": "1",
                "metricUnit": "Count",
                "metricTimestamp": "1456821314"
            }
        }]
    }
}
```

For more information, see CloudWatch Metrics.

DynamoDB Action

The `dynamoDB` action allows you to write all or part of an MQTT message to a DynamoDB table. When creating a DynamoDB rule, you must specify the following information:

hashKeyType

The data type of the hash key (also called the partition key). Valid values are: `"STRING"` or `"NUMBER"`.

hashKeyField

The name of the hash key (also called the partition key).

hashKeyValue

The value of the hash key.

rangeKeyType

Optional. The data type of the range key (also called the sort key). Valid values are: `"STRING"` or `"NUMBER"`.

rangeKeyField

Optional. The name of the range key (also called the sort key).

rangeKeyValue

Optional. The value of the range key.

operation

Optional. The type of operation to be performed. This follows the substitution template, so it can be `${operation}`, but the substitution must result in one of the following: `INSERT`, `UPDATE`, or `DELETE`.

payloadField

Optional. The name of the field where the payload will be written. If this value is omitted, the payload is written to `payload` field.

table

The name of the DynamoDB table.

roleARN

The IAM role that allows access to the DynamoDB table. At a minimum, the role must allow the `dynamoDB:PutItem` IAM action.

The data written to the DynamoDB table is the result from the SQL statement of the rule. The `hashKeyValue` and `rangeKeyValue` fields are usually composed of expressions (for example, "${topic()}" or "${timestamp()}").

Note
Non-JSON data is written to DynamoDB as binary data. The DynamoDB console will display the data as Base64-encoded text.
Ensure the role associated with the rule has a policy granting the `dynamodb:PutItem` permission.

The following JSON example shows how to define a `dynamoDB` action in an AWS IoT rule:

```
{
    "rule": {
        "ruleDisabled": false,
        "sql": "SELECT * AS message FROM 'some/topic'",
        "description": "A test Dynamo DB rule",
        "actions": [{
            "dynamoDB": {
                "hashKeyField": "key",
                "roleArn": "arn:aws:iam::123456789012:role/aws_iot_dynamoDB",
                 "tableName": "my_ddb_table",
                 "hashKeyValue": "${topic()}",
                 "rangeKeyValue": "${timestamp()}",
                 "rangeKeyField": "timestamp"
             }
        }]
    }
```

```
}
```

For more information, see the Amazon DynamoDB Getting Started Guide.

DynamoDBv2 Action

The `dynamoDBv2` action allows you to write all or part of an MQTT message to a DynamoDB table. Each attribute in the payload is written to a separate column in the DynamoDB database. When creating a DynamoDB rule, you must specify the following information:

roleARN

The IAM role that allows access to the DynamoDB table. At a minimum, the role must allow the `dynamoDB:PutItem` IAM action.

tableName

The name of the DynamoDB table.

Note
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.

The data written to the DynamoDB table is the result from the SQL statement of the rule.

Note
Ensure the role associated with the rule has a policy granting the `dynamodb:PutItem` permission.

The following JSON example shows how to define a `dynamoDB` action in an AWS IoT rule:

```
{
    "rule": {
        "ruleDisabled": false,
        "sql": "SELECT * AS message FROM 'some/topic'",
        "description": "A test DynamoDBv2 rule",
        "actions": [{
            "dynamoDBv2": {
                "roleArn": "arn:aws:iam::123456789012:role/aws_iot_dynamoDBv2",
                "putItem": {
                    "tableName": "my_ddb_table"
                }
             }
        }]
    }
}
```

For more information, see the Amazon DynamoDB Getting Started Guide.

Amazon ES Action

The `elasticsearch` action allows you to write data from MQTT messages to an Amazon Elasticsearch Service domain. Data in Amazon ES can then be queried and visualized by using tools like Kibana. When you create an AWS IoT rule with an `elasticsearch` action, you must specify the following information:

endpoint

The endpoint of your Amazon ES domain.

index

The Amazon ES index where you want to store your data.

type

The type of document you are storing.

id

The unique identifier for each document.

Note
Ensure the role associated with the rule has a policy granting the `es:ESHttpPut` permission.

The following JSON example shows how to define an `elasticsearch` action in an AWS IoT rule:

```
{
  "rule":{
   "sql":"SELECT *, timestamp() as timestamp FROM 'iot/test'",
   "ruleDisabled":false,
    "actions":[
      {
        "elasticsearch":{
          "roleArn":"arn:aws:iam::123456789012:role/aws_iot_es",
          "endpoint":"https://my-endpoint",
         "index":"my-index",
          "type":"my-type",
          "id":"${newuuid()}"
        }
      }
    ]
  }
}
```

For more information, see the Amazon ES Developer Guide.

Firehose Action

A `firehose` action sends data from an MQTT message that triggered the rule to an Firehose stream. When creating a rule with a `firehose` action, you must specify the following information:

deliveryStreamName

The Firehose stream to which to write the message data.

roleArn

The IAM role that allows access to Firehose.

separator

A character separator that will be used to separate records written to the firehose stream. Valid values are: '\n' (newline), '\t' (tab), '\r\n' (Windows newline), ',' (comma).

Note
Make sure the role associated with the rule has a policy granting the `firehose:PutRecord` permission.

The following JSON example shows how to create an AWS IoT rule with a `firehose` action:

```
{
    "rule": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "actions": [{
            "firehose": {
                "roleArn": "arn:aws:iam::123456789012:role/aws_iot_firehose",
                "deliveryStreamName": "my_firehose_stream"
            }
        }]
    }
}
```

For more information, see the Firehose Developer Guide.

Kinesis Action

The `kinesis` action allows you to write data from MQTT messages into an Amazon Kinesis stream. When creating an AWS IoT rule with a `kinesis` action, you must specify the following information:

stream

The Amazon Kinesis stream to which to write data.

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()}").

Note
Ensure that the policy associated with the rule has the `kinesis:PutRecord` permission.

The following JSON example shows how to define a `kinesis` action in an AWS IoT rule:

```
{
    "rule": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "actions": [{
            "kinesis": {
                "roleArn": "arn:aws:iam::123456789012:role/aws_iot_kinesis",
                "streamName": "my_kinesis_stream",
                "partitionKey": "${topic()}"
            }
        }],
    }
}
```

For more information, see the Amazon Kinesis Developer Guide.

Lambda Action

A `lambda` action calls a Lambda function, passing in the MQTT message that triggered the rule. In order for AWS IoT to call a Lambda function, you must configure a policy granting the `lambda:InvokeFunction` permission to AWS IoT. Lambda functions use resource-based policies, so you must attach the policy to the Lambda function itself. Use the following CLI command to attach a policy granting `lambda:InvokeFunction` permission:

```
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 whose resource policy you are updating by adding a new permission.

--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` 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. In this case, we want to allow AWS IoT to invoke a Lambda function, so we specify `lambda:InvokeFunction`.

Note
If you add a permission for a AWS IoT principal without providing the source ARN, 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.

When creating a rule with a `lambda` action, you must specify the Lambda function to invoke when the rule is triggered.

The following JSON example shows a rule that calls a Lambda function:

```
{
    "rule": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "actions": [{
            "lambda": {
                "functionArn": "arn:aws:lambda:us-
east-1:123456789012:function:myLambdaFunction"
            }
        }]
    }
}
```

For more information, see the AWS Lambda Developer Guide.

Republish Action

The `republish` action allows you to republish the message that triggered the role to another MQTT topic. When creating a rule with a `republish` action, you must specify the following information:

topic

The MQTT topic to which to republish the message.

roleArn

The IAM role that allows publishing to the MQTT topic.

Note
Make sure the role associated with the rule has a policy granting the `iot:Publish` permission.

```
{
    "rule": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "actions": [{
            "republish": {
                "topic": "another/topic",
                "roleArn": "arn:aws:iam::123456789012:role/aws_iot_republish"
            }
        }]
    }
}
```

S3 Action

A `s3` action writes the data from the MQTT message that triggered the rule to an Amazon S3 bucket. When creating an AWS IoT rule with an `s3` action, you must specify the following information:

bucket

The Amazon S3 bucket to which to write data.

cannedacl

The Amazon S3 canned ACL that controls access to the object identified by the object key. For more information, see S3 Canned ACLs.

key

The path to the file where the data is written. For example, if the value of this argument is "${topic()}/${timestamp()}", the topic the message was sent to is "this/is/my/topic,", and the current timestamp is 1460685389 the data will be written to a file called "1460685389" in the "this/is/my/topic" folder on Amazon S3.

Note
Using a static key will result in a single file in Amazon S3 being overwritten for each invocation of the rule. More common use cases are to use the message timestamp or another unique message identifier, so that a new file will be saved in Amazon S3 for each message received.

roleArn

The IAM role that allows access to the Amazon S3 bucket.

Note
Make sure the role associated with the rule has a policy granting the `s3:PutObject` permission.

The following JSON example shows how to define an `s3` action in an AWS IoT rule:

```
{
    "rule": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "actions": [{
            "s3": {
                "roleArn": "arn:aws:iam::123456789012:role/aws_iot_s3",
                "bucketName": "my-bucket",
                "key": "${topic()}/${timestamp()}"
            }
        }]
    }
}
```

For more information, see the Amazon S3 Developer Guide.

SNS Action

A `sns` action sends the data from the MQTT message that triggered the rule as an SNS push notification. When creating a rule with an `sns` action, you must specify the following information:

messageFormat

The message format. Accepted values are "JSON" and "RAW". The default value of the attribute is "RAW". SNS uses this setting to determine if the payload should be parsed and relevant platform-specific parts of the payload should be extracted.

roleArn

The IAM role that allows access to SNS.

targetArn

The SNS topic or individual device to which the push notification will be sent.

Note
Make sure the policy associated with the rule has the `sns:Publish` permission.

The following JSON example shows how to define an `sns` action in an AWS IoT rule:

```
{
    "rule": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "actions": [{
            "sns": {
                "targetArn": "arn:aws:sns:us-east-1:123456789012:my_sns_topic",
                "roleArn": "arn:aws:iam::123456789012:role/aws_iot_sns"
            }
        }]
    }
}
```

For more information, see the Amazon SNS Developer Guide.

SQS Action

A `sqs` action sends data from the MQTT message that triggered the rule to an SQS queue. When creating a rule with an `sqs` action, you must specify the following information:

queueUrl

The URL of the SQS queue to which to write the data.

useBase64

Set to `true` if you want the MQTT message data to be Base64-encoded before writing to the SQS queue; otherwise, set to `false`.

roleArn

The IAM role that allows access to the SQS queue.

Note
Make sure the role associated with the rule has a policy granting the `sqs:SendMessage` permission.

The following JSON example shows how to create an AWS IoT rule with an `sqs` action:

```
{
    "rule": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "actions": [{
            "sqs": {
                "queueUrl": "https://sqs.us-east-1.amazonaws.com/123456789012/
my_sqs_queue",
                "roleArn": "arn:aws:iam::123456789012:role/aws_iot_sqs",
                "useBase64": false
            }
        }]
    }
}
```

For more information, see the Amazon SQS Developer Guide.

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 incoming payload and performs transformations.

FROM

Required. The MQTT topic filter from which the rule will receive messages.

WHERE

Optional. Adds conditional logic that determines if a rule is evaluated and its actions are executed.

An example SQL statement looks like this:

```
SELECT color AS rgb FROM 'a/b' 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 `'a/b'` topic, the rule is triggered and the SQL statement is evaluated. The SQL statement extracts the value of the `rgb` 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. 148).

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`s with double precision rather than 34-digit precision.
`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 'a/b'"`

Type	Meaning

`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`.
	`{"foo":null, "bar":undefined}`
	is serialized to JSON as:
	`{"foo":null}`
	Similarly, `Undefined` 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) will return `Undefined`.

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
`Int`	A `Decimal` with no decimal point.
`Decimal`	The source value.
`Boolean`	`Undefined`. (You can explicitly use the cast function to transform true = 1.0, false = 0.0.)
`String`	The SQL engine will try to parse the string as a `Decimal`. We will attempt to parse strings matching the regular expression:`^-?\d+(\.\d+)?((?i)E-?\d+)?$`. "0", "-1.2", "5E-12" are all examples of strings that would be automatically converted to `Decimal`s.
Array	`Undefined`.
Object	`Undefined`.
Null	`Null`.
Undefined	`Undefined`.

To Int

Argument Type	Result
`Int`	The source value.
`Decimal`	The source value rounded to the nearest `Int`.
`Boolean`	`Undefined`. (You can explicitly use the cast function to transform true = 1.0, false = 0.0.)
`String`	The SQL engine will try to parse the string as a `Decimal`. We will attempt to parse strings matching the regular expression:`^-?\d+(\.\d+)?((?i)E-?\d+)?$`. "0", "-1.2", "5E-12" are all examples of strings that would automatically be converted to `Decimal`s. We will attempt to convert the `String` to a `Decimal`, and then truncate 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 will be `Undefined`.
Array	`Undefined`.
Object	`Undefined`.
Null	`Undefined`.
Undefined	`Undefined`.

To String

Argument Type	Result
`Int`	A string representation of the `Int` in standard notation.

Argument Type	Result
`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 will be a comma-separated list, enclosed in square brackets. `String`s will be quoted. `Decimal`s, `Int`s, `Boolean`s and `Null` will not.
Object	The object serialized to JSON. The resultant string will be a comma-separated list of key-value pairs and will begin and end with curly braces. `String`s will be quoted. `Decimal`s, `Int`s, `Boolean`s and `Null` will not.
Null	`Undefined`.
Undefined	Undefined.

Operators

The following operators can be used in SELECT, FROM, and WHERE clauses.

AND operator

Returns a `Boolean` result. Performs a logical AND operation. Returns true if left and right operands are true; returns false otherwise. `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 *`boolean`* `AND` *`boolean`*.
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; returns false otherwise. `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 `Boolean`s and processed normally as *`boolean`* `OR` *`boolean`*.
Other Value	Other Value	`Undefined`.

NOT operator

Returns a `Boolean` result. Performs a logical NOT operation. Returns true if the operand is false; returns true otherwise. 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.
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; returns false otherwise.

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; returns false otherwise.

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 `Decimal`s 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 `Decimal`s 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 `Decimal`s 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 `String`s convert to `Decimal`s 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 2015-10-8 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 2015-10-8 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 2015-10-8 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`.
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 2015-10-8 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 2015-10-8 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`.
Array	`Undefined`.
Object	`Undefined`.
Null	`Undefined`.

Argument Type	Result
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 2015-10-8 and later.

Example: `atan2(1, 0)` = 1.5707963267948966

Argument Type	Argument Type	Result
`Int`/`Decimal`	`Int`/`Decimal`	`Decimal` (with double axis and the specified
`Int`/`Decimal`/`String`	`Int`/`Decimal`/`String`	`Decimal`, the inverse t string cannot be conv
Other Value	Other Value	`Undefined`.

bitand(Int, Int)

Performs a bitwise AND on the bit representations of the two `Int`(-converted) arguments. Supported by SQL version 2015-10-8 and later.

Example: `bitand(13, 5)` = 5

Argument Type	Argument Type	Result
`Int`	`Int`	`Int`, a bitwise AND of
`Int`/`Decimal`	`Int`/`Decimal`	`Int`, a bitwise AND of numbers are rounded the arguments canno is `Undefined`.
`Int`/`Decimal`/`String`	`Int`/`Decimal`/`String`	`Int`, a bitwise AND of are converted to `Deci` the nearest `Int`. If the `Undefined`.
Other Value	Other Value	`Undefined`.

bitor(Int, Int)

Performs a bitwise OR of the bit representations of the two arguments. Supported by SQL version 2015-10-8 and later.

Example: `bitor(8, 5)` = 13

Argument Type	Argument Type	Result
`Int`	`Int`	`Int`, the bitwise OR o
`Int/Decimal`	`Int/Decimal`	`Int`, the bitwise OR o numbers are rounded conversion fails, the r
`Int/Decimal/String`	`Int/Decimal/String`	`Int`, the bitwise OR o converted to `Decimal` `Int`. If the conversion
Other Value	Other Value	`Undefined`.

bitxor(Int, Int)

Performs a bitwise XOR on the bit representations of the two `Int`(-converted) arguments. Supported by SQL version 2015-10-8 and later.

Example:`bitor(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
Other Value	Other Value	`Undefined`.

bitnot(Int)

Performs a bitwise NOT on the bit representations of the `Int`(-converted) argument. Supported by SQL version 2015-10-8 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. `String`s are converted to `Decimal`s 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/from `Boolean`s. If we cannot determine how to cast one type to another, the result is `Undefined`. Supported by SQL version 2015-10-8 and later. Format: cast(*`value`* as *`type`*).

Example:

`cast(true as Decimal)` = 1.0

The following keywords may appear after "as" when calling `cast`:

Keyword	Result
`Decimal`	Casts value to `Decimal`.
Bool	Casts value to `Boolean`.
`Boolean`	Casts value to `Boolean`.
`String`	Casts value to `String`.
Nvarchar	Casts value to `String`.
Text	Casts value to `String`.
Ntext	Casts value to `String`.
varchar	Casts value to `String`.
`Int`	Casts value to `Int`.
`Int`	Casts value to `Int`.

Casting rules:

Cast to Decimal

Argument Type	Result
`Int`	A `Decimal` with no decimal point.
`Decimal`	The source value.
`Boolean`	true = 1.0, false = 0.0.
`String`	Will try to parse the string as a `Decimal`. We will attempt to parse strings matching the regex: ^-?\d+(\.\d+)?((?i)E-?\d+)?$. "0", "-1.2", "5E-12" are all examples of `String`s that would be converted automatically to `Decimal`s.
Array	`Undefined`.
Object	`Undefined`.
Null	`Undefined`.
Undefined	`Undefined`.

Cast to Int

Argument Type	Result
`Int`	The source value.
`Decimal`	The source value, rounded down to the nearest `Int`.
`Boolean`	true = 1.0, false = 0.0.
`String`	Will try to parse the string as a `Decimal`. We will attempt to parse strings matching the regex: ^-?\d+(\.\d+)?((?i)E-?\d+)?$. "0", "-1.2", "5E-12" are all examples of `String`s that would be converted automatically to `Decimal`s. Will attempt 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`.

Argument Type	Result
Array	The array serialized to JSON. The result string will be a comma-separated list enclosed in square brackets. `String`s are quoted. `Decimal`s, `Int`s, `Boolean`s are not.
Object	The object serialized to JSON. The JSON string will be a comma-separated list of key-value pairs and will begin and end with curly braces. `String`s are quoted. `Decimal`s, `Int`s, `Boolean`s and `Null` are not.
Null	`Undefined`.
Undefined	`Undefined`.

ceil(Decimal)

Rounds the given `Decimal` up to the nearest `Int`. Supported by SQL version 2015-10-8 and later.

Examples:

`ceil(1.2)` = 2

`ceil(11.2)` = -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 2015-10-8 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

Argument Type	Result
	`Int`. If the argument is not a valid ASCII value, the result is `Undefined`.
`Boolean`	`Undefined`.
`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 `Undefined` if the message wasn't sent over MQTT. Supported by SQL version 2015-10-8 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 2015-10-8 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 `Array`s, and at least one argument is a `String`, the result

Number of Arguments	Result
	is the concatenation of the `String` representations of all the arguments. Arguments will be converted to `String`s 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 2015-10-8 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 2015-10-8 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`.

Argument Type	Result
`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`.
Object	`Undefined`.
Null	`Undefined`.
Undefined	`Undefined`.

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. 158). In addition, 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 will first be 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 2015-10-8 and later.

Example: `endswith("cat","at")` = true.

argument Type 1	argument Type 2	Result
`String`	`String`	True if the first argun otherwise, false.
Other Value	Other Value	Both arguments are c standard conversion r ends in the second ar argument is `Null` or `U`

exp(Decimal)

Returns e raised to the `Decimal` argument. `Decimal` arguments are rounded to double precision before function application. Supported by SQL version 2015-10-8 and later.

Example: `exp(1)` = e.

Argument Type	Result
`Int`	`Decimal` (with double precision), e ^ argument.
`Decimal`	`Decimal` (with double precision), e ^ argument.
`String`	`Decimal` (with double precision), e ^ argument. If the `String` cannot be converted to a `Decimal`, the result is `Undefined`.
Other Value	`Undefined`.

get

Extracts a value from a collection-like type (Array, String, Object). No conversion will be applied to the first argument. Conversion applies as documented in the table to the second argument. Supported by SQL version 2015-10-8 and later.

Examples:

`get(["a", "b", "c"], 1)` = "b"

`get({"a":"b"}, "a")` = "b"

`get("abc", 1)` = "b"

argument Type 1	argument Type 2	Result
Array	Any Type (converted to `Int`)	The item at the 0-bas the second argument is unsuccessful, the re is outside the bounds array.length), the resu
String	Any Type (converted to `Int`)	The character at the (by the second argum conversion is unsucce index is outside the b string.length), the res
Object	`String` (no conversion is applied)	The value stored in th corresponding to the argument.
Other Value	Any Value	`Undefined`.

get_thing_shadow(thingName, roleARN)

Returns the 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.

roleArn

String: A role ARN with `iot:GetThingShadow` permission.

Example:

```
SELECT * from 'a/b'

WHERE get_thing_shadow("MyThing","arn:aws:iam::123456789012:role/
AllowsThingShadowAccess") .state.reported.alarm = 'ON'
```

Hashing Functions

We provide 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 2015-10-8 and later.

Examples:

`md2("hello")` = "a9046c73e00331af68917d3804f70655"

`md5("hello")` = "5d41402abc4b2a76b9719d911017c592"

hsin(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 2015-10-8 and later.

Example: `sinh(2.3)` = 4.936961805545957

Argument Type	Result
`Int`	`Decimal` (with double precision), the hyperbolic sine of the argument.
`Decimal`	`Decimal` (with double precision), the hyperbolic sine of the argument.
`Boolean`	`Undefined`.
`String`	`Decimal` (with double precision), the hyperbolic sine of the argument. If the string cannot be converted to a `Decimal`, the result is `Undefined`.
Array	`Undefined`.
Object	`Undefined`.
Null	`Undefined`.
Undefined	`Undefined`.

htan(Decimal)

Returns the hyperbolic tangent of a number in radians. `Decimal` values are rounded to double precision before function application. Supported by SQL version 2015-10-8 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`.

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 2015-10-8 and later.

Examples:

`indexof("abcd", "bc")` = 1

isNull()

Returns whether the argument is the `Null` value. Supported by SQL version 2015-10-8 and later.

Examples:

`isNull(5)` = false.

`isNull(Null)` = true.

Argument Type	Result
`Int`	false
`Decimal`	false
`Boolean`	false

Argument Type	Result
`String`	false
`Array`	false
`Object`	false
`Null`	true
`Undefined`	false

isUndefined()

Returns whether the argument is `Undefined`. Supported by SQL version 2015-10-8 and later.

Examples:

`isUndefined(5)` = false.

`isNull(floor([1,2,3]))` = true.

Argument Type	Result
`Int`	false
`Decimal`	false
`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 2015-10-8 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 2015-10-8 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 2015-10-8 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 `String`s using the standard conversion rules. Supported by SQL version 2015-10-8 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 will be set to the nearest valid value (0 or 1000). Supported by SQL version 2015-10-8 and later.

Examples:

`lpad("hello", 2)` = " hello".

`lpad(1, 3)` = " 1"

argument Type 1	argument Type 2	Result
`String`	`Int`	`String`, the provided a number of spaces e
`String`	`Decimal`	The `Decimal` argumer nearest `Int` and the s specified number of s
`String`	`String`	The second argumen which is rounded dow `String` is padded wit the left. If the second an `Int`, the result is `U`
Other Value	`Int`/`Decimal`/`String`	The first value will be standard conversions be applied on that `St` result is `Undefined`.
Any Value	Other Value	`Undefined`.

ltrim(String)

Removes all leading whitespace (tabs and spaces) from the provided `String`. Supported by SQL version 2015-10-8 and later.

Example:

`Ltrim(" h i ")` = "hi ".

Argument Type	Result
`Int`	The `String` representation of the `Int` with all leading whitespace removed.
`Decimal`	The `String` representation of the `Decimal` with all leading whitespace removed.
`Boolean`	The `String` representation of the boolean ("true" or "false") with all leading whitespace removed.

Argument Type	Result
`String`	The argument with all leading whitespace removed.
Array	The `String` representation of the `Array` (using standard conversion rules) with all leading whitespace removed.
Object	The `String` representation of the Object (using standard conversion rules) with all leading whitespace removed.
Null	`Undefined`.
Undefined	`Undefined`.

machinelearning_predict(modelId)

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 2015-10-8 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 will be 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. Supported by SQL version 2015-10-8 and later. You can also use "%" as an infix operator for the same modulo functionality. Supported by SQL version 2015-10-8 and later.

Example: `mod(8, 3)` = 2.

Left Operand	Right Operand	Output
`Int`	`Int`	`Int`, the first argume
`Int/Decimal`	`Int/Decimal`	`Decimal`, the first arg
`String/Int/Decimal`	`String/Int/Decimal`	If all strings convert t argument modulo th `Undefined`.
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 2015-10-8 and later.

Example: `Nanvl(8, 3)` = 8.

argument Type 1	argument Type 2	Output
Undefined	Any Value	The second argumen
Null	Any Value	The second argumen
`Decimal` (NaN)	Any Value	The second argumen
`Decimal` (not NaN)	Any Value	The first argument.
Other Value	Any Value	The first argument.

newuuid()

Returns a random 16-byte UUID. Supported by SQL version 2015-10-8 and later.

Example: `uuid()` = `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 2015-10-8 and later.

Examples:

`numbytes("hi")` = 2

`numbytes("€")` = 3

principal()

Returns the X.509 certificate fingerprint or thing name, depending on which endpoint, MQTT or HTTP, received the request. Supported by SQL version 2015-10-8 and later.

Example:

`principal()` = "ba67293af50bf2506f5f93469686da660c7c844e7b3950bfb16813e0d31e9373"

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 2015-10-8 and later. Supported by SQL version 2015-10-8 and later.

Example: `power(2, 5)` = 32.0.

argument Type 1	argument Type 2	Output
`Int/Decimal`	`Int/Decimal`	A `Decimal` (with doub raised to the second a
`Int/Decimal/String`	`Int/Decimal/String`	A `Decimal` (with doub raised to the second a converted to `Decimal` to `Decimal`, the result
Other Value	Other Value	`Undefined`.

rand()

Returns a pseudorandom, uniformly distributed double between 0.0 and 1.0. Supported by SQL version 2015-10-8 and later.

Example:

`rand()` = 0.8231909191640703

regexp_matches(String, String)

Returns whether the first argument contains a match for the second argument (regex).

Example:

`Regexp_matches("aaaa", "a{2,}")` = true.

`Regexp_matches("aaaa", "b")` = false.

First argument:

Argument Type	Result
`Int`	The `String` representation of the `Int`.

Argument Type	Result
`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 may or may not be a valid regular expression. If the (converted) argument is not valid regex, the result is `Undefined`.

Third argument:

Must be a valid regex replacement string. (Can reference capture groups.) Non-string types will be converted to `String` using the standard conversion rules. If the (converted) argument is not a valid regex replacement string, 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 2015-10-8 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).

Argument Type	Result
Null	`Undefined`.
Undefined	`Undefined`.

Second argument:

Must be a valid regex expression. Non-string types are converted to `String`s using the standard conversion rules. Depending on the type, the resultant string may or may 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 will be converted to `String`s 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 2nd parameter (regex) in the first parameter. Reference capture groups with "$". Supported by SQL version 2015-10-8 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).
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`s using the standard conversion rules. Depending on the type, the resultant string may or may 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 will be converted to `String` using the standard conversion rules. If the argument is not a valid regex replacement string, the result is `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 will be set to the nearest valid value (0 or 1000). Supported by SQL version 2015-10-8 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 will be rounded down to the nearest `Int` and the string is padded on the right side with a number of spaces equal

argument Type 1	argument Type 2	Result
		to the provided `Int`.
`String`	`String`	The second argument will be 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 will be converted to a `String` using the standard conversions, and the rpad function will be 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 2015-10-8 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 whitespace (tabs and spaces) from the provided `String`. Supported by SQL version 2015-10-8 and later.

Examples:

`rtrim(" h i ")` = " h i"

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").
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`

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 2015-10-8 and later.

Examples:

`sign(-7)` = -1.

`sign(0)` = 0.

`sign(13)` = 1.

Argument Type	Result
`Int`	`Int`, the sign of the `Int` value.
`Decimal`	`Int`, the sign of the `Decimal` value.
`String`	`Int`, the sign of the `Decimal` value. The string is converted to a `Decimal` value, and the sign of the `Decimal` value is returned. If the `String` cannot be converted to a `Decimal`, the result is `Undefined`. Supported by SQL version 2015-10-8 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 2015-10-8 and later.

Example: `sin(0)` = 0.0

Argument Type	Result
`Int`	`Decimal` (with double precision), the sine of the argument.
`Decimal`	`Decimal` (with double precision), the sine of the argument.
`Boolean`	`Undefined`.
`String`	`Decimal` (with double precision), the sine of the argument. If the string cannot be converted to a `Decimal`, the result is `Undefined`.
Array	`Undefined`.
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 will be set to zero. Indices that are greater than the `String` length will be 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 will be 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 2015-10-8 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)` = "rue".

`substring(false, -2.411E247)` = "false".

`substring("012345", 1, 3)` = "12".

`substring("012345", -50, 50)` = "012345".

`substring("012345", 3, 1)` = "".

sqrt(Decimal)

Returns the square root of a number. `Decimal` arguments are rounded to double precision before function application. Supported by SQL version 2015-10-8 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 2015-10-8 and later.

Example:

`startswith("ranger","ran")` = true

argument Type 1	argument Type 2	Result
`String`	`String`	Whether the first stri
Other Value	Other Value	Both arguments will standard conversion string starts with the `Null` or `Undefined`, th

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 2015-10-8 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, one-based topic segment. For the topic `foo/bar/baz`, topic(1) will return `foo`, topic(2) will return `bar`, and so on. Supported by SQL version 2015-10-8 and later.

Examples:

`topic()` = "things/myThings/thingOne"

`topic(1)` = "things"

tan(Decimal)

Returns the tangent of a number in radians. `Decimal` values are rounded to double precision before function application. Supported by SQL version 2015-10-8 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.
`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`.

traceid()

Returns the trace ID (UUID) of the MQTT message, or `Undefined` if the message wasn't sent over MQTT. Supported by SQL version 2015-10-8 and later.

Example:

`traceid()` = "12345678-1234-1234-1234-123456789012"

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 will be set to zero. If the second argument is greater than 34, it will be set to 34. Trailing zeroes are stripped from the result. Supported by SQL version 2015-10-8 and later.

Examples:

`trunc(2.3, 0)` = 2.

`trunc(2.3123, 2` = 2.31.

`trunc(2.888, 2)` = 2.88.

`(2.00, 5)` = 2.

argument Type 1	argument Type 2	Result
`Int`	`Int`	The source value.
`Int`/`Decimal`	`Int`/`Decimal`	The first argument is by the second argum `Int`, will be rounded
`Int`/`Decimal`/`String`	The first argument is truncated to the length described by the second argument. The second argument, if not an `Int`, will be rounded down to the nearest `Int`. `String`s are converted to `Decimal` values. If the string conversion fails, the result is `Undefined`.	
Other Value	`Undefined`.	

trim(String)

Removes all leading and trailing whitespace from the provided `String`. Supported by SQL version 2015-10-8 and later.

Example:

`Trim(" hi ")` = "hi"

Argument Type	Result
`Int`	The `String` representation of the `Int` with all leading and trailing whitespace removed.
`Decimal`	The `String` representation of the `Decimal` with all leading and trailing whitespace removed.
`Boolean`	The `String` representation of the `Boolean` ("true" or "false") with all leading and trailing whitespace removed.
`String`	The `String` with all leading and trailing whitespace removed.

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`.

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 2015-10-8 and later.

Examples:

`upper("hello")` = "HELLO"

`upper(["hello"])` = "[\"HELLO\"]"

SELECT Clause

The AWS IoT SELECT clause is essentially the same as the ANSI SQL SELECT clause, with some minor differences.

You can use the SELECT clause to extract information from incoming MQTT messages. `SELECT *` can be used to retrieve the entire incoming message payload. For example:

```
Incoming payload published on topic 'a/b': {"color":"red", "temperature":50}
SQL statement: SELECT * FROM 'a/b'
Outgoing payload: {"color":"red", "temperature":50}
```

If the payload is a JSON object, you can reference keys in the object. Your outgoing payload will contain the key-value pair. For example:

```
Incoming payload published on topic 'a/b': {"color":"red", "temperature":50}
SQL statement: SELECT color FROM 'a/b'
Outgoing payload: {"color":"red"}
```

You can use the AS keyword to rename keys. For example:

```
Incoming payload published on topic 'a/b':{"color":"red", "temperature":50}
SQL:SELECT color AS my_color FROM 'a/b'
Outgoing payload: {"my_color":"red"}
```

You can select multiple items by separating them with a comma. For example:

```
Incoming payload published on topic 'a/b': {"color":"red", "temperature":50}
SQL: SELECT color as my_color, temperature as farenheit FROM 'a/b'
Outgoing payload: {"my_color":"red","farenheit":50}
```

You can select multiple items including '*' to add items to the incoming payload. For example:

```
Incoming payload published on topic 'a/b': {"color":"red", "temperature:50}
SQL: SELECT *, 15 as speed FROM 'a/b'
Outgoing payload: {"color":"red", "temperature:50, speed:15}"
```

You can use the `"VALUE"` keyword to produce outgoing payloads that are not JSON objects. You may only select one item. For example:

```
Incoming payload published on topic 'a/b': {"color":"red", "temperature":50}
SQL: SELECT VALUE color FROM 'a/b'
Outgoing payload: "red"
```

You can use `'.'` syntax to drill into nested JSON objects in the incoming payload. For example:

```
Incoming payload published on topic 'a/b': {"color":{"red":255,"green":0,"blue":0},
 "temperature":50}
SQL: SELECT color.red as red_value FROM 'a/b'
Outgoing payload: {"red_value":255}
```

You can use functions (see Functions (p. 167)) to transform the incoming payload. Parentheses can be used for grouping. For example:

```
Incoming payload published on topic 'a/b': {"color":"red", "temperature":50}
SQL: SELECT (temperature - 32) * 5 / 9 AS celsius, upper(color) as my_color FROM 'a/b'
Outgoing payload: {"celsius":10,"my_color":"RED"}
```

Working with Binary Payloads

When used in a `SELECT` clause the * operator refers to the original message. Usually it refers to the original message as a JSON object, but other times as the raw binary of the original message.

All of these rules must be true to refer to * as binary:

1. The SQL statement and templates must not refer to other JSON names, other than *.
2. The SELECT statement must have only a single selected item, * (i.e. SELECT * FROM 'a/b').

Binary Payload Examples

The following `SELECT` clause can be used with binary payloads because it doesn't refer to any JSON names.

```
SELECT * FROM 'a/b'
```

The following `SELECT` can not be used with binary payloads because it referrs to `device_type` in the WHERE clause.

```
SELECT * FROM 'a/b' WHERE device_type = 'thermostat'
```

The following `SELECT` can not be used with binary payloads because it violates rule #2.

```
SELECT *, timestamp() AS timestamp FROM 'a/b'
```

The following `SELECT` can be used with binary payloads because it doesn't violate either rule #1 or #2.

```
SELECT * FROM 'a/b' WHERE timestamp() % 12 = 0
```

The following AWS IoT rule can not be used with payloads because it violates rule #1.

```
{
    "sql": "SELECT * FROM 'a/b'"
    "actions": [{
        "republish": {
            "topic":"device/${device_id}"
         }
     }]
}
```

FROM Clause

The FROM clause subscribes your rule to a topic or topic filter. A topic filter allows you to subscribe to a group of similar topics.

Example:

Incoming payload published on topic `'a/b'`: `{temperature: 50}`

Incoming payload published on topic `'a/c'`: `{temperature: 50}`

SQL: `"SELECT temperature AS t FROM 'a/b'"`.

The rule is subscribed to `'a/b'`, so the incoming payload is passed to the rule, and the outgoing payload (passed to the rule actions) is: `{t: 50}`. The rule is not subscribed to `'a/c'`, so the rule is not triggered for the message published on `'a/c'`.

You can use the # wildcard character to match any subpath in a topic filter:

Example:

Incoming payload published on topic `'a/b'`: `{temperature: 50}`.

Incoming payload published on topic `'a/c'`: `{temperature: 60}`.

Incoming payload published on topic `'a/e/f'`: `{temperature: 70}`.

Incoming payload published on topic `'b/x'`: `{temperature: 80}`.

SQL: `"SELECT temperature AS t FROM 'a/#'"`.

The rule is subscribed to any topic beginning with `'a'`, so it is executed three times, sending outgoing payloads of `{t: 50} (for a/b), {t: 60}` (for a/c), and `{t: 70}` (for a/e/f) to its actions. It is not subscribed to `'b/x'`, so the rule will not be triggered for the {temperature: 80} message.

You can use the '+' character to match any one particular path element:

Example:

Incoming payload published on topic `'a/b'`: `{temperature: 50}`.

Incoming payload published on topic `'a/c'`: `{temperature: 60}`.

Incoming payload published on topic `'a/e/f'`: `{temperature: 70}`.

Incoming payload published on topic `'b/x'`: `{temperature: 80}`.

SQL: `"SELECT temperature AS t FROM 'a/+'"`.

The rule is subscribed to all topics with two path elements where the first element is `'a'`. The rule is executed for the messages sent to `'a/b'` and `'a/c'`, but not `'a/e/f'` or `'b/x'`.

You can use functions and operators in the WHERE clause. In the WHERE clause, you cannot reference any aliases created with the AS keyword in the SELECT. (The WHERE clause is evaluated first, to determine if the SELECT clause is evaluated.)

WHERE Clause

The WHERE clause determines if a rule is evaluated for a message sent to an MQTT topic to which the rule is subscribed. If the WHERE clause evaluates to true, the rule is evaluated; otherwise, the rule is not evaluated.

Example:

Incoming payload published on `a/b`: `{"color":"red", "temperature":40}`.

SQL: `SELECT color AS my_color FROM 'a/b' WHERE temperature > 50 AND color <> 'red'`.

In this case, the rule would not be evaluated; there would be no outgoing payload; and rules actions would not be triggered.

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.)

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 only available when using SQL Version 2016-03-23 or newer.

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 `a/b`: `"{lat_long: [47.606,-122.332]}"`

SQL statement: `SELECT {'latitude': get(lat_long, 0),'longitude':get(lat_long, 1)} as lat_long FROM 'a/b'`

The resulting outgoing payload would be: `{'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 `a/b`: `{lat: 47.696, long: -122.332}`

SQL statement: `SELECT [lat,long] as lat_long FROM 'a/b'`

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, or if/else statements.

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 each `t[i]` expression. If a match is found, the corresponding `r[i]` expression becomes the result of the case statement. If there is more than one possible match, the first match is selected. If there are no matches, the else statement's `re` is used as the result. If there is no match and no else statement, the result of the case statement is `Undefined`. For example:

Incoming payload published on topic `a/b`: `{"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 'a/b'`

The resulting output payload would be: `{"instructions":"caution"}`.

Case statements require at least one WHEN clause. An ELSE clause is not required.

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 'a/b'
```

`*` 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 'a/b'
```

Applying a Function to an Attribute Value

The following is an example JSON payload that could 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 or WHERE clauses. For more information about supported expressions, see AWS IoT SQL Reference (p. 158).

Substitution templates appear in the SELECT clause within a rule:

```
{
    "sql": "SELECT *, topic() AS topic 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:

```
{
    "deviceid" : "iot123",
    "temp" : 54.98,
    "humidity" : 32.43,
    "coords" : {
        "latitude" : 47.615694,
        "longitude" : -122.3359976
}
```

Here is the output of the rule:

```
{
    "coords":{
        "longitude":-122.3359976,
        "latitude":47.615694
    },
    "humidity":32.43,
    "temp":54.98,
    "deviceid":"iot123",
    "topic":"my/iot/topic"
}
```

Device Shadows for AWS IoT

A *thing shadow* (sometimes referred to as a device shadow) is a JSON document that is used to store and retrieve current state information for a thing (device, app, and so on). The Thing Shadows service maintains a thing shadow for each thing you connect to AWS IoT. You can use thing shadows to get and set the state of a thing over MQTT or HTTP, regardless of whether the thing is connected to the Internet. Each thing shadow is uniquely identified by its name.

Contents

Device Shadows Data Flow

The Thing Shadows services acts as an intermediary, allowing devices and applications to retrieve and update thing shadows.

To illustrate how devices and applications communicate with the Thing Shadows service, this section walks you through the use of the AWS IoT MQTT client and the AWS CLI to simulate communication between an internet-connected light bulb, an application, and the Thing Shadows service.

The Thing Shadows service uses MQTT topics to facilitate communication between applications and devices. To see how this works, use the AWS IoT MQTT client to subscribe to the following MQTT topics with QoS 1:

$aws/things/myLightBulb/shadow/update/accepted

The Thing Shadows service sends messages to this topic when an update is successfully made to the thing shadow.

$aws/things/myLightBulb/shadow/update/rejected

The Thing Shadows service sends messages to this topic when an update to the thing shadow is rejected.

$aws/things/myLightBulb/shadow/update/delta

The Thing Shadows service sends messages to this topic when a difference is detected between the reported and desired sections of the thing shadow. For more information, see /update/delta (p. 228).

$aws/things/myLightBulb/shadow/get/accepted

The Thing Shadows service sends messages to this topic when a request for the thing shadow is made successfully.

$aws/things/myLightBulb/shadow/get/rejected

The Thing Shadows service sends messages to this topic when a request for the thing shadow is rejected.

$aws/things/myLightBulb/shadow/delete/accepted

The Thing Shadows service sends messages to this topic when the thing shadow is deleted.

$aws/things/myLightBulb/shadow/delete/rejected

The Thing Shadows service sends messages to this topic when a request to delete the thing shadow is rejected.

$aws/things/myLightBulb/shadow/update/documents

The Thing Shadows service publishes a state document to this topic whenever an update to the thing shadow is successfully performed.

To learn more about all of the MQTT topics used by the Thing Shadows service, see Device Shadow MQTT Topics (p. 225).

Note
We recommend you subscribe to the `.../rejected` topics to see any errors sent by the Thing Shadows service.

When the light bulb comes online, it sends its current state to the Thing Shadows service by sending an MQTT message to the `$aws/things/myLightBulb/shadow/update` topic.

To simulate this, use the AWS IoT MQTT client to publish the following message to the `$aws/things/myLightbulb/shadow/update` topic:

```
{
    "state": {
        "reported": {
            "color": "red"
        }
    }
}
```

This message sets the color of the light bulb to "red".

The Thing Shadows service responds by sending the following message to the `$aws/things/myLightBulb/shadow/update/accepted` topic:

```
{
  "messageNumber": 4,
  "payload": {
    "state": {
      "reported": {
        "color": "red"
      }
```

```
    },
    "metadata": {
      "reported": {
        "color": {
          "timestamp": 1469564492
        }
      }
    },
    "version": 1,
    "timestamp": 1469564492
  },
  "qos": 0,
  "timestamp": 1469564492848,
  "topic": "$aws/things/myLightBulb/shadow/update/accepted"
}
```

This message indicates the Thing Shadows service received the UPDATE request and updated the thing shadow. If the thing shadow doesn't exist, it is created. Otherwise, the thing shadow is updated with the data in the message. If you don't see a message published to `$aws/things/myLightBulb/shadow/update/accepted`, check the subscription to `$aws/things/myLightBulb/shadow/update/rejected` to see any error messages.

In addition, the Thing Shadows service publishes the following message to the `$aws/things/myLightBulb/shadow/update/documents` topic.

```
{
    "previous":null,
    "current":{
        "state":{
            "reported":{
                "color":"red"
            }
         },
         "metadata":{
            "reported":{
                "color":{
                    "timestamp":1483467764
                }
            }
         },
         "version":1
     },
     "timestamp":1483467764
}
```

Messages are published to the `/update/documents` topic whenever an update to the thing shadow is successfully performed. For more information of the contents of messages published to this topic, see Device Shadow MQTT Topics (p. 225).

An application that interacts with the light bulb comes online and requests the light bulb's current state. The application sends an empty message to the `$aws/things/myLightBulb/shadow/get` topic. To simulate this, use the AWS IoT MQTT client to publish an empty message ("") to the `$aws/things/myLightBulb/shadow/get` topic.

The Thing Shadows service responds by publishing the requested thing shadow to the `$aws/things/myLightBulb/shadow/get/accepted` topic:

```
{
  "messageNumber": 1,
  "payload": {
    "state": {
```

```
      "reported": {
        "color": "red"
      }
    },
    "metadata": {
      "reported": {
        "color": {
          "timestamp": 1469564492
        }
      }
    },
    "version": 1,
    "timestamp": 1469564571
  },
  "qos": 0,
  "timestamp": 1469564571533,
  "topic": "$aws/things/myLightBulb/shadow/get/accepted"
}
```

If you don't see a message on the `$aws/things/myLightBulb/shadow/get/accepted` topic, check the `$aws/things/myLightBulb/shadow/get/rejected` topic for any error messages.

The application displays this information to the user, and the user requests a change to the light bulb's color (from red to green). To do this, the application publishes a message on the `$aws/things/myLightBulb/shadow/update` topic:

```
{
    "state": {
        "desired": {
            "color": "green"
        }
    }
}
```

To simulate this, use the AWS IoT MQTT client to publish the preceding message to the `$aws/things/myLightBulb/shadow/update` topic.

The Thing Shadows service responds by sending a message to the `$aws/things/myLightBulb/shadow/update/accepted` topic:

```
{
  "messageNumber": 5,
  "payload": {
    "state": {
      "desired": {
        "color": "green"
      }
    },
    "metadata": {
      "desired": {
        "color": {
          "timestamp": 1469564658
        }
      }
    },
    "version": 2,
    "timestamp": 1469564658
  },
  "qos": 0,
  "timestamp": 1469564658286,
  "topic": "$aws/things/myLightBulb/shadow/update/accepted"
}
```

and to the `$aws/things/myLightBulb/shadow/update/delta` topic:

```
{
  "messageNumber": 1,
  "payload": {
    "version": 2,
    "timestamp": 1469564658,
    "state": {
      "color": "green"
    },
    "metadata": {
      "color": {
        "timestamp": 1469564658
      }
    }
  },
  "qos": 0,
  "timestamp": 1469564658309,
  "topic": "$aws/things/myLightBulb/shadow/update/delta"
}
```

The Thing Shadow service publishes a message to this topic when it accepts a thing shadow update and the resulting thing shadow contains different values for desired and reported states.

The Thing Shadow service also publishes a message to the `$aws/things/myLightBulb/shadow/update/documents` topic:

```
{
    "previous":{
        "state":{
            "reported":{
                "color":"red"
            }
        },
        "metadata":{
            "reported":{
                "color":{
                    "timestamp":1483467764
                }
            }
        },
        "version":1
    },
    "current":{
        "state":{
            "desired":{
                "color":"green"
            },
            "reported":{
                "color":"red"
            }
        },
        "metadata":{
            "desired":{
                "color":{
                    "timestamp":1483468612
                }
            },
            "reported":{
                "color":{
                    "timestamp":1483467764
                }
            }
       },
```

```
        "version":2
    },
    "timestamp":1483468612
}
```

The light bulb is subscribed to the `$aws/things/myLightBulb/shadow/update/delta` topic, so it receives the message, changes its color, and publishes its new state. To simulate this, use the AWS IoT MQTT client to publish the following message to the `$aws/things/myLightbulb/shadow/update` topic to update the shadow state:

```
{
    "state":{
        "reported":{
            "color":"green"
        },
        "desired":null}
    }
}
```

In response, the Thing Shadows service sends a message to the `$aws/things/myLightBulb/shadow/update/accepted` topic:

```
{
  "messageNumber": 6,
  "payload": {
    "state": {
      "reported": {
        "color": "green"
      },
      "desired": null
    },
    "metadata": {
      "reported": {
        "color": {
          "timestamp": 1469564801
        }
      },
      "desired": {
        "timestamp": 1469564801
      }
    },
    "version": 3,
    "timestamp": 1469564801
  },
  "qos": 0,
  "timestamp": 1469564801673,
  "topic": "$aws/things/myLightBulb/shadow/update/accepted"
}
```

and to the `$aws/things/myLightBulb/shadow/update/documents` topic:

```
{
    "previous":{
    "state":{
        "reported":{
            "color":"red"
        }
    },
    "metadata":{
         "reported":{
              "color":{
```

```
                "timestamp":1483470355
            }
        }
    },
    "version":3
},
"current":{
    "state":{
        "reported":{
            "color":"green"
        }
    },
    "metadata":{
        "reported":{
            "color":{
                "timestamp":1483470364
            }
        }
    },
    "version":4
},
"timestamp":1483470364
}
```

The app requests the current state from the Thing Shadows service and displays the most recent state data. To simulate this, run the following command:

```
aws iot-data get-thing-shadow --thing-name "myLightBulb" "output.txt" && cat "output.txt"
```

Note
On Windows, omit the `&& cat "output.txt"`, which displays the contents of output.txt to the console. You can open the file in Notepad or any text editor to see the contents of the thing shadow.

The Thing Shadows service returns the thing shadow document:

```
{
    "state":{
        "reported":{
            "color":"green"
        }
    },
    "metadata":{
        "reported":{
            "color":{
                "timestamp":1469564801
            }
        }
    },
    "version":3,
    "timestamp":1469564864}
```

To delete the thing shadow, publish an empty message to the `$aws/things/myLightBulb/shadow/delete` topic. AWS IoT will respond by publishing a message to the `$aws/things/myLightBulb/shadow/delete/accepted` topic:

```
{
  "version" : 1,
  "timestamp" : 1488565234
}
```

Detecting a Thing is Connected

To determine if a device is currently connected, include a connected setting in the thing shadow and use an MQTT Last Will and Testament (LWT) message that will set the connected setting to `false` if a device is disconnected due to error.

Note
Currently, LWT messages sent to AWS IoT reserved topics (topics that begin with $) are ignored by the AWS IoT Shadows service, but are still processed by subscribed clients and by the AWS IoT rules engine. If you want the AWS IoT Shadows service to receive LWT messages, register an LWT message to a non-reserved topic and create a rule that republishes the message on the reserved topic. The following example shows how to create a republish rule that listens for a messages from the `my/things/myLightBulb/update` topic and republishes it to `$aws/things/myLightBulb/shadow/update`.

```
{
    "rule": {
    "ruleDisabled": false,
    "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 a device connects, it registers an LWT that sets the connected setting to `false`:

```
{
    "state": {
        "reported":
        {
            "connected":"false"
        }
    }
}
```

It also publishes a message on its update topic (`$aws/things/myLightBulb/shadow/update`), setting its connected state to `true`:

```
{
     "state": {
        "reported":
        {
            "connected":"true"
        }
    }
}
```

When the device disconnects gracefully, it publishes a message on its update topic and sets its connected state to `false`:

```
{
    "state": {
        "reported":{
            "connected":"false"
```

```
        }
    }
}
```

If the device disconnects due to an error, its LWT message is posted automatically to the update topic.

Device Shadows Documents

The Thing Shadows service respects all rules of the JSON specification. Values, objects, and arrays are stored in the thing shadow document.

Contents

Document Properties

A thing shadow document has the following properties:

`state`

`desired`

The desired state of the thing. Applications can write to this portion of the document to update the state of a thing without having to directly connect to a thing.

`reported`

The reported state of the thing. Things write to this portion of the document to report their new state. Applications read this portion of the document to determine the state of a thing.

`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.

`timestamp`

Indicates when the message was transmitted by AWS IoT. By using the timestamp in the message and the timestamps for individual attributes in the `desired` or `reported` section, a thing can determine how old an updated item is, even if it doesn't feature 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 Device Shadow Document Syntax (p. 231).

Versioning of a Thing Shadow

The Thing Shadows service supports versioning on every update message (both request and response), which means that with every update of a thing 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 thing 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.

In some cases, a client might bypass version matching by not submitting a version.

Client Token

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.

Example Document

Here is an example thing shadow document:

```
{
    "state" : {
        "desired" : {
          "color" : "RED",
          "sequence" : [ "RED", "GREEN", "BLUE" ]
        },
        "reported" : {
          "color" : "GREEN"
        }
    },
    "metadata" : {
        "desired" : {
            "color" : {
                "timestamp" : 12345
            },
            "sequence" : {
                "timestamp" : 12345
            }
        },
        "reported" : {
            "color" : {
                "timestamp" : 12345
            }
        }
    },
    "version" : 10,
    "clientToken" : "UniqueClientToken",
    "timestamp": 123456789
}
```

Empty Sections

A thing shadow document contains a `desired` section only if it has a desired state. For example, the following is a valid state document with no `desired` section:

```
{
    "reported" : { "temp": 55 }
}
```

The `reported` section can also be empty:

```
{
    "desired" : { "color" : "RED" }
}
```

If an update causes the `desired` or `reported` sections to become null, the section is removed from the document. To remove the `desired` section from a document (in response, for example, to a device updating its state), set the desired section to `null`:

```
{
    "state": {
        "reported": {
            "color": "red"
        },
        "desired": null
    }
}
```

It is also possible a thing shadow document will not contain `desired` or `reported` sections. In that case, the shadow document is empty. For example, this is a valid document:

```
{
}
```

Arrays

Thing 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" ]
    }
}
```

Using Device Shadows

AWS IoT provides three methods for working with thing shadows:

`UPDATE`

Creates a thing shadow if it doesn't exist, or updates the content of a thing shadow with the data provided in the request. The data is stored with timestamp information to indicate when it was last updated. Messages are sent to all subscribers with the difference between `desired` or `reported` state (delta). Things or apps that receive a message can perform an action based on the difference between `desired` or `reported` states. For example, a device can update its state to the desired state, or an app can update its UI to show the change in the device's state.

`GET`

Retrieves the latest state stored in the thing shadow (for example, during start-up of a device to retrieve configuration and the last state of operation). This method returns the full JSON document, including metadata.

`DELETE`

Deletes a thing shadow, including all of its content. This removes the JSON document from the data store. You can't restore a thing shadow you deleted, but you can create a new thing shadow with the same name.

Protocol Support

These methods are supported through both MQTT and a RESTful API over HTTPS. Because MQTT is a publish/subscribe communication model, AWS IoT implements a set of reserved topics. Things or applications subscribe to these topics before publishing on a request topic in order to implement a request–response behavior. For more information, see Device Shadow MQTT Topics (p. 225) and Device Shadow RESTful API (p. 222).

Updating a Thing Shadow

You can update a thing shadow by using the UpdateThingShadow (p. 223) RESTful API or by publishing to the /update (p. 225) topic. Updates affect only the fields specified in the request.

Initial state:

```
{
    "state": {
        "reported" : {
            "color" : { "r" :255, "g": 255, "b": 0 }
        }
    }
}
```

An update message is sent:

```
{
```

```
    "state": {
        "desired" : {
            "color" : { "r" : 10 },
            "engine" : "ON"
        }
    }
}
```

The device receives the `desired` state on the `/update/delta` topic that is triggered by the previous `/update` message and then executes the desired changes. When finished, the device should confirm its updated state through the `reported` section in the thing shadow JSON document.

Final state:

```
{
    "state": {
        "reported" : {
            "color" : {  "r" : 10, "g" : 255, "b": 0 },
            "engine" : "ON"
        }
    }
}
```

Retrieving a Thing Shadow Document

You can retrieve a thing shadow by using the GetThingShadow (p. 223) RESTful API or by subscribing and publishing to the /get (p. 228) topic. This retrieves the entire document plus the delta between the `desired` or `reported` states.

Example document:

```
{
    "state": {
        "desired": {
            "lights": {
                "color": "RED"
            },
            "engine": "ON"
        },
        "reported": {
            "lights": {
                "color": "GREEN"
            },
            "engine": "ON"
        }
    },
    "metadata": {
        "desired": {
            "lights": {
                "color": {
                    "timestamp": 123456
                },
                "engine": {
                    "timestamp": 123456
                }
            }
        },
        "reported": {
            "lights": {
                "color": {
                    "timestamp": 789012
                }
```

```
            },
            "engine": {
                "timestamp": 789012
            }
        },
        "version": 10,
        "timestamp": 123456789
    }
}
```

Response:

```
{
    "state": {
        "desired": {
            "lights": {
                "color": "RED"
            },
            "engine": "ON"
        },
        "reported": {
            "lights": {
                "color": "GREEN"
            },
            "engine": "ON"
        },
        "delta": {
            "lights": {
                "color": "RED"
            }
        }
    },
    "metadata": {
        "desired": {
            "lights": {
                "color": {
                    "timestamp": 123456
                },
             }
             "engine": {
                "timestamp": 123456
            }
        },
        "reported": {
            "lights": {
                "color": {
                    "timestamp": 789012
                }
            },
            "engine": {
                "timestamp": 789012
            }
        },
        "delta": {
            "lights": {
                "color": {
                    "timestamp": 123456
                }
            }
        }
    },
    "version": 10,
    "timestamp": 123456789
}
```

Optimistic Locking

You can use the state document version to ensure you are updating the most recent version of a thing 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; request will be rejected)

```
{
    "state": {
        "desired": {
            "colors": [
                "BLUE"
            ]
        }
    },
    "version": 9
}
```

Result:

```
409 Conflict
```

Update: (version matches; this request will be accepted)

```
{
    "state": {
        "desired": {
            "colors": [
                "BLUE"
            ]
        }
    },
    "version": 10
}
```

Final state:

```
{
    "state": {
        "desired": {
            "colors": [
                "BLUE"
            ]
        }
    },
    "version": 11
```

```
}
```

Deleting Data

You can delete data from a thing shadow by publishing to the /update (p. 225) topic, setting the fields to be deleted to null. Any field with a value of `null` is removed from the document.

Initial state:

```
{
    "state": {
        "desired" : {
            "lights": { "color": "RED" },
            "engine" : "ON"
        },
        "reported" : {
            "lights" : { "color": "GREEN"  },
            "engine" : "OFF"
        }
    }
}
```

An update message is sent:

```
{
    "state": {
        "desired": null,
        "reported": {
            "engine": null
        }
    }
}
```

Final state:

```
{
    "state": {
        "reported" : {
            "lights" : { "color" : "GREEN" }
        }
    }
}
```

You can delete all data from a thing shadow by setting its state to `null`. For example, sending the following message will delete all of the state data, but the thing shadow will remain.

```
{
    "state": null
}
```

The thing shadow still exists even if its state is `null`. The version of the thing shadow will be incremented when the next update occurs.

Deleting a Thing Shadow

You can delete a thing shadow document by using the DeleteThingShadow (p. 224) RESTful API or by publishing to the /delete (p. 230) topic.

Note
Deleting a thing shadow does not delete the thing, and deleting a thing does not delete the thing shadow.

Initial state:

```
{
    "state": {
        "desired" : {
            "lights": { "color": "RED" },
            "engine" : "ON"
        },
        "reported" : {
            "lights" : { "color": "GREEN"  },
            "engine" : "OFF"
        }
    }
}
```

An empty message is published to the /delete topic.

Final state:

```
HTTP 404 - resource not found
```

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 Thing Shadows 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.

Observing State Changes

When a thing shadow is updated, messages are published on two MQTT topics:

- $aws/things/*thing-name*/shadow/update/accepted

- $aws/things/*thing-name*/shadow/update/delta

The message sent to the `update/delta` topic is intended for the thing whose state is being updated. This message contains only the difference between the `desired` and `reported` sections of the thing shadow document. Upon receiving this message, the thing decides whether to make the requested change. If the thing's state is changed, it publishes its new current state to the `$aws/things/thing-name/shadow/update` topic.

Devices and applications can subscribe to either of these topics to be notified when the state of the document has changed.

Here is an example of that flow:

1. Device reports state.
2. The system updates the state document in its persistent data store.
3. The system publishes a delta message, which contains only the delta and is targeted at the subscribed devices. Devices should subscribe to this topic to receive updates.
4. The thing shadow publishes an accepted message, which contains the entire received document, including metadata. Applications should subscribe to this topic to receive updates.

Message Order

There is no guarantee that messages from the AWS IoT service will arrive at the device in any specific order.

Initial state document:

```
{
    "state" : {
        "reported" : { "color" : "blue" }
    },
    "version" : 10,
    "timestamp": 123456777
}
```

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 Device Shadow Messages

To reduce the size of thing shadow messages sent to your device, define a rule that selects only the fields your device needs and 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(2)}/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 thing 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(2)` evaluates to the thing name in the original topic. For more information about creating rules, see Rules.

Device Shadow RESTful API

A thing shadow exposes the following URI for updating state information:

```
https://endpoint/things/thingName/shadow
```

The endpoint is specific to your AWS account. To retrieve your endpoint, use the describe-endpoint command. The format of the endpoint is as follows:

```
identifier.iot.region.amazonaws.com
```

API Actions

- GetThingShadow (p. 223)
- UpdateThingShadow (p. 223)
- DeleteThingShadow (p. 224)

GetThingShadow

Gets the thing 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
```

Response

Upon success, the response includes the standard HTTP headers plus the following code and body:

```
HTTP 200
BODY: response state document
```

For more information, see Example Response State Document (p. 232).

Authorization

Retrieving a thing shadow requires a policy that allows the caller to perform the `iot:GetThingShadow` action. The Thing Shadows 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 thing shadow:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": "iot:GetThingShadow",
        "Resource": ["arn:aws:iot:region:account:thing/thing"]
    }]
}
```

UpdateThingShadow

Updates the thing 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 thing shadow.

Request

The request includes the standard HTTP headers plus the following URI and body:

```
HTTP POST https://endpoint/things/thingName/shadow
BODY: request state document
```

For more information, see Example Request State Document (p. 231).

Response

Upon success, the response includes the standard HTTP headers plus the following code and body:

```
HTTP 200
BODY: response state document
```

For more information, see Example Response State Document (p. 232).

Authorization

Updating a thing shadow requires a policy that allows the caller to perform the `iot:UpdateThingShadow` action. The Thing Shadows 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 thing shadow:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": "iot:UpdateThingShadow",
        "Resource": ["arn:aws:iot:region:account:thing/thing"]
    }]
}
```

DeleteThingShadow

Deletes the thing shadow for the specified thing.

Request

The request includes the standard HTTP headers plus the following URI:

```
HTTP DELETE https://endpoint/things/thingName/shadow
```

Response

Upon success, the response includes the standard HTTP headers plus the following code and body:

```
HTTP 200
BODY: Empty response state document
```

Authorization

Deleting a thing shadow requires a policy that allows the caller to perform the `iot:DeleteThingShadow` action. The Thing Shadows 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 thing shadow:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": "iot:DeleteThingShadow",
        "Resource": ["arn:aws:iot:region:account:thing/thing"]
    }]
}
```

Device Shadow MQTT Topics

The Thing Shadows service uses reserved MQTT topics to enable applications and things to get, update, or delete the state information for a thing (thing shadow). The names of these topics start with $aws/things/*thingName*/shadow. Publishing and subscribing on thing shadow topics requires topic-based authorization. AWS IoT reserves the right to add new topics to the existing topic structure. For this reason, we recommend that you avoid wild card subscriptions to shadow topics. For example, avoid subscribing to topic filters like `$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 Device Shadows Data Flow (p. 203).

The following are the MQTT topics used for interacting with thing shadows.

Topics

/update

Publish a request state document to this topic to update the thing shadow:

```
$aws/things/thingName/shadow/update
```

A client attempting to update the state of a thing would send a JSON request state document like this:

```
{
    "state" : {
        "desired" : {
            "color" : "red",
            "power" : "on"
```

```
        }
    }
}
```

A thing updating its thing shadow would send a JSON request state document like this:

```
{
    "state" : {
        "reported" : {
            "color" : "red",
            "power" : "on"
        }
    }
}
```

AWS IoT responds by publishing to either /update/accepted (p. 226) or /update/rejected (p. 227).

For more information, see Request State Documents (p. 231).

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/accepted

AWS IoT publishes a response state document to this topic when it accepts a change for the thing shadow:

```
$aws/things/thingName/shadow/update/accepted
```

For more information, see Response State Documents (p. 232).

Example Policy

The following is an example of the required policy:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": [
            "iot:Subscribe",
            "iot:Receive"
        ],
        "Resource": ["arn:aws:iot:region:account:topicfilter/$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:

```
$aws/things/thingName/shadow/update/documents
```

The JSON document will contain two primary nodes: `previous` and `current`. The `previous` node will contain the contents of the full shadow document before the update was performed while `current` will contain the full shadow document after the update is successfully applied. When the device shadow is updated (created) for the first time, the `previous` node will contain `null`.

Example Policy

The following is an example of the required policy:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": [
            "iot:Subscribe",
            "iot:Receive"
        ],
        "Resource": ["arn:aws:iot:region:account:topicfilter/$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 thing shadow:

```
$aws/things/thingName/shadow/update/rejected
```

For more information, see Error Response Documents (p. 233).

Example Policy

The following is an example of the required policy:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": [
            "iot:Subscribe",
            "iot:Receive"
        ],
        "Resource": ["arn:aws:iot:region:account:topicfilter/$aws/things/thingName/shadow/
update/rejected"]
    }]
}
```

/update/delta

AWS IoT publishes a response state document to this topic when it accepts a change for the thing shadow and the request state document contains different values for `desired` and `reported` states:

```
$aws/things/thingName/shadow/update/delta
```

For more information, see Response State Documents (p. 232).

Publishing 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",
            "iot:Receive"
        ],
        "Resource": ["arn:aws:iot:region:account:topicfilter/$aws/things/thingName/shadow/
update/delta"]
    }]
}
```

/get

Publish an empty message to this topic to get the thing shadow:

```
$aws/things/thingName/shadow/get
```

AWS IoT responds by publishing to either /get/accepted (p. 229) or /get/rejected (p. 229).

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/get"]
    }]
}
```

/get/accepted

AWS IoT publishes a response state document to this topic when returning the thing shadow:

```
$aws/things/thingName/shadow/get/accepted
```

For more information, see Response State Documents (p. 232).

Example Policy

The following is an example of the required policy:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": [
            "iot:Subscribe",
            "iot:Receive"
        ],
        "Resource": ["arn:aws:iot:region:account:topicfilter/$aws/things/thingName/shadow/
get/accepted"]
    }]
}
```

/get/rejected

AWS IoT publishes an error response document to this topic when it can't return the thing shadow:

```
$aws/things/thingName/shadow/get/rejected
```

For more information, see Error Response Documents (p. 233).

Example Policy

The following is an example of the required policy:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Action": [
            "iot:Subscribe",
            "iot:Receive"
        ],
        "Resource": ["arn:aws:iot:region:account:topicfilter/$aws/things/thingName/shadow/
get/rejected"]
    }]
}
```

/delete

To delete a thing shadow, publish an empty message to the delete topic:

```
$aws/things/thingName/shadow/delete
```

The content of the message is ignored.

AWS IoT responds by publishing to either /delete/accepted (p. 230) or /delete/rejected (p. 230).

Example Policy

The following is an example of the required policy:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": [
            "iot:Subscribe",
            "iot:Receive"
        ],
        "Resource": ["arn:aws:iot:region:account:topic filter/$aws/things/thingName/shadow/
delete"]
    }]
}
```

/delete/accepted

AWS IoT publishes a message to this topic when a thing shadow is deleted:

```
$aws/things/thingName/shadow/delete/accepted
```

Example Policy

The following is an example of the required policy:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": [
            "iot:Subscribe",
            "iot:Receive"
        ],
        "Resource": ["arn:aws:iot:region:account:topicfilter/$aws/things/thingName/shadow/
delete/accepted"]
    }]
}
```

/delete/rejected

AWS IoT publishes an error response document to this topic when it can't delete the thing shadow:

```
$aws/things/thingName/shadow/delete/rejected
```

For more information, see Error Response Documents (p. 233).

Example Policy

The following is an example of the required policy:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": [
            "iot:Subscribe",
            "iot:Receive"
        ],
        "Resource": ["arn:aws:iot:region:account:topicfilter/$aws/things/thingName/shadow/
delete/rejected"]
    }]
}
```

Device Shadow Document Syntax

The Thing Shadows service uses the following documents in UPDATE, GET, and DELETE operations using the RESTful API (p. 222) or MQTT Pub/Sub Messages (p. 225). For more information, see Device Shadows Documents (p. 211).

Examples

- Request State Documents (p. 231)
- Response State Documents (p. 232)
- Error Response Documents (p. 233)

Request State Documents

Request state documents have 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.

- `clientToken` — If used, you can verify that the request and response contain the same client token.
- `version` — If used, the Thing Shadows 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:

```
{
    "state": {
        "desired": {
            "attribute1": integer2,
            "attribute2": "string2",
            ...
            "attributeN": boolean2
        },
        "reported": {
            "attribute1": integer1,
            "attribute2": "string1",
            ...
            "attributeN": boolean1
        },
        "delta": {
            "attribute3": integerX,
            "attribute5": "stringY"
        }
    },
    "metadata": {
        "desired": {
            "attribute1": {
                "timestamp": timestamp
            },
            "attribute2": {
                "timestamp": timestamp
            },
            ...
            "attributeN": {
                "timestamp": timestamp
            }
        },
        "reported": {
            "attribute1": {
                "timestamp": timestamp
            },
            "attribute2": {
                "timestamp": timestamp
            },
            ...
            "attributeN": {
                "timestamp": timestamp
            }
        }
    },
    "timestamp": timestamp,
    "clientToken": "token",
    "version": version
}
```

- `state`
  - `reported` — Only present if a thing reported any data in the `reported` section and contains only fields that were in the request state document.

  - `desired` — Only present if a thing reported any data in the `desired` section and contains only fields that were in the request state document.
- `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 thing shadow shared in AWS IoT. It is increased by one over the previous version of the document.

Error Response Documents

Error response documents have 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.
- `clientToken` — Present only if a client token was used when publishing valid JSON to the `/update` topic.

For more information, see Device Shadow Error Messages (p. 233).

Device Shadow Error Messages

The Thing Shadows 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)	• 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 • JSON contains too many levels of nesting; maximum is 6 • State contains an invalid node
401 (Unauthorized)	• Unauthorized

HTTP Error Code	Error Messages
403 (Forbidden)	• Forbidden
404 (Not Found)	• Thing not found
409 (Conflict)	• Version conflict
413 (Payload Too Large)	• The payload exceeds the maximum size allowed
415 (Unsupported Media Type)	• Unsupported documented encoding; supported encoding is UTF-8
429 (Too Many Requests)	• The Thing Shadow service will generate this error message when there are more than 10 in-flight requests.
500 (Internal Server Error)	• Internal service failure

AWS IoT SDKs

Contents

The AWS IoT Device SDKs help you to easily and quickly connect your devices to AWS IoT. 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.

AWS Mobile SDK for Android

The AWS SDK for Android contains a library, samples, and documentation for developers to build connected mobile applications using AWS. This SDK also includes support for calling AWS IoT APIs. 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

Arduino Yún SDK

The AWS IoT Arduino Yún SDK allows developers to connect their Arduino Yún-compatible boards to AWS IoT. By connecting a device to AWS IoT, users can securely work with the message broker, rules, and thing shadows provided by AWS IoT and with other AWS services like AWS Lambda, Amazon Kinesis, and Amazon S3. For more information, see the following:

- Arduino Yún SDK on GitHub
- Arduino Yún SDK Readme

AWS IoT Device SDK for Embedded C

The AWS IoT Device SDK for Embedded C is a collection of C source files that can be used in embedded applications to securely connect to the AWS IoT platform. It includes transport clients, TLS implementations, and examples for their use. It also supports AWS IoT-specific features such as an API to access the Thing Shadows service. It is distributed as source code and is intended to be built into customer firmware along with application code, other libraries, and RTOS. For more information see the following:

- AWS IoT Device SDK for Embedded C GitHub
- AWS IoT Device SDK for Embedded C Readme
- AWS IoT Device SDK for Embedded C Porting Guide

AWS Mobile SDK for iOS

The AWS 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 calling the AWS IoT API.

- AWS SDK for iOS on GitHub
- AWS SDK for iOS Readme
- AWS SDK for iOS Samples

AWS IoT Device SDK for Java

The AWS IoT Device SDK for Java enables Java developers to access the AWS IoT platform through MQTT or MQTT over the WebSocket protocol. The SDK is built with AWS IoT thing shadow support. You can access thing shadows by using HTTP methods, including GET, UPDATE, and DELETE. The SDK also supports a simplified thing shadow access model, which allows developers to exchange data with thing 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 on GitHub
- AWS IoT Device SDK for Java readme

AWS IoT Device SDK for JavaScript

The aws-iot-device-sdk.js package allows 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 on GitHub
- AWS IoT Device SDK for JavaScript readme

AWS IoT Device SDK for Python

The AWS IoT Device SDK for Python allows 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 thing shadows provided by AWS IoT and with other AWS services like AWS Lambda, Amazon Kinesis, and Amazon S3, and more.

- AWS IoT Device SDK for Python on GitHub
- AWS IoT Device SDK for Python readme

Monitoring AWS IoT

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. Before you start monitoring AWS IoT, you should create a monitoring plan that includes answers to the following questions:

- 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?

The next step is to establish a baseline for normal AWS IoT performance in your environment, by measuring performance at various times and under different load conditions. As you monitor AWS IoT, store historical monitoring data so that you can compare it with current performance data, identify normal performance patterns and performance anomalies, and devise methods to address issues.

For example, if you're using Amazon EC2, you can monitor CPU utilization, disk I/O, and network utilization for your instances. When performance falls outside your established baseline, you might need to reconfigure or optimize the instance to reduce CPU utilization, improve disk I/O, or reduce network traffic.

To establish a baseline you should, at a minimum, monitor the following items:

- PublishIn.Success
- PublishOut.Success
- Subscribe.Success
- Ping.Success
- Connect.Success
- GetThingShadow.Accepted
- UpdateThingShadow.Accepted
- DeleteThingShadow.Accepted
- RulesExecuted

Topics

- Monitoring with Amazon CloudWatch (p. 240)
- Logging AWS IoT API Calls with AWS CloudTrail (p. 246)

Monitoring Tools

AWS provides various tools that you can use to monitor AWS IoT. You can configure some of these tools to do the monitoring for you, while 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 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 Monitoring with Amazon CloudWatch (p. 240).
- **Amazon CloudWatch Logs** – Monitor, store, and access your log files from AWS CloudTrail or other sources. For more information, 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 Using 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 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 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
  - Polices
  - Rules
  - Things
- CloudWatch home page shows:
  - Current alarms and status
  - Graphs of alarms and resources
  - Service health status

  In addition, 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

Monitoring with 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 automatically sent to CloudWatch in 1 minute periods. For more information, see What Are Amazon CloudWatch, Amazon CloudWatch Events, and Amazon CloudWatch Logs? in the *Amazon CloudWatch User Guide*.

Topics

AWS IoT Metrics and Dimensions

When you interact with AWS IoT, it 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 using the 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 at https://console.aws.amazon.com/cloudwatch/.
2. In the navigation pane, choose **Metrics**.
3. In the **CloudWatch Metrics by Category** pane, under the metrics category for AWS IoT, select a metrics category, and then in the upper pane, scroll down to view the full list of metrics.

To view metrics using the AWS CLI

- At a command prompt, use the following command:

```
aws cloudwatch list-metrics --namespace "AWS/IoT"
```

CloudWatch displays the following metrics for AWS IoT:

AWS IoT Metrics

AWS IoT sends the following metrics to CloudWatch once per received request.

IoT Metrics

Metric	Description
RulesExecuted	The number of AWS IoT rules executed.

Rule Metrics

Metric	Description
TopicMatch	The number of incoming messages published on a topic on which a rule is listening. The `RuleName` dimension contains the name of the rule.
ParseError	The number of JSON parse errors that occurred in messages published on a topic on which a rule is listening. The `RuleName` dimension contains the name of the rule.

Rule Action Metrics

Metric	Description
Success	The number of successful rule action invocations. The `RuleName` dimension contains the name of the rule that specifies the action. The `ActionType` dimension contains the type of action that was invoked.
Failure	The number of failed rule action invocations. The `RuleName` dimension contains the name of the rule that specifies the action. The `RuleName` dimension contains the name of the rule that specifies the action. The `ActionType` dimension contains the type of action that was invoked.

Message Broker Metrics

Metric	Description
Connect.AuthError	The number of connection requests that could not be authorized by the message broker. The `Protocol` 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 Limits. The `Protocol` 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 Protocol dimension contains the protocol used to send the `CONNECT` message.
Connect.Success	The number of successful connections to the message broker. The `Protocol` dimension contains the protocol used to send the `CONNECT` message.
Connect.Throttle	The number of connection requests that were throttled because the client exceeded the allowed connect request rate. The `Protocol` dimension contains the protocol used to send the `CONNECT` message.

Metric	Description
Ping.Success	The number of ping messages received by the message broker. The `Protocol` dimension contains the protocol used to send the ping message.
PublishIn.AuthError	The number of publish requests the message broker was unable to authorize. The `Protocol` 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 Limits. The `Protocol` 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 `Protocol` dimension contains the protocol used to send the `PUBLISH` message.
PublishIn.Success	The number of publish requests successfully processed by the message broker. The `Protocol` dimension contains the protocol used to send the `PUBLISH` message.
PublishIn.Throttle	The number of publish request that were throttled because the client exceeded the allowed inbound message rate. The `Protocol` dimension contains the protocol used to send the `PUBLISH` message.
PublishOut.AuthError	The number of publish requests made by the message broker that could not be authorized by AWS IoT. The `Protocol` dimension contains the protocol used to send the `PUBLISH` 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 Limits. The `Protocol` dimension contains the protocol used to send the `PUBLISH` message.
PublishOut.Success	The number of publish requests successfully made by the message broker. 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 Limits. The `Protocol` dimension contains the protocol used to send the `SUBSCRIBE` message.

Metric	Description
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 Limits. 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.

Note
The message broker metrics are displayed in the AWS IoT console under **Protocol Metrics**.

Thing Shadow 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.
UpdateThingShadow.Accepted	The number of UpdateThingShadow requests processed successfully. The `Protocol` dimension contains the protocol used to make the request.

Note
The thing shadow metrics are displayed in the AWS IoT console under **Protocol Metrics**.

Dimensions for Metrics

Metrics use the namespace and provide metrics for the following dimension(s):

Dimension	Description
ActionType	The action type specified by the rule that triggered by the request.
Protocol	The protocol used to make the request. Valid values are: MQTT or HTTP
RuleName	The name of the rule triggered by the request.

How Do I Use 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.

These are suggestions to get you started, not a comprehensive list.

- How can I be notified if my things do not connect successfully each day? (p. 244)
- How can I be notified if my things are not publishing data each day? (p. 245)
- How can I be notified if my thing's shadow updates are being rejected each day? (p. 246)

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 and performs 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 SNS topic or Auto Scaling policy. Alarms invoke actions for sustained state changes only. 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.

How can I be notified if my things do not connect successfully each day?

1. Create an Amazon SNS topic, arn:aws:sns:us-east-1:123456789012:things-not-connecting-successfully.

   For more information, see Set Up Amazon Simple Notification Service.

2. Create the alarm.

```
Prompt>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 \
    --unit Count \
    --evaluation-periods 1 \
    --alarm-actions arn:aws:sns:us-east-1:1234567890:things-not-connecting-successfully
```

```
Prompt>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 \
    --unit Count \
    --evaluation-periods 1 \
    --alarm-actions arn:aws:sns:us-east-1:1234567890:things-not-connecting-successfully
```

3. Test the alarm.

```
Prompt>aws cloudwatch set-alarm-state --alarm-name ConnectSuccessAlarm --state-reason
 "initializing" --state-value OK
```

```
 Prompt>aws cloudwatch set-alarm-state --alarm-name ConnectSuccessAlarm --state-reason
 "initializing" --state-value ALARM
```

How can I be notified if my things are not publishing data each day?

1. Create an Amazon SNS topic, `arn:aws:sns:us-east-1:123456789012:things-not-publishing-data`.

   For more information, see Set Up Amazon Simple Notification Service.

2. Create the alarm.

```
Prompt>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 \
    --unit Count \
```

```
    --evaluation-periods 1 \
    --alarm-actions arn:aws:sns:us-east-1:1234567890:things-not-publishing-data
```

3. Test the alarm.

```
Prompt>aws cloudwatch set-alarm-state --alarm-name PublishInSuccessAlarm --state-reason
 "initializing" --state-value OK
```

```
Prompt>aws cloudwatch set-alarm-state --alarm-name PublishInSuccessAlarm --state-reason
 "initializing" --state-value ALARM
```

How can I be notified if my thing's shadow updates are being rejected each day?

1. Create an Amazon SNS topic, arn:aws:sns:us-east-1:1234567890:things-shadow-updates-rejected.

   For more information, see Set Up Amazon Simple Notification Service.

2. Create the alarm.

```
Prompt>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 arn:aws:sns:us-east-1:1234567890:things-shadow-updates-rejected
```

3. Test the alarm.

```
Prompt>aws cloudwatch set-alarm-state --alarm-name UpdateThingShadowSuccessAlarm --
state-reason "initializing" --state-value OK
```

```
Prompt>aws cloudwatch set-alarm-state --alarm-name UpdateThingShadowSuccessAlarm --
state-reason "initializing" --state-value ALARM
```

Logging AWS IoT API Calls with AWS CloudTrail

AWS IoT is integrated with CloudTrail, a service that captures all of the AWS IoTAPI calls and delivers the log files to an Amazon S3 bucket that you specify. CloudTrail captures API calls from the AWS IoT console or from your code to the AWS IoT APIs. Using the information collected by CloudTrail, you can determine the request that was made to AWS IoT, the source IP address from which the request was made, who made the request, when it was made, and so on.

To learn more about CloudTrail, including how to configure and enable it, see the *AWS CloudTrail User Guide*.

AWS IoT Information in CloudTrail

When CloudTrail logging is enabled in your AWS account, API calls made to AWS IoT actions are tracked in CloudTrail log files where they are written with other AWS service records. CloudTrail determines when to create and write to a new file based on a time period and file size.

All AWS IoT actions are logged by CloudTrail and are documented in the AWS IoT API Reference. For example, calls to the **CreateThing**, **ListThings**, and **ListTopicRules** sections generate entries in the CloudTrail log files.

Every log entry contains information about who generated the request. The user identity information in the log entry 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.

You can store your log files in your Amazon S3 bucket for as long as you want, but you can also define Amazon S3 lifecycle rules to archive or delete log files automatically. By default, your log files are encrypted with Amazon S3 server-side encryption (SSE).

If you want to be notified upon log file delivery, you can configure CloudTrail to publish Amazon SNS notifications when new log files are delivered. For more information, see Configuring Amazon SNS Notifications for CloudTrail.

You can also aggregate AWS IoT log files from multiple AWS regions and multiple AWS accounts into a single Amazon S3 bucket.

For more information, see Receiving CloudTrail Log Files from Multiple Regions and Receiving CloudTrail Log Files from Multiple Accounts.

Understanding AWS IoT Log File Entries

CloudTrail log files can contain one or more log entries. Each entry lists multiple JSON-formatted events. A log entry 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. Log entries 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 `AttachPrincipalPolicy` action.

```
{
    "timestamp":"1460159496",
    "AdditionalEventData":"",
    "Annotation":"",
    "ApiVersion":"",
    "ErrorCode":"",
    "ErrorMessage":"",
    "EventID":"8bff4fed-c229-4d2d-8264-4ab28a487505",
    "EventName":"AttachPrincipalPolicy",
    "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":"AKIAI44QH8DHBEXAMPLE",
        "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":""
}
```

Troubleshooting AWS IoT

The following information might help you troubleshoot common issues in AWS IoT.

Tasks

Diagnosing Connectivity Issues

Authentication

How do my devices authenticate AWS IoT endpoints?

Add the AWS IoT CA certificate to your client's trust store. You can download the CA certificate from here.

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.us-east-1.amazonaws.com:8443 -CAfile
CA.pem -cert cert.pem -key privateKey.pem
```

Authorization

I received a `PUBNACK` or `SUBNACK` response from the broker. What do I do?

Make sure there is a policy attached to the certificate you are using to call AWS IoT. All publish/subscribe operations are denied by default.

Setting Up CloudWatch Logs

As messages from your devices pass through the message broker and the rules engine, AWS IoT sends progress events about each message. You can opt in to view these events in CloudWatch Logs. For more information, see CloudWatch Logs.

Note
Before you enable AWS IoT logging, be sure you understand the access permissions to CloudWatch Logs in your AWS account. Users with access to CloudWatch Logs will be able to see debugging information from your devices.

Configuring an IAM Role for Logging

Use the IAM console to create a logging role.

Create an IAM Role for Logging

The following policy documents provide the role policy and trust policy that allow AWS IoT to submit logs to CloudWatch on your behalf.

Role policy:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents",
                "logs:PutMetricFilter",
                "logs:PutRetentionPolicy"
             ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

Trust policy:

```
{
 "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "Service": "iot.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

Register the Logging Role with AWS IoT

Use the AWS IoT console or the following CLI command to register the logging role with AWS IoT.

```
aws iot set-logging-options --logging-options-payload roleArn="arn:aws:iam::<your-aws-
account-num>:role/IoTLoggingRole",logLevel="INFO"
```

The log level can be DEBUG, INFO, ERROR, or DISABLED:

- DEBUG provides the most detailed information of AWS IoT activity.
- INFO provides a summarized view of most actions. This is sufficient for most users.
- ERROR provides error cases only.
- DISABLED removes logging altogether, but keeps your logging role intact.

CloudWatch Log Entry Format

Each log entry has the following information:

Event

Describes the actions that take place in AWS IoT.

LogLevel

The logging level. Can be DEBUG, INFO, ERROR, WARN, DISABLED.

TimeStamp

The time the log was generated.

TraceId

An identifier generated randomly for an incoming request that can be used to filter all of the corresponding logs to one incoming message.

PrincipalId

A certificate fingerprint or a thing name, depending on which endpoint (MQTT or HTTP) received the request from a device.

Depending upon the message context the following fields may also be included in log messages:

Topic Name

The MQTT topic name, which is added to an entry when an MQTT publish or subscribe message is received.

ClientId

The ID of the client that sent an MQTT message.

ThingName

The thing name, which is added to an entry when a request is sent to an HTTP endpoint to update or delete thing state.

RuleId

The rule identifier, which contains the ID of a rule when the rule is triggered.

Log Level

The log level specifies which types of logs will be generated.

ERROR

Any error that causes an operation to fail.

Logs will include ERROR information only.

WARN

Anything that can potentially cause inconsistencies in the system, but might not necessarily cause the operation to fail.

Logs will include ERROR and WARN information.

INFO

High-level information about the flow of things.

Logs will include INFO, ERROR, and WARN information.

DEBUG

Information that might be helpful when debugging a problem.

Logs will include DEBUG, INFO, ERROR, and WARN information.

DISABLED

All logging is disabled.

Logging Events and Error Codes

This section lists the logging events and error codes sent by AWS IoT.

Identity and Security

Operation/Event Name	Description
Authentication Success	Successfully authenticated a certificate.
Authentication Failure	Failed to authenticate a certificate.

Identity and Security Error Codes

Error Code	Error Description
401	Unauthorized

Message Broker

Operation/Event Name	Description
MQTT Publish	MQTT Publish received.
MQTT Subscribe	MQTT Subscribe received.
MQTT Connect	MQTT Connect received.
MQTT Disconnect	MQTT Disconnect received.
HTTP/1.1 POST	MHTTP/1.1 POST received.

Operation/Event Name	Description
HTTP/1.1 GET	HTTP/1.1 GET received.
HTTP/1.1 Unsupported Method	Used when a message contains a syntax error or the action (HTTP PUT/DELETE/) is forbidden.
Malformed HTTP Message	The connection was terminated because of a malformed HTTP message.
Malformed MQTT Message	The connection was terminated because of a malformed MQTT message.
Authorization Failed	This client attempted to publish to or subscribe on a topic for which it has no authorization.
Package Exceeds Maximum Payload Size	This client attempted to publish a payload that exceeds the message broker's upper limit.

Message Broker Error Codes

Error Code	Error Description
400	Bad Request
401	Unauthorized
403	Forbidden
503	Service Unavailable

Rules Engine Events

Operation/Event Name	Description
MessageReceived	Received a request for a topic.
DynamoActionSuccess	Successfully put DynamoDB record.
DynamoActionFailure	Failed to put DynamoDB record.
KinesisActionSuccess	Successfully published Amazon Kinesis message.
KinesisActionFailure	Failed to publish Amazon Kinesis message.
LambdaActionSuccess	Successfully invoked Lambda function.
LambdaActionFailure	Failed to invoke Lambda function.
RepublishActionSuccess	Successfully republished message.
MessageReceived	Received request for a topic.
RepublishActionFailure	Failed to republish message.
S3ActionSuccess	Successfully put Amazon S3 object.
S3ActionFailure	Failed to put Amazon S3 object.
SNSActionSuccess	Successfully published to Amazon SNS topic.

Operation/Event Name	Description
SNSActionFailure	Failed to publish to Amazon SNS topic.
SQSActionSuccess	Successfully sent message to Amazon SQS.
SQSActionFailure	Failed to send message to Amazon SQS.

Thing Shadow Events

Operation/Event Name	Description
UpdateThingState	A thing's state is updated over HTTP or MQTT.
DeleteThing	A thing is deleted.

Thing 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.

Diagnosing Rules Issues

CloudWatch Logs is the best place to debug issues you are having with rules. When you enable CloudWatch Logs for AWS IoT, you get visibility into which rules are triggered and their success or failure. You also get information about whether WHERE clause conditions match.

The most common issue is authorization. In this case, the logs will tell you your role is not authorized to perform AssumeRole on the resource.

To view CloudWatch logs (console)

1. In the AWS Management Console, navigate to the CloudWatch console.
2. Choose **Logs**, and then choose the **AWSIoTLogs** log group from the list.
3. On the **Streams for AWSIoTLogs** page, you will find a log stream for each principal (X.509 certificate, IAM user, or Amazon Cognito identity) that called into AWS IoT under your account.

For more information, see CloudWatch Logs.

External services are controlled by the end user. Before rule execution, make sure external services are set up with enough throughput and capacity units.

Diagnosing Problems with Thing Shadows

Diagnosing Thing Shadows

Issue	Troubleshooting Guidelines
A thing shadow document is rejected with "Invalid JSON document."	If you are unfamiliar with JSON, modify the examples provided in this guide for your own use. For more information, see Thing Shadow Document Syntax.
I submitted correct JSON, but none or only parts of it are stored in the thing shadow document.	Be sure you are following the JSON formatting guidelines. Only JSON fields in the `desired` and `reported` sections will be stored. JSON content (even if formally correct) outside of those sections will be ignored.
I received an error that the thing shadow exceeds the allowed size.	The thing shadow supports 8 KB of data only. Try shortening field names inside of your JSON document or simply create more thing shadows. A device can have an unlimited number of thing shadows. The only requirement is that the thing name is unique in your account.
When I receive a thing shadow, it is larger than 8 KB. How can this happen?	Upon receipt, the AWS IoT service adds metadata to the thing 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 thing 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 ./update/accepted topic so you will 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 ./update/accepted and ./update/delta 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 thing shadow topics, but when I attempt to update the thing 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 Thing Shadows service will log errors to CloudWatch Logs. To identify device and configuration issues, enable CloudWatch Logs and view the logs for debug information.