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. 2).
- To learn how to connect a thing to AWS IoT, see Getting Started with AWS IoT (p. 18).

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. 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 SDKs**—Build your IoT applications using language-specific APIs. For more information, see AWS SDKs and Tools.
- **AWS IoT API**—Build your IoT applications using HTTP or HTTPS requests. For more information about the API actions for AWS IoT, see Actions in the *AWS IoT API Reference*.
- **AWS IoT Thing SDK for C**—Build IoT applications for resource-constrained things, such as microcontrollers.

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

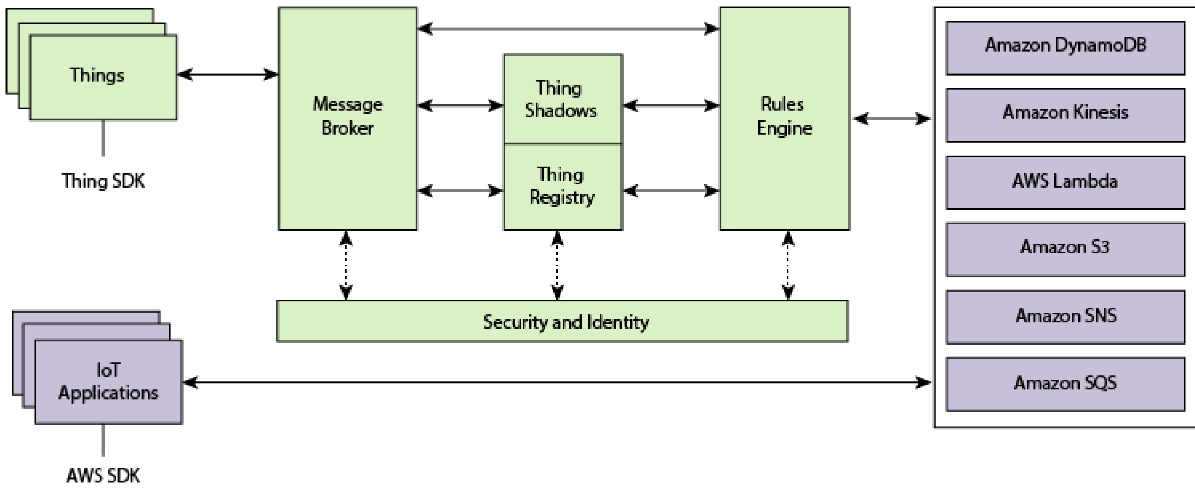
AWS IoT enables Internet-connected things to connect to the AWS cloud and lets applications in the cloud interact with Internet-connected things. Common IoT applications either collect and process telemetry from devices or enable users to control a device remotely.

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. You can use all or only some JSON properties in a message. 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 AWS CloudFormation Quickstart (p. 12)
- Next Steps (p. 17)

AWS IoT Button Wizard Quickstart

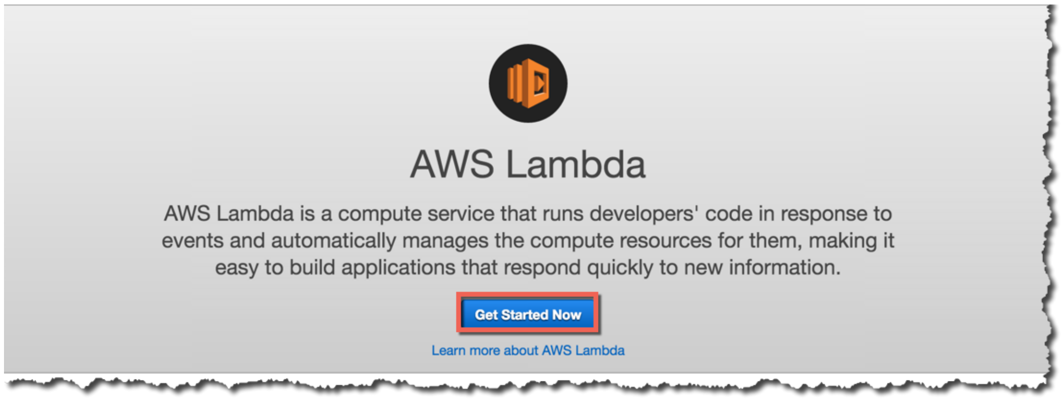
The AWS IoT button wizard is a Lambda blueprint, so you need to sign in to the AWS Lambda console in order to use it. If you do not have an AWS account, you can create one by following these steps.

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

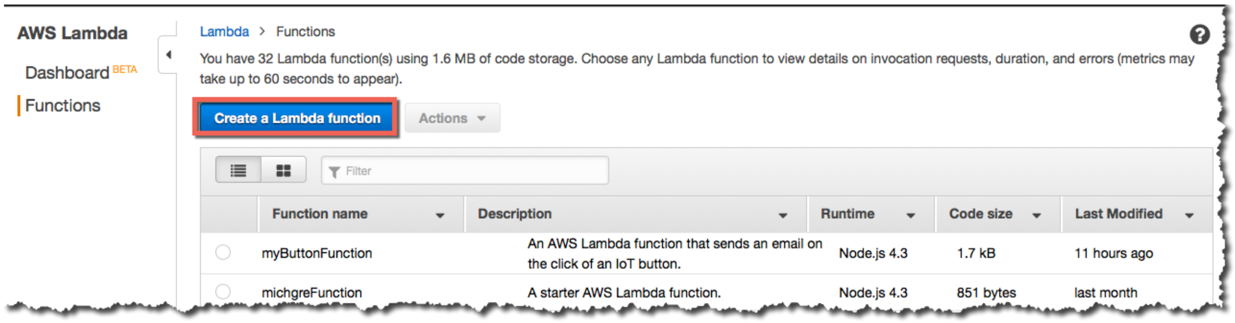
1. Sign in to the AWS Management Console and open the AWS Lambda console.
2. If this is your first time in the AWS Lambda console, you will see the following page. Choose the **Get Started Now** button.



AWS Lambda

AWS Lambda is a compute service that runs developers' code in response to events and automatically manages the compute resources for them, making it easy to build applications that respond quickly to new information.

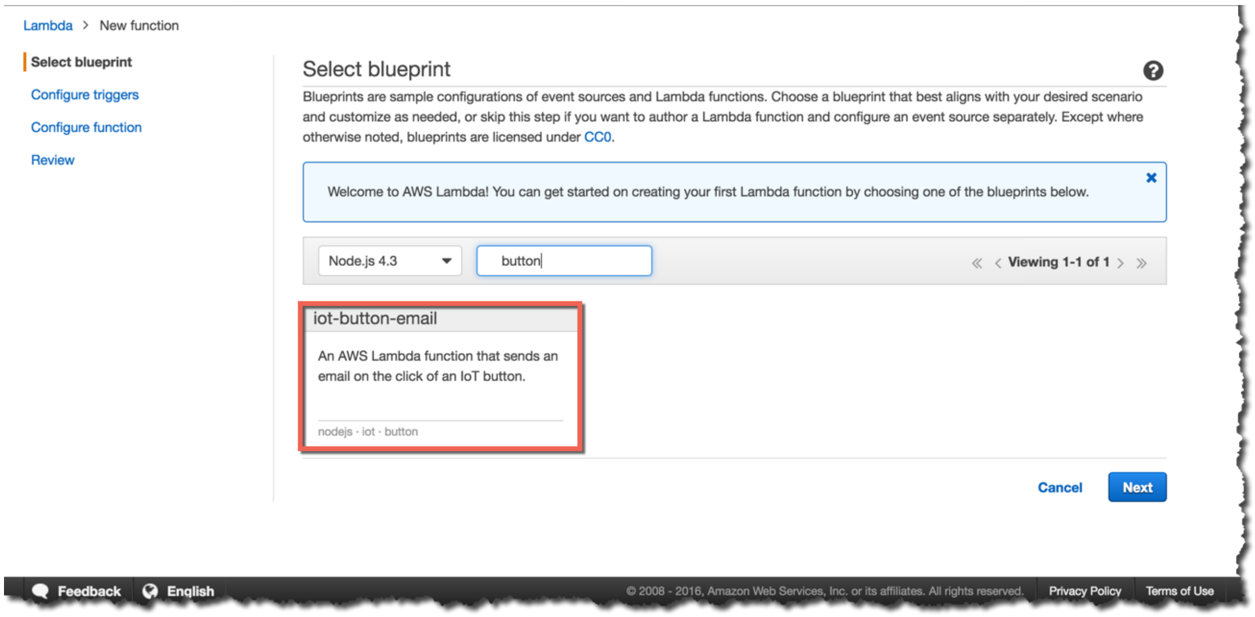
If you have used the AWS Lambda console before, you will see the following page. Choose the **Create a Lambda function** button.



You have 32 Lambda function(s) using 1.6 MB of code storage. Choose any Lambda function to view details on invocation requests, duration, and errors (metrics may take up to 60 seconds to appear).

	Function name	Description	Runtime	Code size	Last Modified
○	myButtonFunction	An AWS Lambda function that sends an email on the click of an IoT button.	Node.js 4.3	1.7 kB	11 hours ago
○	michgreFunction	A starter AWS Lambda function.	Node.js 4.3	851 bytes	last month

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.


Lambda > New function

Select blueprint

Blueprints are sample configurations of event sources and Lambda functions. Choose a blueprint that best aligns with your desired scenario and customize as needed, or skip this step if you want to author a Lambda function and configure an event source separately. Except where otherwise noted, blueprints are licensed under CC0.

Welcome to AWS Lambda! You can get started on creating your first Lambda function by choosing one of the blueprints below.

Node.js 4.3 | button | Viewing 1-1 of 1

iot-button-email

An AWS Lambda function that sends an email on the click of an IoT button.

nodejs · iot · button

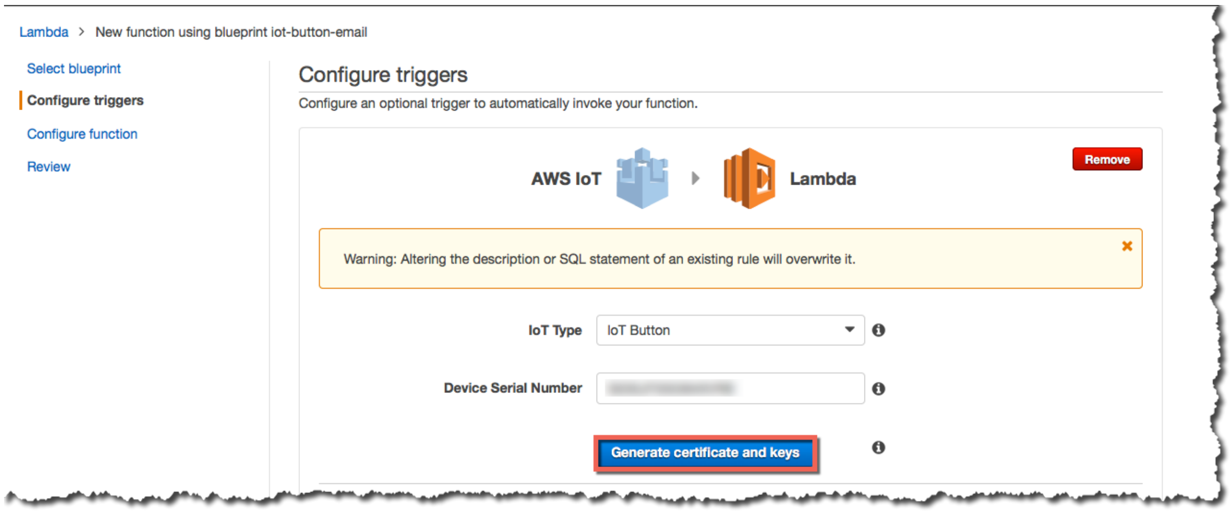
© 2008 - 2016, Amazon Web Services, Inc. or its affiliates. All rights reserved.

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.


Lambda > New function using blueprint iot-button-email

Configure triggers

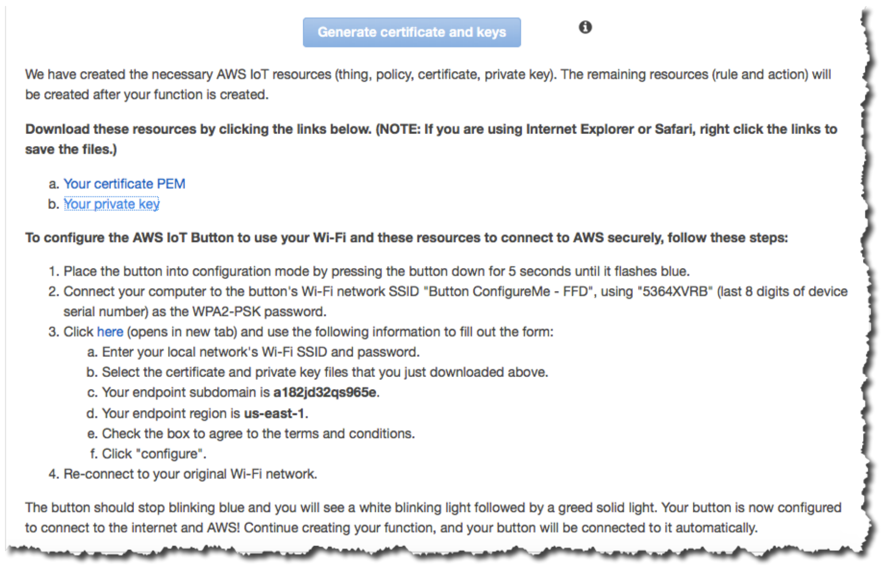
Configure an optional trigger to automatically invoke your function.

Warning: Altering the description or SQL statement of an existing rule will overwrite it.

IoT Type: IoT Button

Device Serial Number: [redacted]

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



We have created the necessary AWS IoT resources (thing, policy, certificate, private key). The remaining resources (rule and action) will be created after your function is created.

Download these resources by clicking the links below. (NOTE: If you are using Internet Explorer or Safari, right click the links to save the files.)

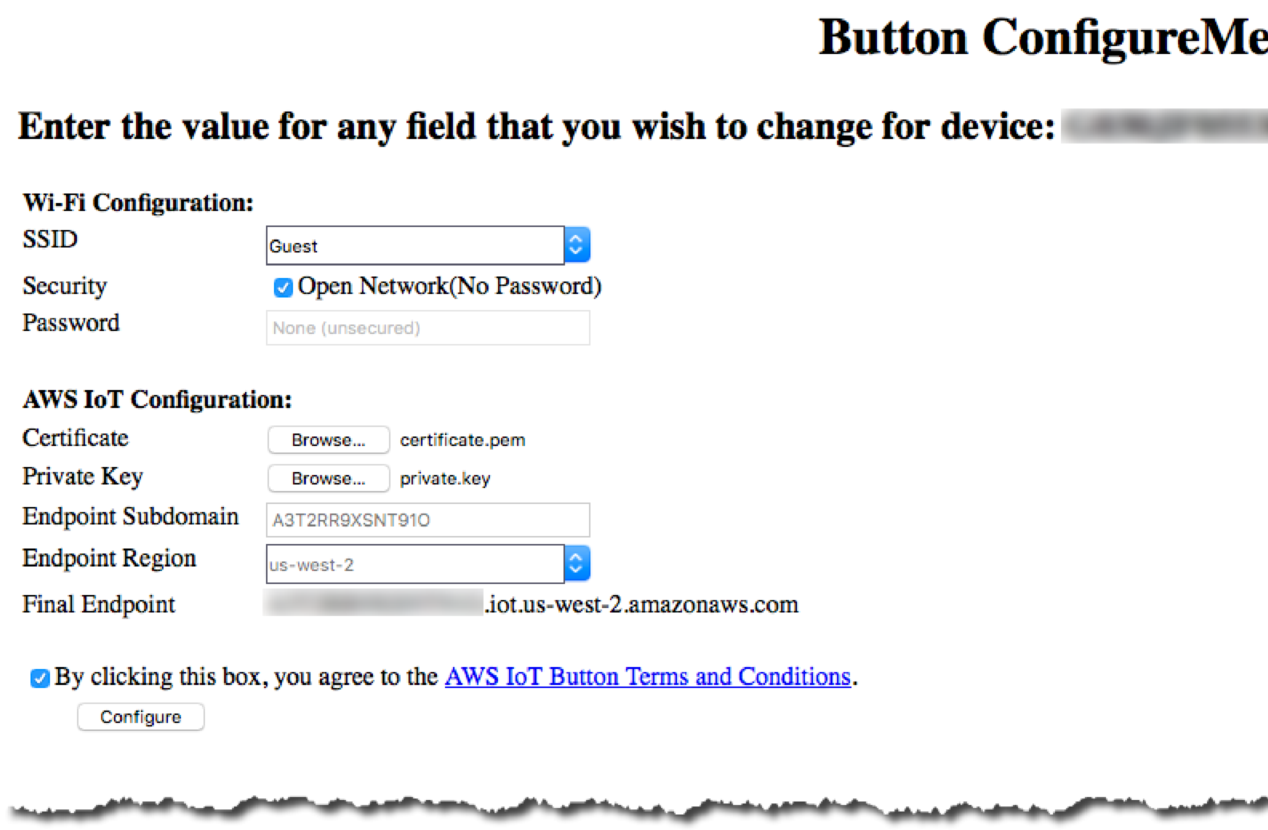
a. Your certificate PEM
b. Your private key

To configure the AWS IoT Button to use your Wi-Fi and these resources to connect to AWS securely, follow these steps:

1. Place the button into configuration mode by pressing the button down for 5 seconds until it flashes blue.
2. Connect your computer to the button's Wi-Fi network SSID "Button ConfigureMe - FFD", using "5364XVRB" (last 8 digits of device serial number) as the WPA2-PSK password.
3. Click here (opens in new tab) and use the following information to fill out the form:
   a. Enter your local network's Wi-Fi SSID and password.
   b. Select the certificate and private key files that you just downloaded above.
   c. Your endpoint subdomain is **a182jd32qs965e**.
   d. Your endpoint region is **us-east-1**.
   e. Check the box to agree to the terms and conditions.
   f. Click "configure".
4. Re-connect to your original Wi-Fi network.

The button should stop blinking blue and you will see a white blinking light followed by a greed solid light. Your button is now configured to connect to the internet and AWS! Continue creating your function, and your button will be connected to it automatically.

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.



Button ConfigureMe

Enter the value for any field that you wish to change for device: [redacted]

Wi-Fi Configuration:

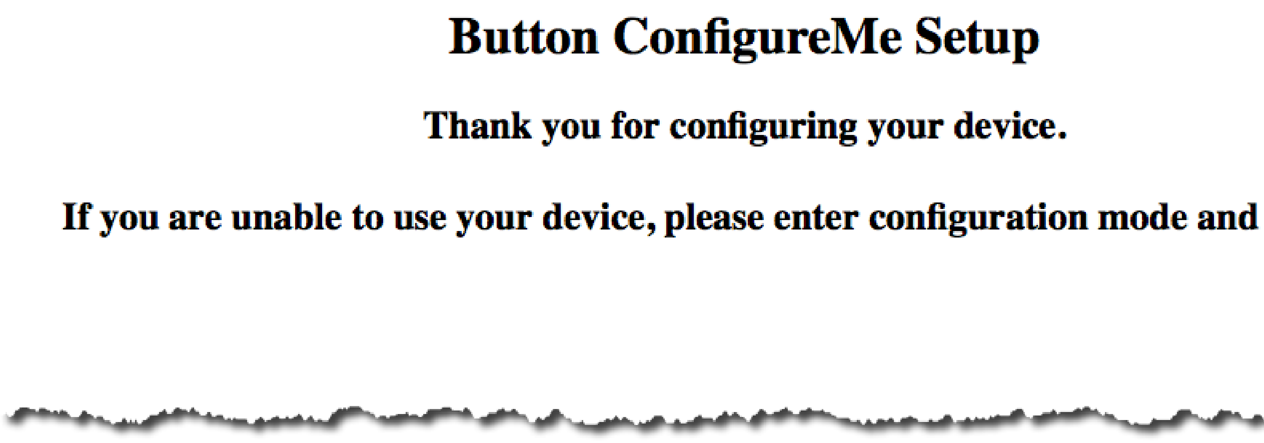
SSID	Guest
Security	✓ Open Network(No Password)
Password	None (unsecured)

AWS IoT Configuration:

Certificate	Browse... certificate.pem
Private Key	Browse... private.key
Endpoint Subdomain	A3T2RR9XSNT91O
Endpoint Region	us-west-2
Final Endpoint	[redacted].iot.us-west-2.amazonaws.com

✓ By clicking this box, you agree to the AWS IoT Button Terms and Conditions.

A configuration confirmation page will be displayed.



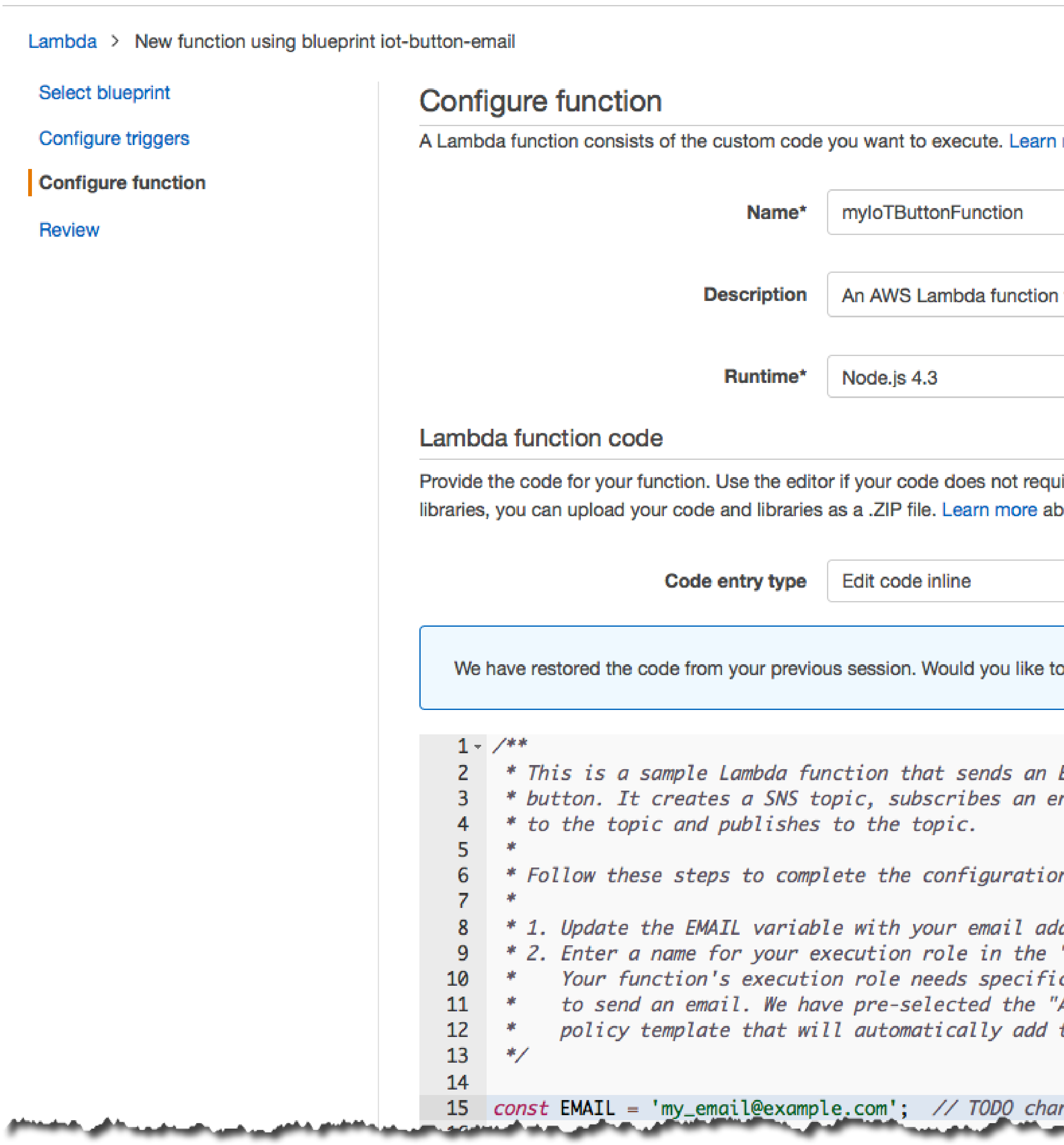
Button ConfigureMe Setup

Thank you for configuring your device.

If you are unable to use your device, please enter configuration mode and

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.



Lambda > New function using blueprint iot-button-email

Select blueprint

Configure triggers

Configure function

Review

Configure function

A Lambda function consists of the custom code you want to execute. Learn

Name* myIoTButtonFunction

Description An AWS Lambda function

Runtime* Node.js 4.3

Lambda function code

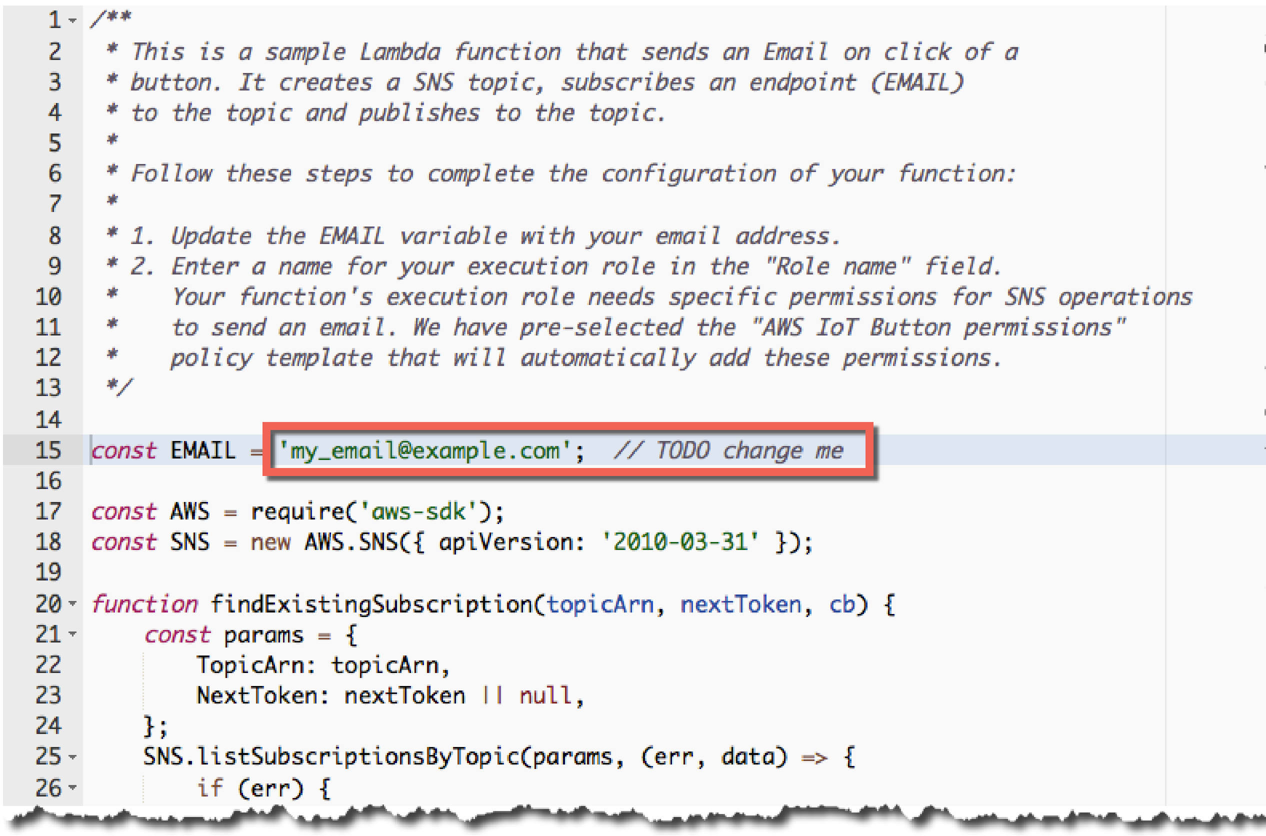
Provide the code for your function. Use the editor if your code does not requi
libraries, you can upload your code and libraries as a .ZIP file. Learn more ab

Code entry type Edit code inline

We have restored the code from your previous session. Would you like to

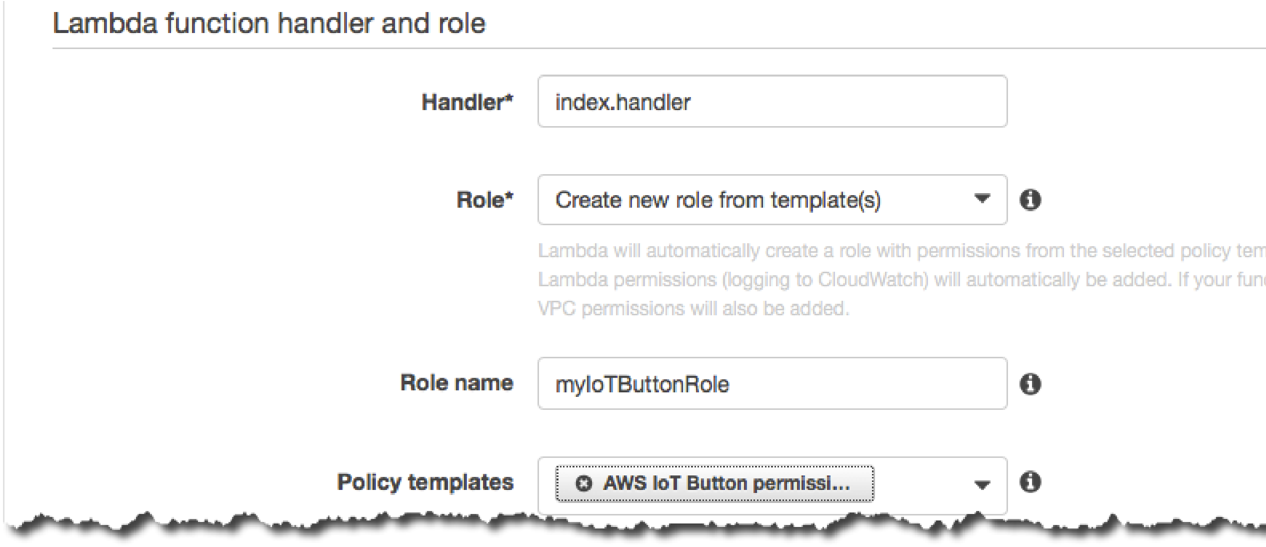
```
/**
 * This is a sample Lambda function that sends an E
 * button. It creates a SNS topic, subscribes an er
 * to the topic and publishes to the topic.
 *
 * Follow these steps to complete the configuratior
 *
 * 1. Update the EMAIL variable with your email add
 * 2. Enter a name for your execution role in the '
 *    Your function's execution role needs specific
 *    to send an email. We have pre-selected the "A
 *    policy template that will automatically add t
 */

const EMAIL = 'my_email@example.com';  // TODO char
```

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.



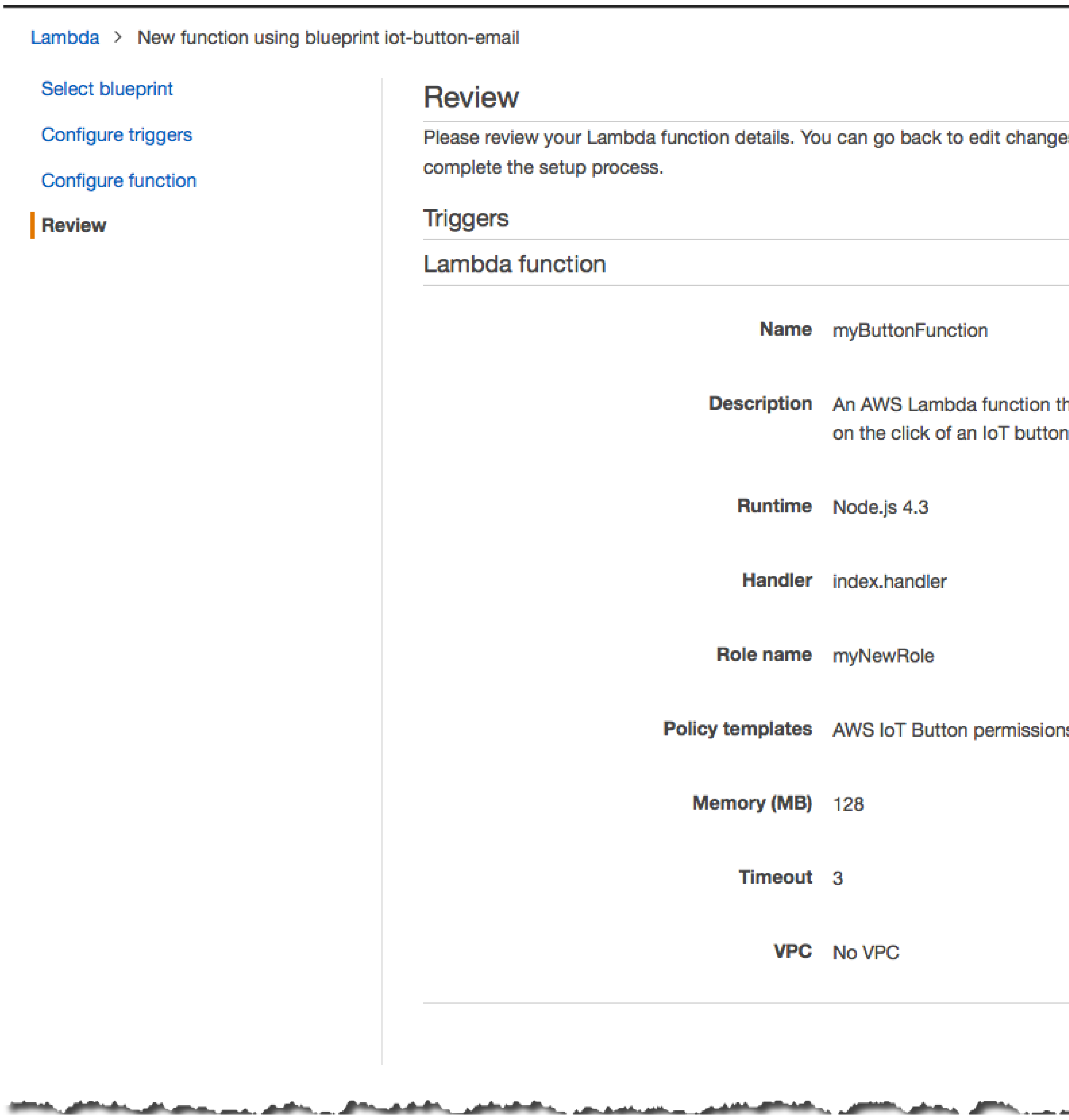
Lambda function handler and role

Handler*	index.handler
Role*	Create new role from template(s)
Role name	myIoTButtonRole
Policy templates	AWS IoT Button permissi...

Lambda will automatically create a role with permissions from the selected policy tem
Lambda permissions (logging to CloudWatch) will automatically be added. If your fun
VPC permissions will also be added.

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

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



Review

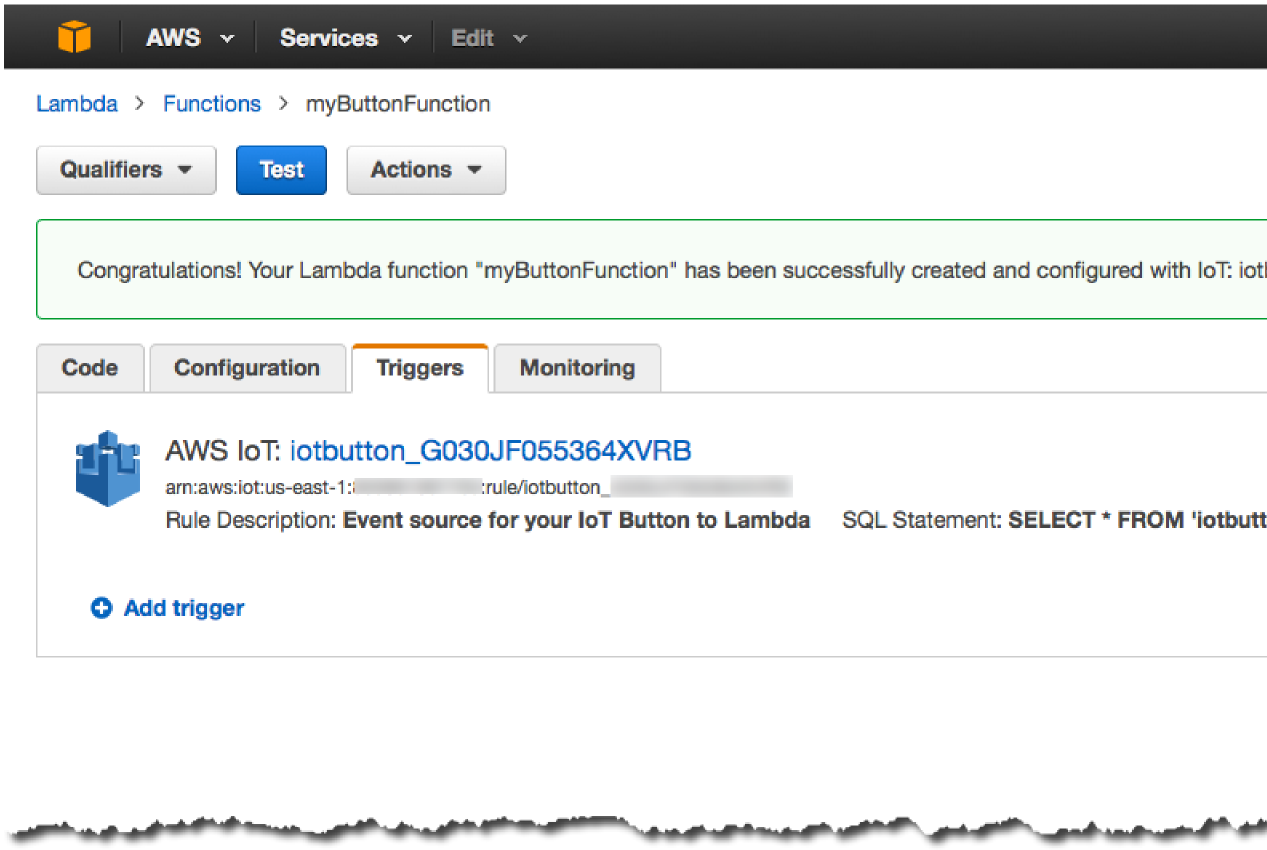
Please review your Lambda function details. You can go back to edit change
complete the setup process.

Triggers

Lambda function

Name	myButtonFunction
Description	An AWS Lambda function th on the click of an IoT button
Runtime	Node.js 4.3
Handler	index.handler
Role name	myNewRole
Policy templates	AWS IoT Button permission
Memory (MB)	128
Timeout	3
VPC	No VPC

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



Congratulations! Your Lambda function "myButtonFunction" has been successfully created and configured with IoT: iot

AWS IoT: iotbutton_G030JF055364XVRB

arn:aws:iot:us-east-1:[redacted]:rule/iotbutton_[redacted]

Rule Description: **Event source for your IoT Button to Lambda** SQL Statement: **SELECT * FROM 'iotbutt**

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 **Resources** page, choose **Create a certificate**.
e. Select the **Activate** box, and then choose **1-Click certificate create**.
f. Choose **Download private key**, and then choose **Download certificate**.
g. Select the box that represents the AWS IoT certificate (the box with the handshake icon).
h. In the **Detail** 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 **Decativate**, 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 section will guide you through the creation of resources required to send, receive, and process MQTT messages from devices using AWS IoT. You will need a computer with Wi-Fi access to complete this tutorial. 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 complete this tutorial. For more information about AWS IoT, see What Is AWS IoT (p. 1).

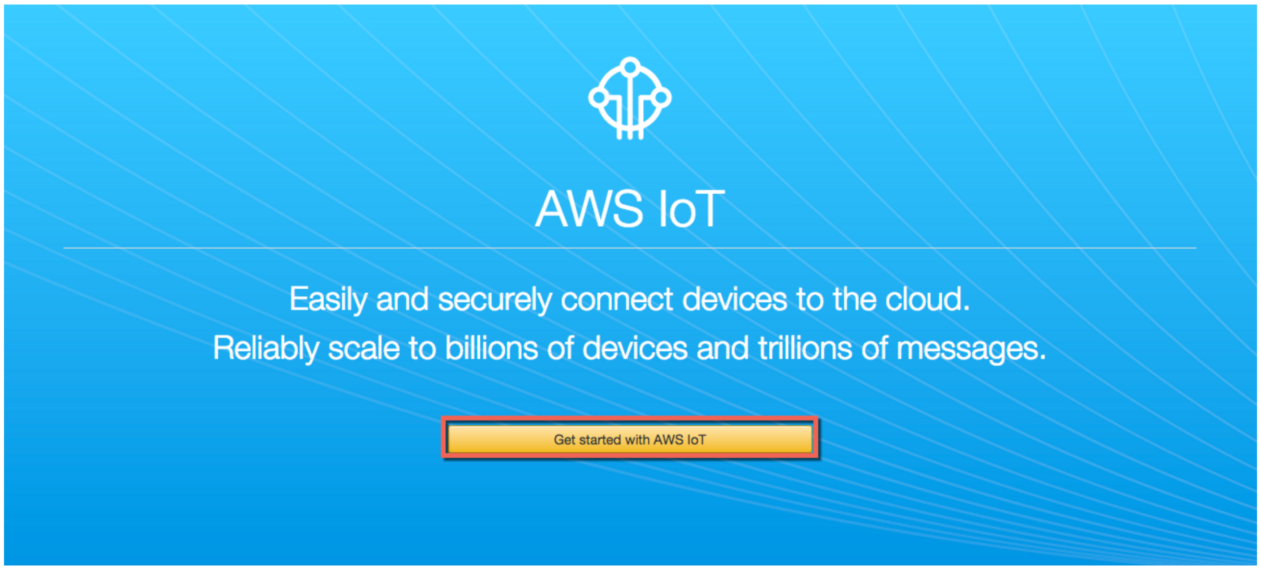
Note
This tutorial uses Amazon Simple Notification Service (Amazon SNS), which is not available in all regions. When you create AWS resources for this tutorial, make sure to sign in to the US East (N. Virginia) Region. For more information about AWS regions, see Regions and Endpoints.

Topics

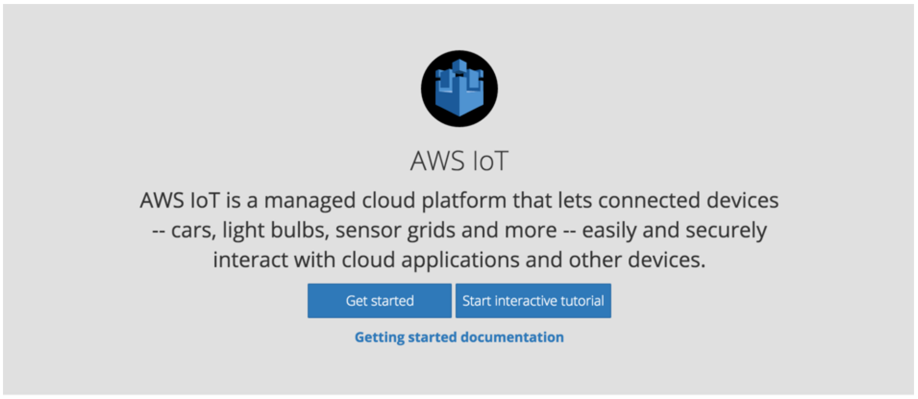
Sign in to the AWS IoT Console

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

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 with AWS IoT**.


AWS IoT
Easily and securely connect devices to the cloud.
Reliably scale to billions of devices and trillions of messages.
Get started with AWS IoT

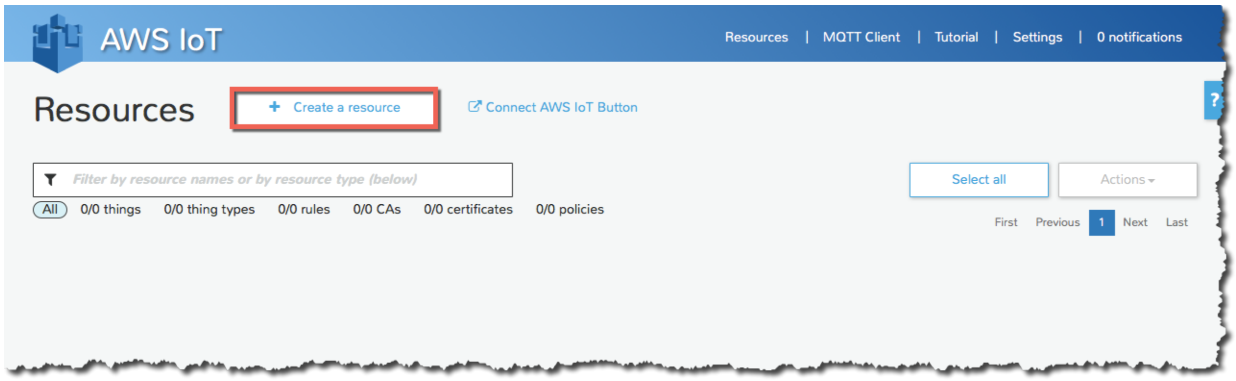
5. If this is your first time using the AWS IoT console, you will see two options: **Get started** and **Start interactive tutorial**. Choose **Get Started**.



AWS IoT

AWS IoT is a managed cloud platform that lets connected devices -- cars, light bulbs, sensor grids and more -- easily and securely interact with cloud applications and other devices.

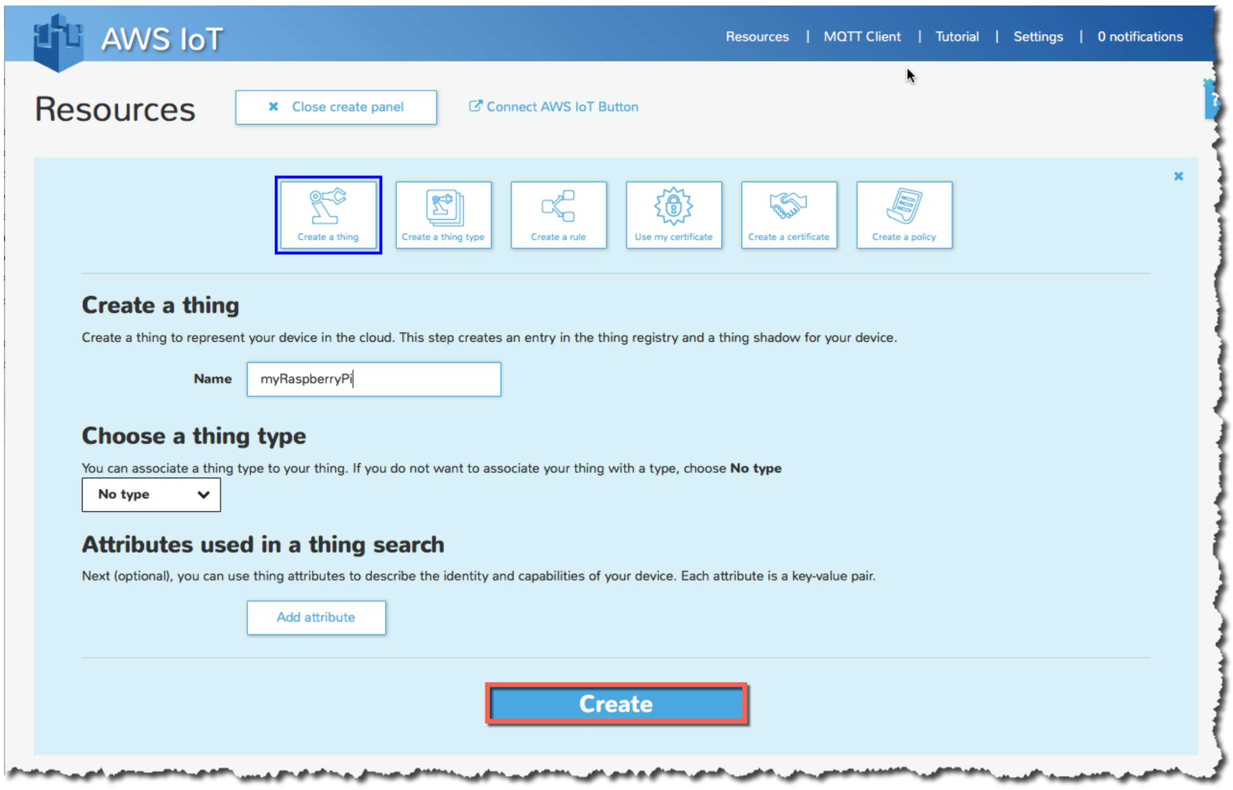
6. On the **Resources** page, if you don't see a blue banner with **Create a thing**, **Create a rule**, **Create a certificate**, and **Create a policy** buttons, choose **Create a resource**.


AWS IoT
Resources | MQTT Client | Tutorial | Settings | 0 notifications
Resources
Create a resource
Connect AWS IoT Button
Filter by resource names or by resource type (below)
All 0/0 things 0/0 thing types 0/0 rules 0/0 CAs 0/0 certificates 0/0 policies
Select all
Actions
First Previous 1 Next Last

Create a Device in the Thing Registry

To connect a device to AWS IoT, we recommend that you first create a device in the thing registry. This registry allows you to keep a record of all of the devices that are connected to your AWS IoT account.

1. Choose **Create a thing**, and then type a name for your device. You can also choose **Add attribute** to provide information about your device (for example, its serial number, manufacturer, and more). Choose **Create** to add your device to the thing registry.



Create a thing

Create a thing to represent your device in the cloud. This step creates an entry in the thing registry and a thing shadow for your device.

Name: myRaspberryPi

Choose a thing type

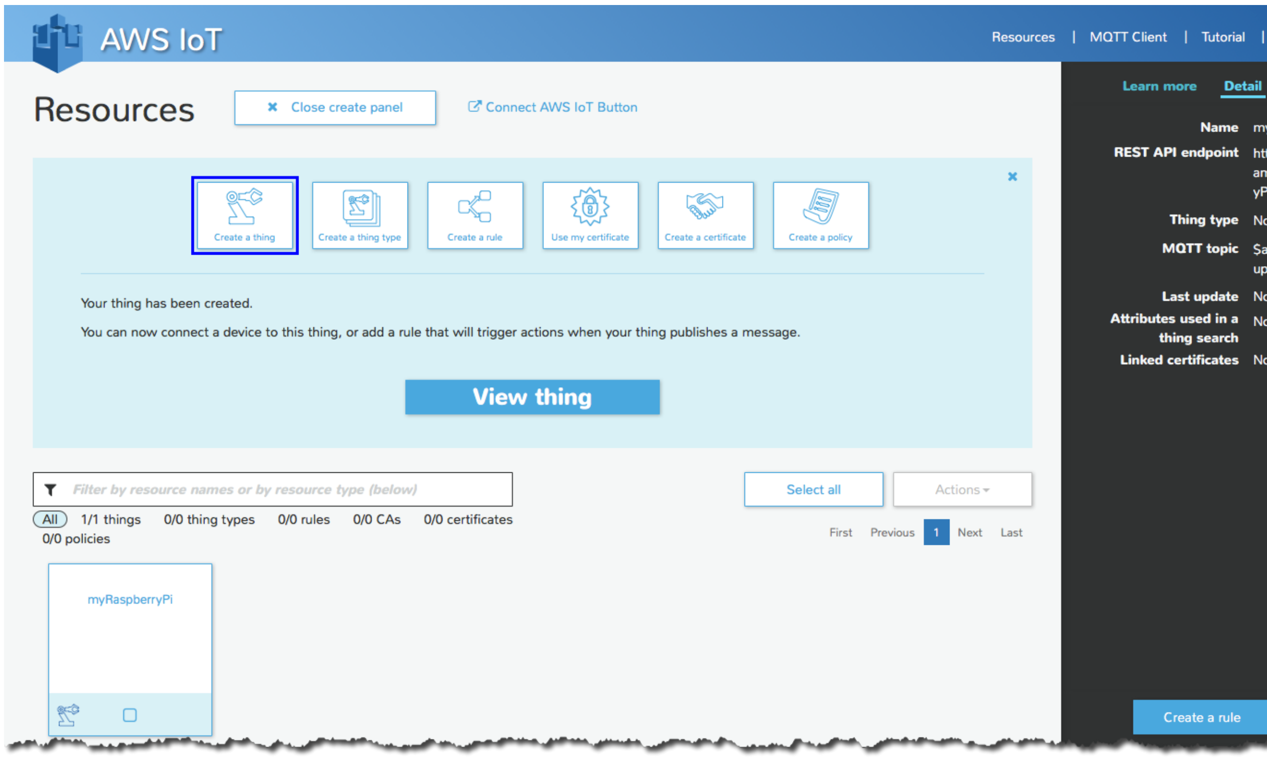
You can associate a thing type to your thing. If you do not want to associate your thing with a type, choose **No type**

No type

Attributes used in a thing search

Next (optional), you can use thing attributes to describe the identity and capabilities of your device. Each attribute is a key-value pair.

2. Choose **View thing** to display information about your device.



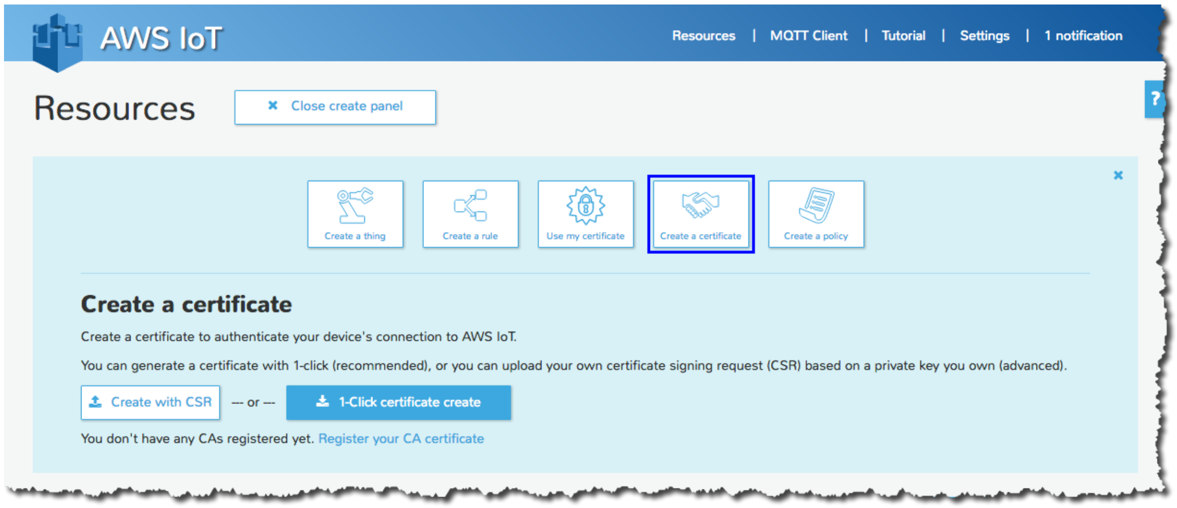
Your thing has been created.

You can now connect a device to this thing, or add a rule that will trigger actions when your thing publishes a message.

Create and Activate a Device Certificate

Communication between your AWS IoT button 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. In the **Create a Certificate** section, choose **1-Click certificate create**.



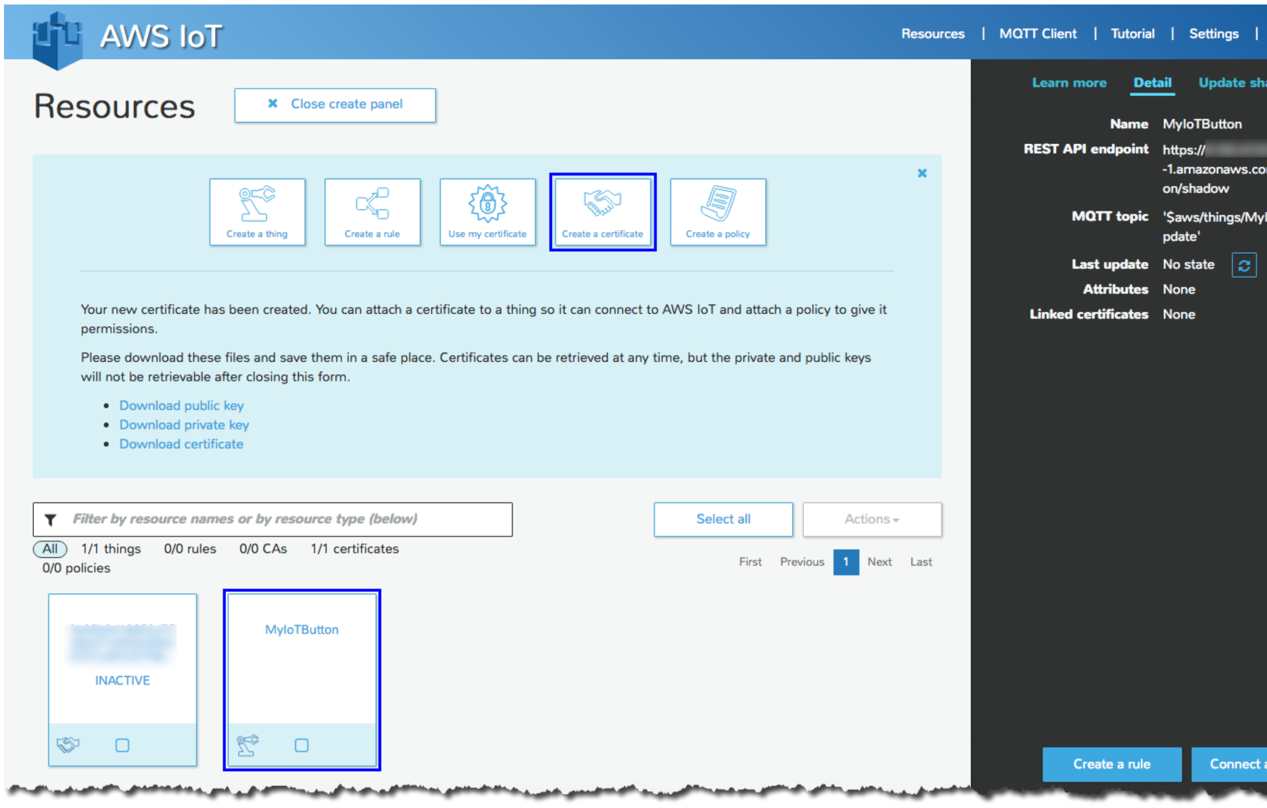
Create a certificate

Create a certificate to authenticate your device's connection to AWS IoT.

You can generate a certificate with 1-click (recommended), or you can upload your own certificate signing request (CSR) based on a private key you own (advanced).

You don't have any CAs registered yet. Register your CA certificate

2. On the **Resources** page, choose the **Download private key** and **Download certificate** links, and then save the private key and certificate to your computer.



Your new certificate has been created. You can attach a certificate to a thing so it can connect to AWS IoT and attach a policy to give it permissions.

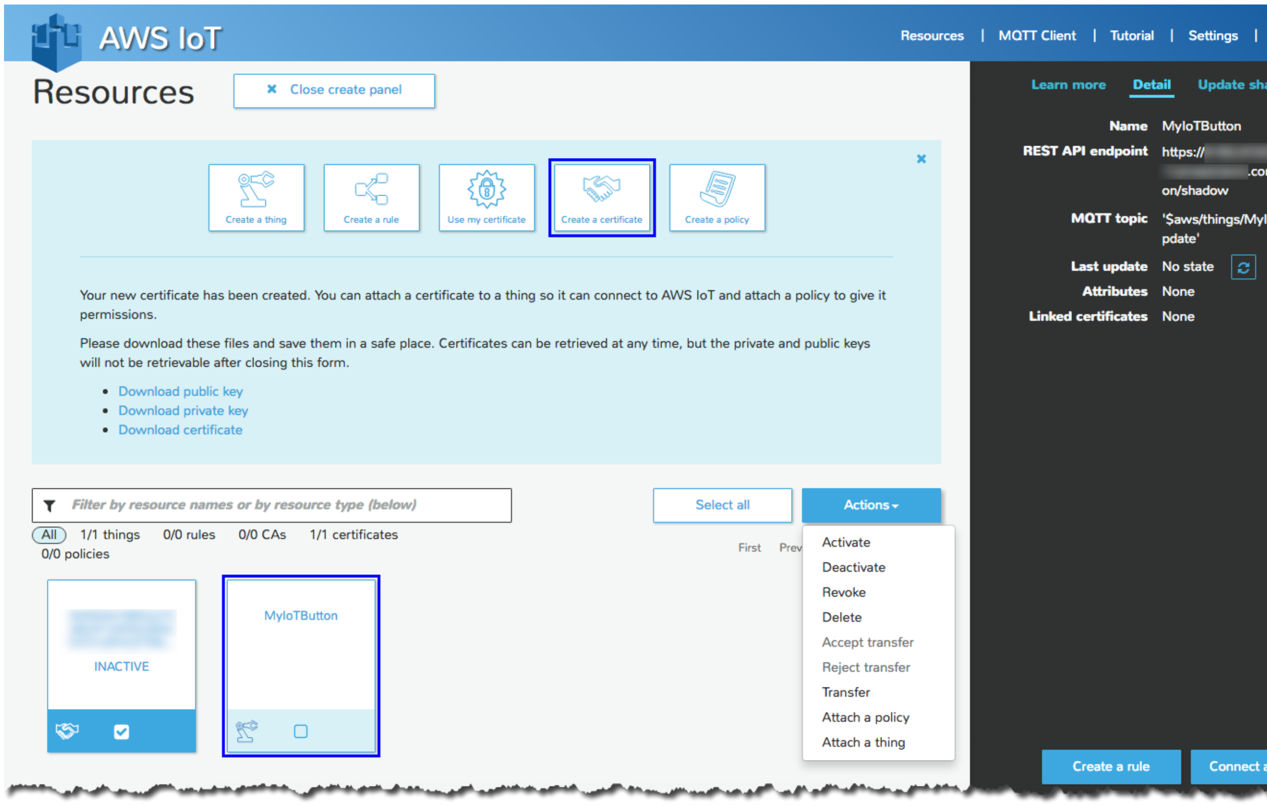
Please download these files and save them in a safe place. Certificates can be retrieved at any time, but the private and public keys will not be retrievable after closing this form.

- Download public key
- Download private key
- Download certificate

Filter by resource names or by resource type (below)
Select all
Actions
All 1/1 things 0/0 rules 0/0 CAs 1/1 certificates 0/0 policies
First Previous 1 Next Last
INACTIVE
MyIoTButton
Learn more
Detail
Update sh

Name	MyIoTButton
REST API endpoint	https://[redacted] -1.amazonaws.co on/shadow
MQTT topic	'$aws/things/MyI pdate'
Last update	No state
Attributes	None
Linked certificates	None

3. Select the check box on the certificate, and from the **Actions** menu, choose **Activate**.



Your new certificate has been created. You can attach a certificate to a thing so it can connect to AWS IoT and attach a policy to give it permissions.

Please download these files and save them in a safe place. Certificates can be retrieved at any time, but the private and public keys will not be retrievable after closing this form.

- Download public key
- Download private key
- Download certificate

Name	MyIoTButton
REST API endpoint	https://[redacted] .co on/shadow
MQTT topic	'$aws/things/MyI pdate'
Last update	No state
Attributes	None
Linked certificates	None

Create an AWS IoT Policy

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

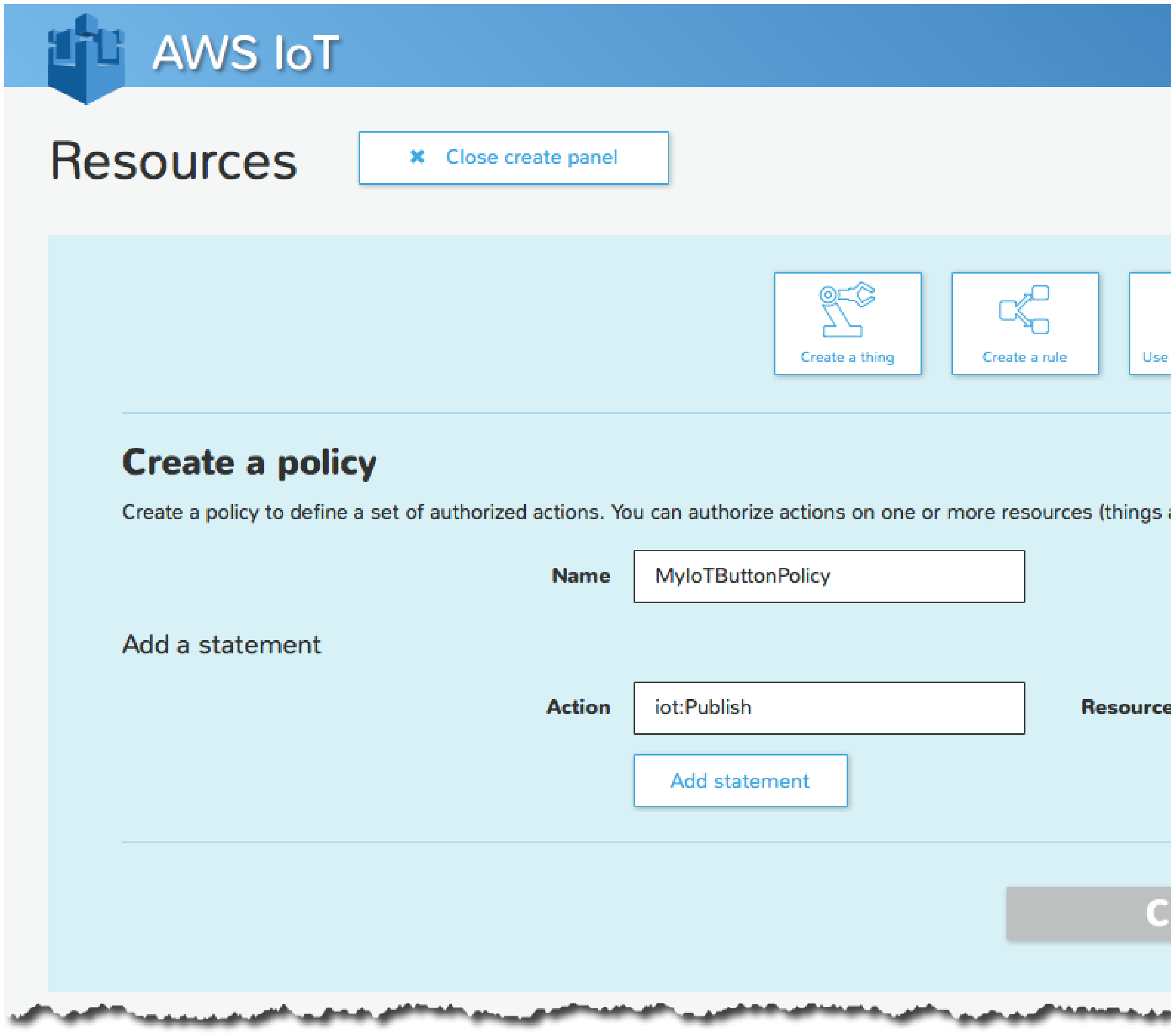
1. In the AWS IoT console, if you don't see the **Create** panel, choose **Create a resource**.
2. Choose **Create a policy**.
3. In the **Create a policy** section, type a name for the policy. From the **Action** menu, choose **iot:Publish**. In the **Resource** field, type the ARN of your AWS IoT button, and then select the **Allow** check box. This allows your button to publish messages to AWS IoT.

   Note
   The 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.

   The settings explained in this step assume you are using an AWS IoT button which is programmed to publish on a specific MQTT topic: `topic/iotbutton/`*`button-serial-number`*. The policy created gives permission to publish to that topic. If you are not using an AWS IoT button, you should modify the ARN described above to contain the MQTT topic on which your device publishes. If your device is programmed to publish on `myDevice/myTopic` you would use the following ARN:

   `arn:aws:iot:us-east-1:123456789012:topic/myDevice/myTopic`.

   Choose Add statement, and then choose Create.



Create a policy

Create a policy to define a set of authorized actions. You can authorize actions on one or more resources (things

Name: MyIoTButtonPolicy

Add a statement

Action: iot:Publish

Resource

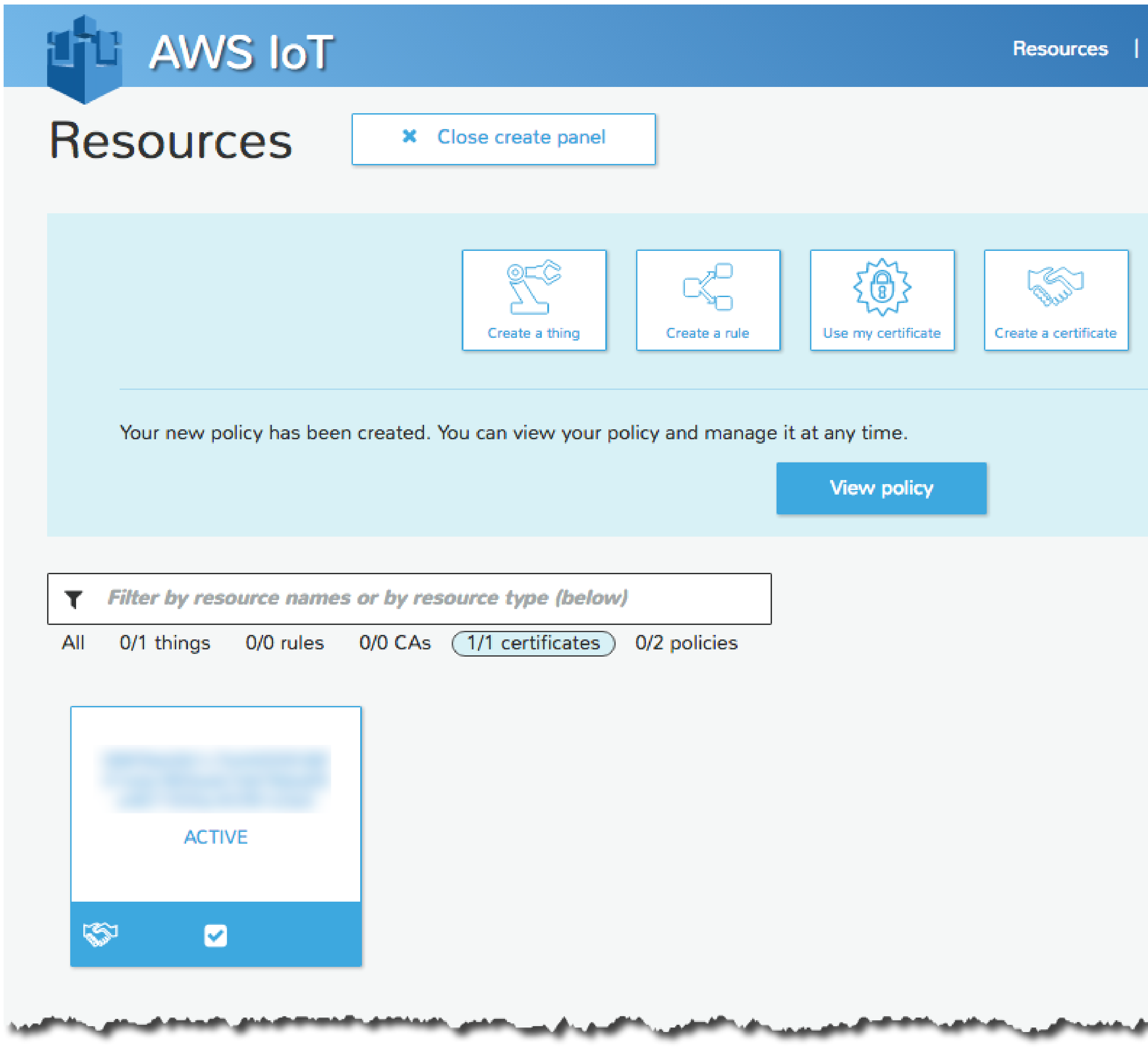
Add statement

For more information about AWS IoT policies, see Managing AWS IoT Policies.

Attach an AWS IoT Policy to a Device Certificate

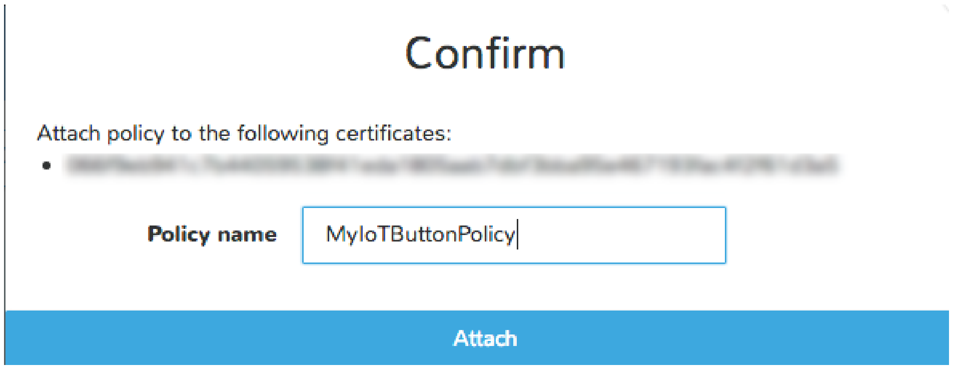
Now that you have created a policy, you must attach it to your device certificate. Attaching an AWS IoT policy to a certificate gives the device the permissions specified in the policy.

1. From the AWS IoT console, choose your device certificate, and from the **Actions** menu, choose **Attach a policy**.



Your new policy has been created. You can view your policy and manage it at any time.

2. In the **Confirm** dialog box, type the name of the AWS IoT policy you created in the previous step, and then choose **Attach**.



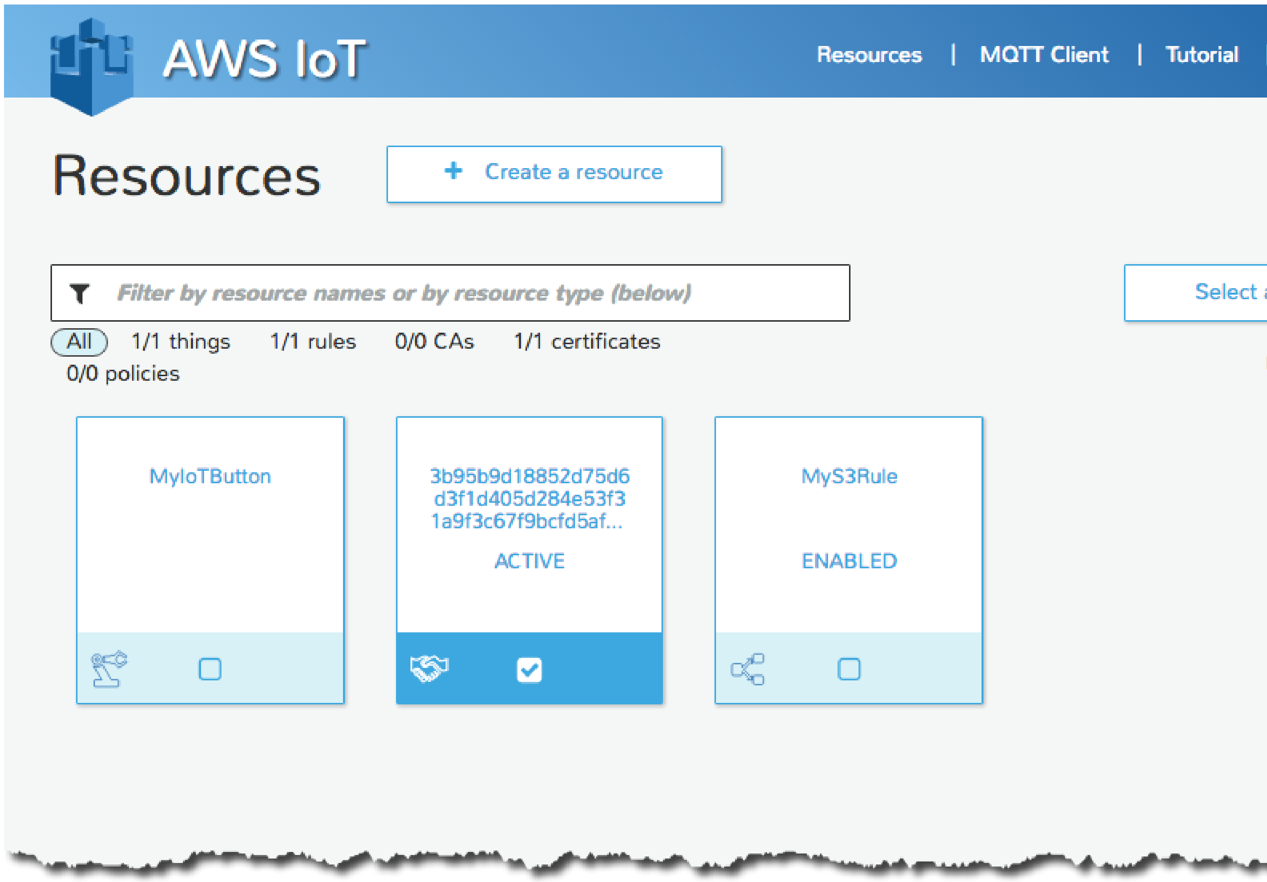
Attach policy to the following certificates:

Policy name: MyIoTButtonPolicy

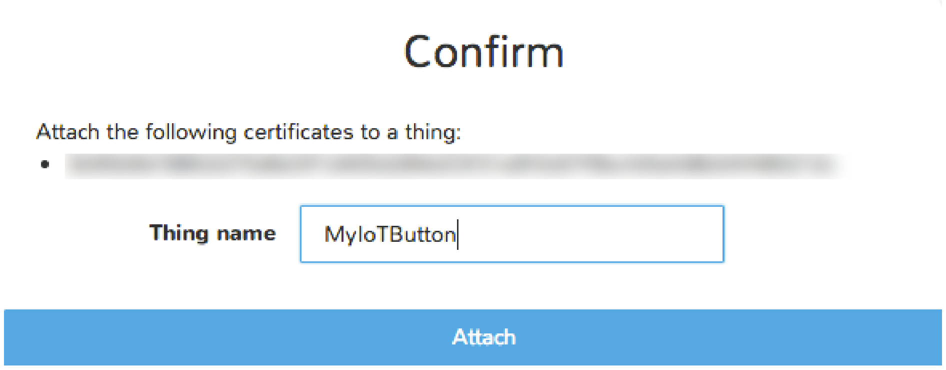
Attach a Thing to a Certificate

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

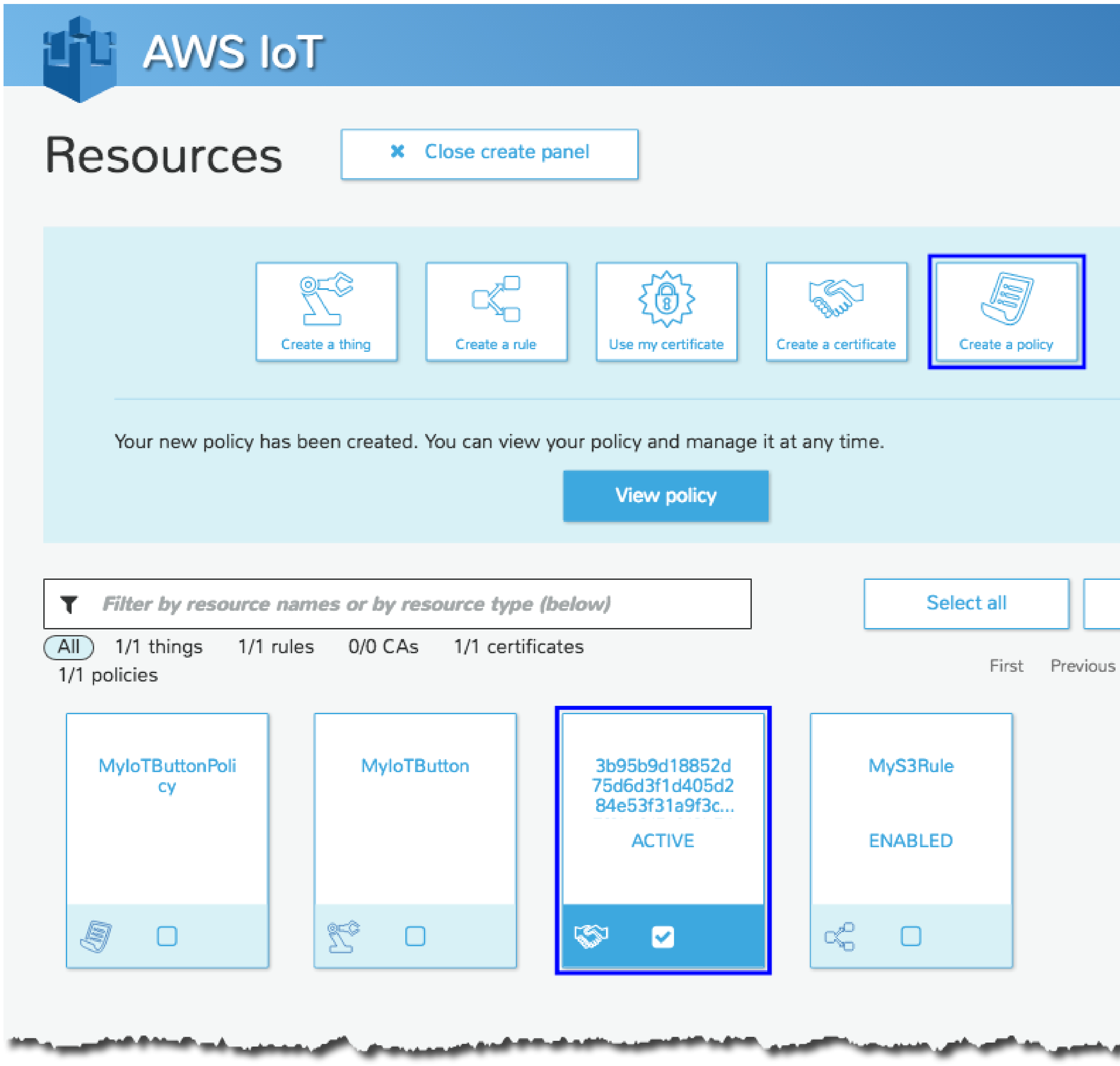
1. In the AWS IoT console, choose the certificate you want to attach, and from the **Actions** menu, choose **Attach a thing**.



2. In the **Confirm** dialog box, type the name of the thing to which you will attach the certificate, and then choose **Attach**.



3. To verify the thing is attached, double-click the certificate. The policy and thing should appear in the detail pane.



Your new policy has been created. You can view your policy and manage it at any time.

Configure Your Device

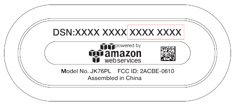
Configuring your device allows it to connect to your Wi-Fi network. Your device must be connected to your Wi-Fi network to install the device certificate and to send messages to AWS IoT. All devices must have a device certificate in order to communicate with AWS IoT.

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 for 15 seconds until a blue blinking light appears.
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:


DSN:XXXX XXXX XXXX XXXX
powered by amazon web services
Model No. JK76PL FCC ID: 2ACBE-0610
Assembled in China

Copy your device certificate onto your AWS IoT button

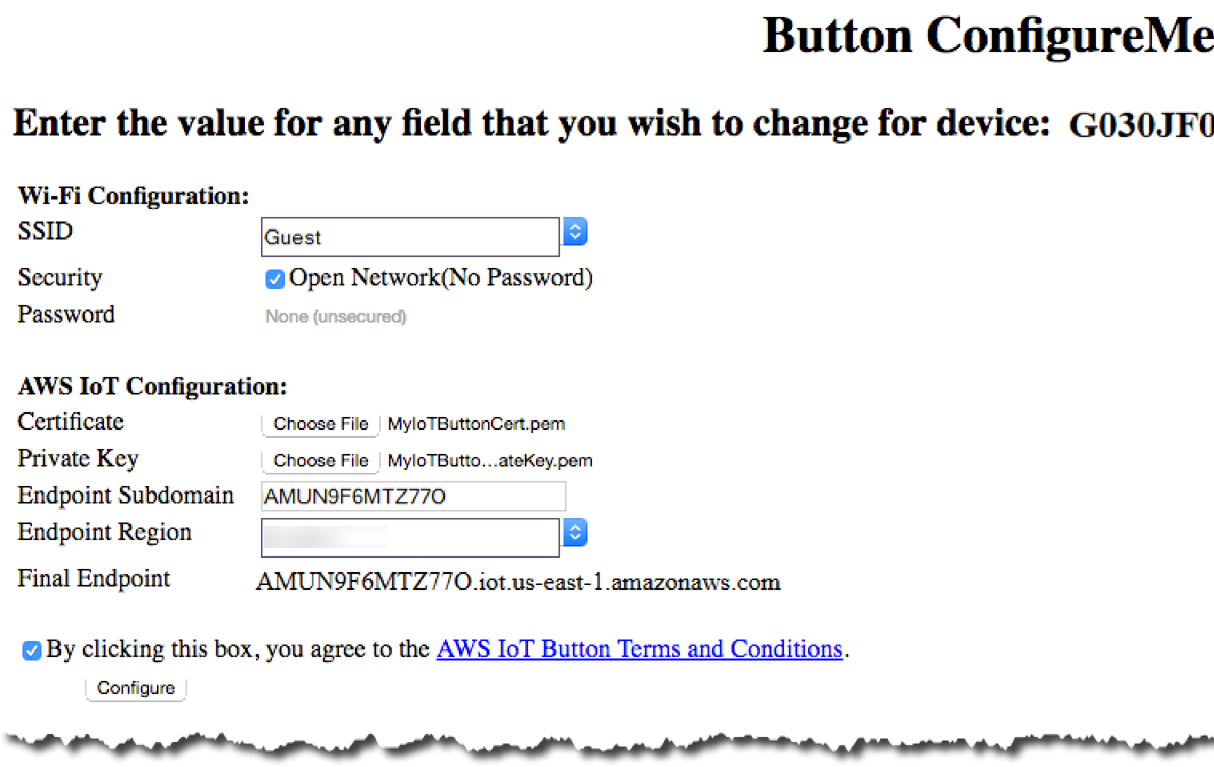
To connect to AWS IoT, you must copy your device certificate onto the AWS IoT button.

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

   1. Type your Wi-Fi SSID and password.
   2. Browse to and select your certificate and private key.
   3. Find your custom endpoint in the AWS IoT console. Your endpoint will look something like the following:

      ```
      ABCDEFG1234567.iot.us-east-1.amazonaws.com
      ```

      where `ABCDEFG1234567` is the subdomain and `us-east-1` is the region.
   4. On the **Button ConfigureMe** page, type the subdomain, and then choose the region that matches the region in your AWS IoT endpoint.
   5. Select the **Terms and Conditions** check box. Your settings should now look like the following:



Button ConfigureMe

Enter the value for any field that you wish to change for device: G030JF0

Wi-Fi Configuration:	
SSID	Guest
Security	✓ Open Network(No Password)
Password	None (unsecured)
AWS IoT Configuration:	
Certificate	Choose File MyIoTButtonCert.pem
Private Key	Choose File MyIoTButto...ateKey.pem
Endpoint Subdomain	AMUN9F6MTZ77O
Endpoint Region	
Final Endpoint	AMUN9F6MTZ77O.iot.us-east-1.amazonaws.com

✓ By clicking this box, you agree to the AWS IoT Button Terms and Conditions.

6. Your button should now connect to your Wi-Fi.

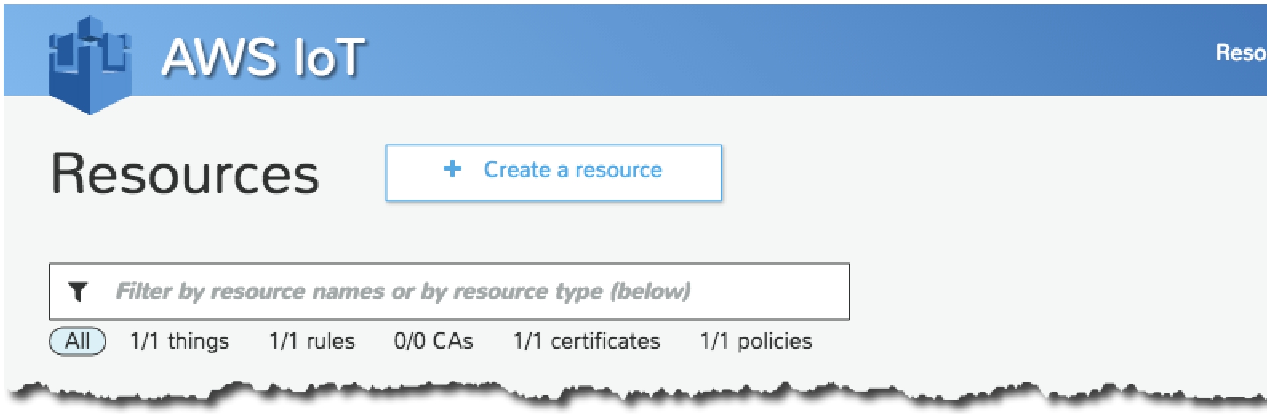
View Device MQTT Messages with the AWS IoT MQTT Client

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

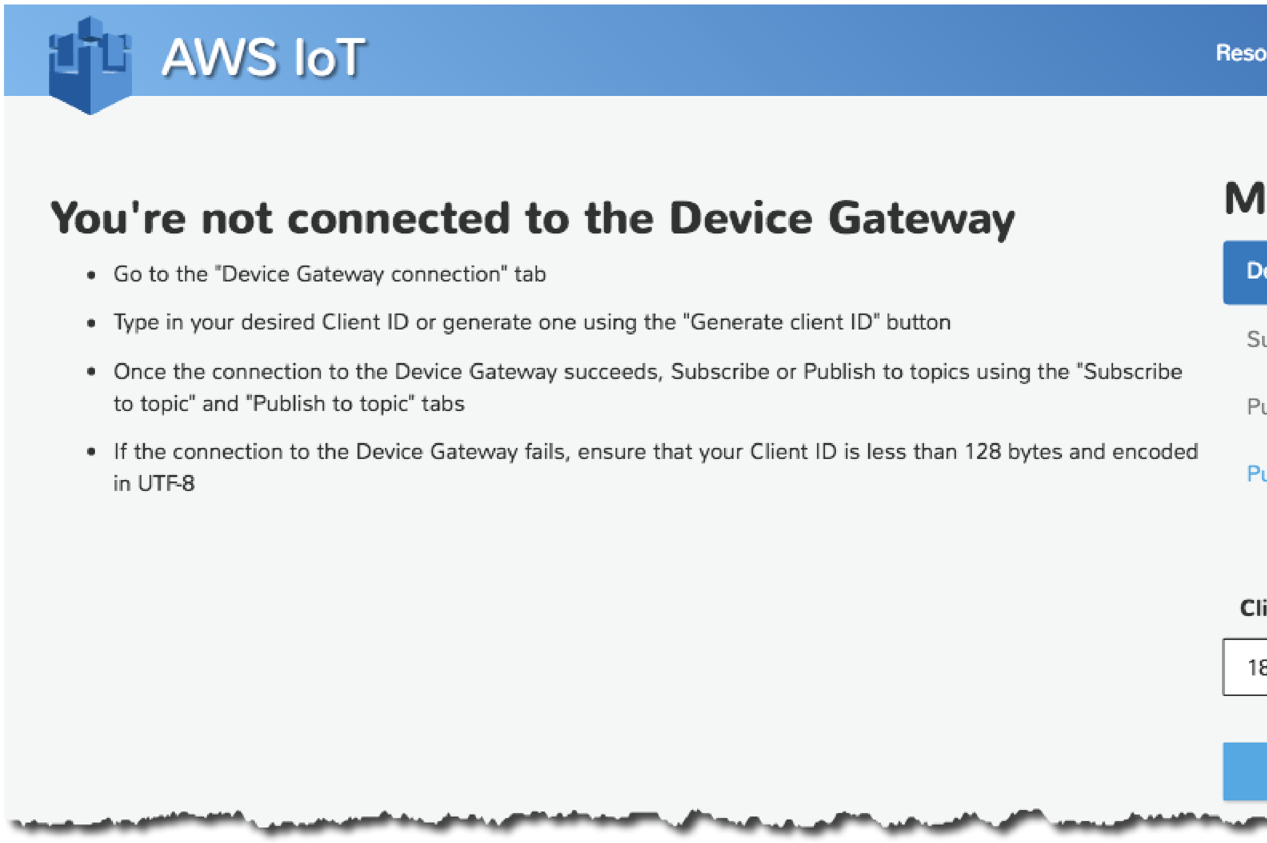
Devices publish MQTT messages on topics. You can use the AWS IoT MQTT client to subscribe to these topics to see the content of these messages.

To view MQTT messages:

1. In the AWS IoT console, choose **MQTT Client**.



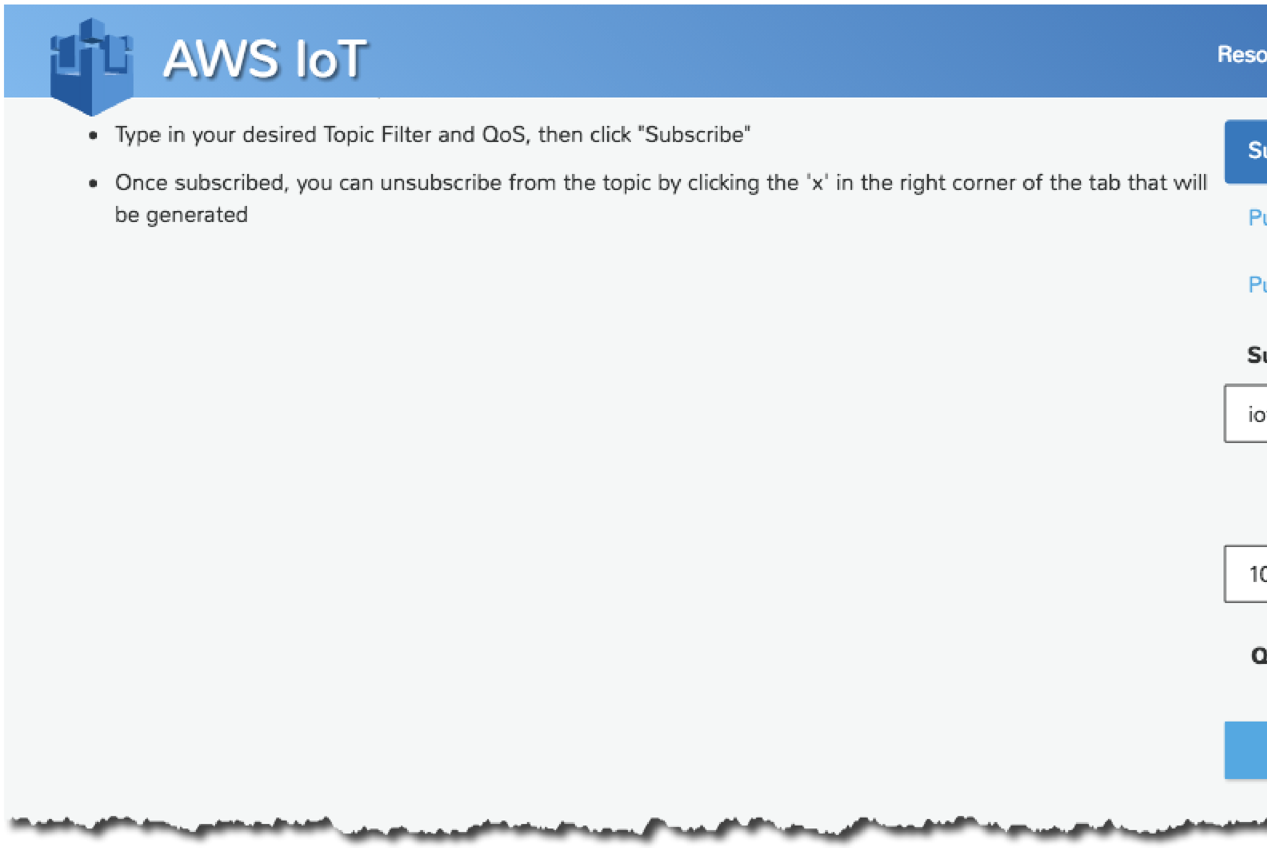
2. Type a client ID or choose **Generate client ID**, and then choose **Connect**.



You're not connected to the Device Gateway

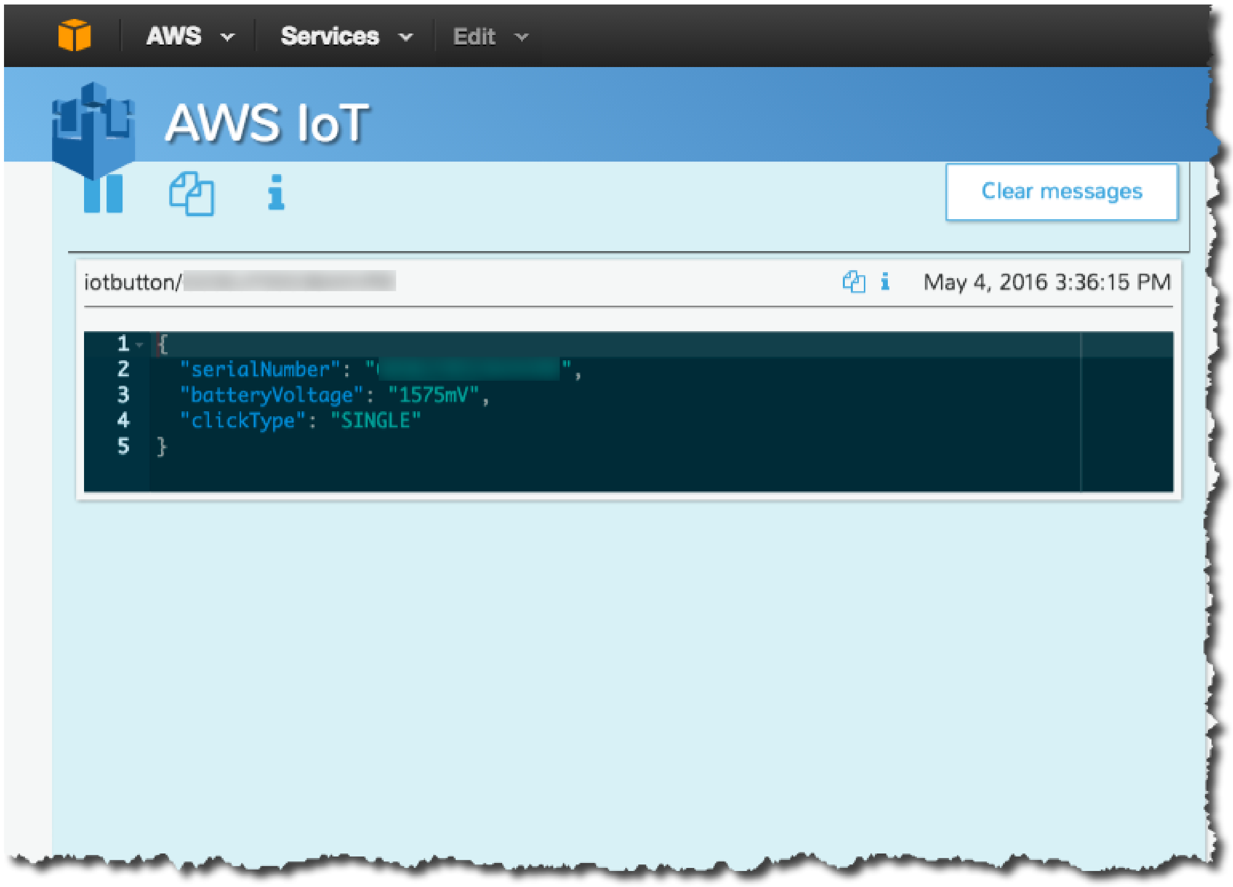
- Go to the "Device Gateway connection" tab
- Type in your desired Client ID or generate one using the "Generate client ID" button
- Once the connection to the Device Gateway succeeds, Subscribe or Publish to topics using the "Subscribe to topic" and "Publish to topic" tabs
- If the connection to the Device Gateway fails, ensure that your Client ID is less than 128 bytes and encoded in UTF-8

3. Subscribe to the topic on which your thing publishes. In the case of the AWS IoT button, you can subscribe to `iotbutton/+`. Choose **Subscribe to topic**, in **Subscription topic**, type **`iotbutton/+`**, and then choose **Subscribe**.



- Type in your desired Topic Filter and QoS, then click "Subscribe"
- Once subscribed, you can unsubscribe from the topic by clicking the 'x' in the right corner of the tab that will be generated

4. Press your AWS IoT button, and then view the message in the AWS IoT MQTT client.



iotbutton/[redacted]   May 4, 2016 3:36:15 PM

```
{
  "serialNumber": "[redacted]",
  "batteryVoltage": "1575mV",
  "clickType": "SINGLE"
}
```

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.

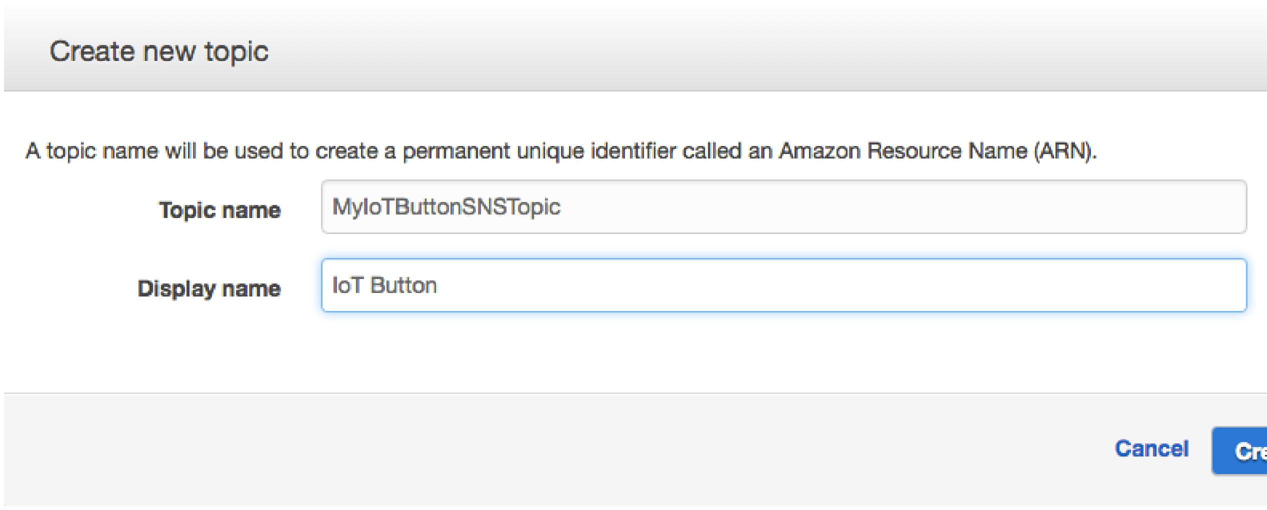
In the upper-right corner of this page, there is a **Filter View** drop-down list. You can choose **AWS IoT Button** to see instructions for testing your rule by using the AWS IoT button or **MQTT Client** to see instructions for testing your rule by using the AWS IoT MQTT client.

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 https://console.aws.amazon.com/sns/.
2. From the left pane, choose **Topics**, and on the right pane, choose **Create new topic**.
3. Type a topic name and a display name, and then choose **Create topic**.

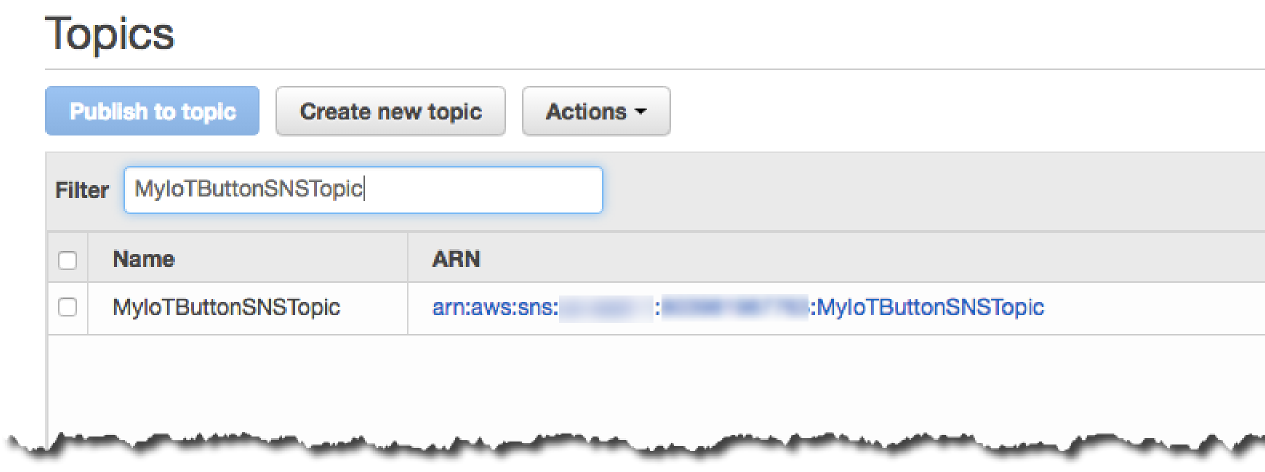

Create new topic

A topic name will be used to create a permanent unique identifier called an Amazon Resource Name (ARN).

Topic name: MyIoTButtonSNSTopic

Display name: IoT Button

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


Topics

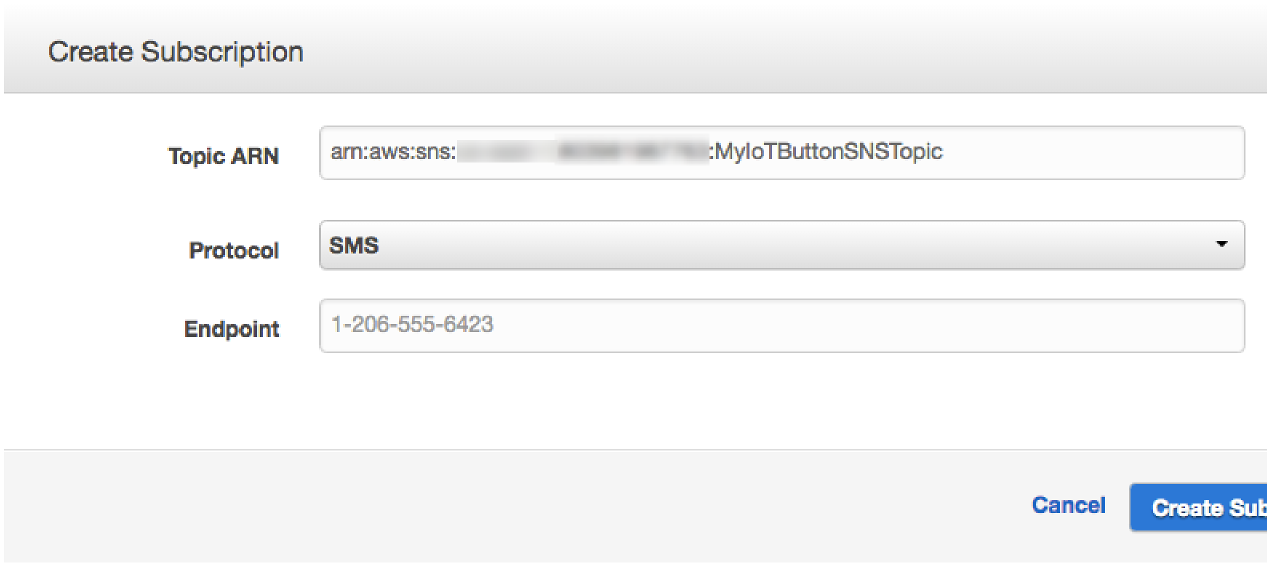
Filter: MyIoTButtonSNSTopic

Name	ARN
MyIoTButtonSNSTopic	arn:aws:sns:[redacted]:[redacted]:MyIoTButtonSNSTopic

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, from the **Actions** menu, choose **Subscribe to topic**.
2. From the **Protocol** drop-down list, choose **SMS**.



Create Subscription

Topic ARN: arn:aws:sns:[redacted]:MyIoTButtonSNSTopic

Protocol: SMS

Endpoint: 1-206-555-6423

3. In **Endpoint**, 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.

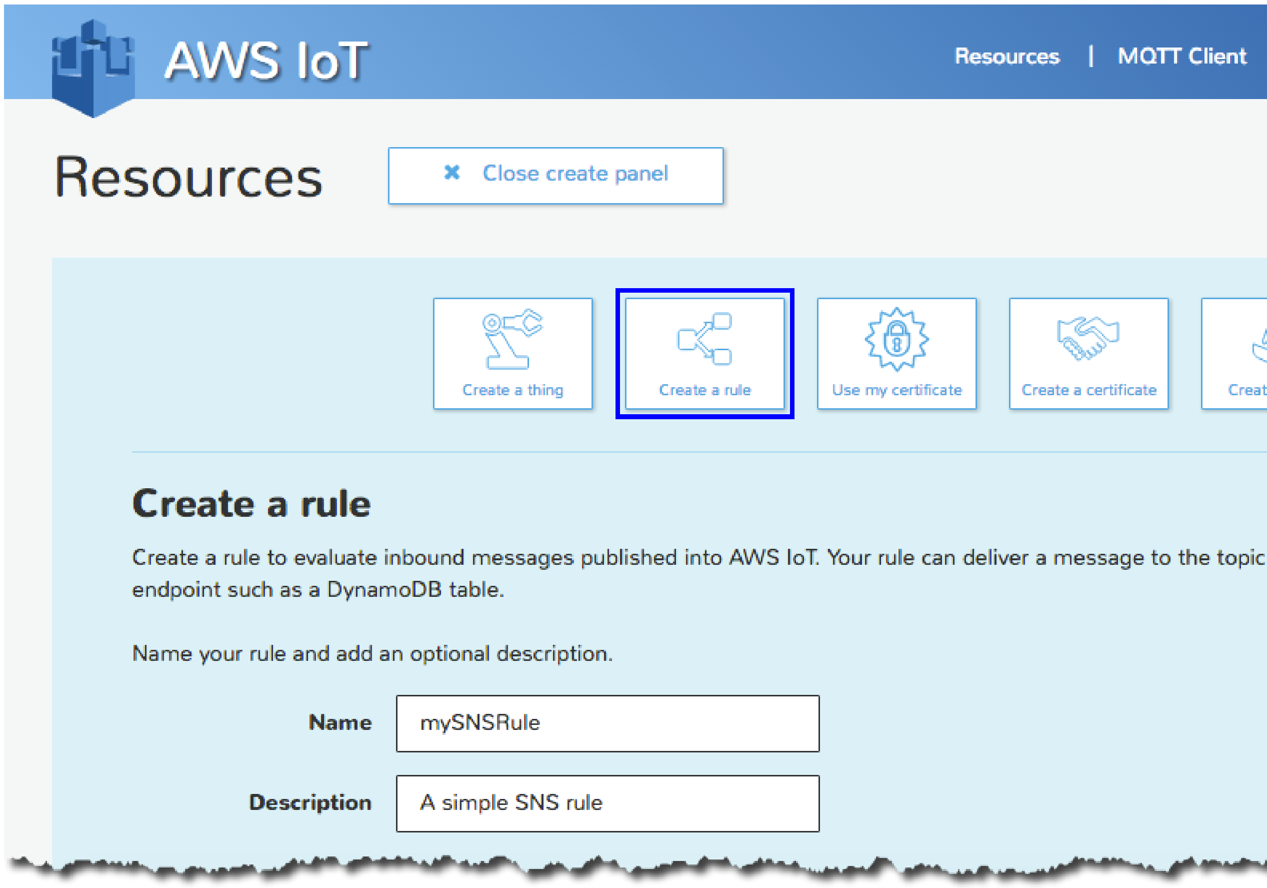
You will receive a text message that confirms you successfully created the subscription.

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, choose **Create a rule**.



Create a rule

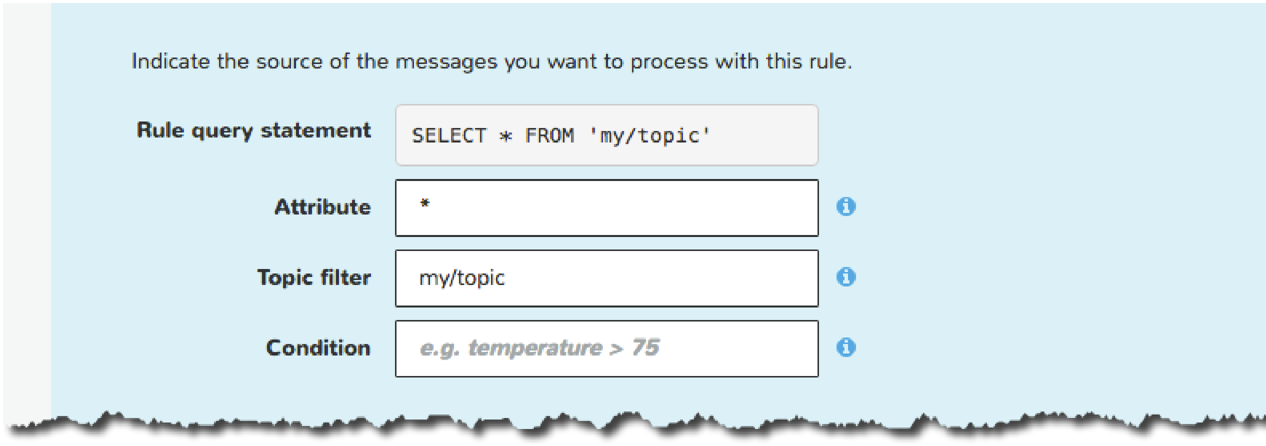
Create a rule to evaluate inbound messages published into AWS IoT. Your rule can deliver a message to the topic endpoint such as a DynamoDB table.

Name your rule and add an optional description.

Name	mySNSRule
Description	A simple SNS rule

2. On the **Create a rule** page, in **Name**, type a name for your rule.
3. In **Description**, type a description for the rule.
4. In **Attribute**, 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 **Topic filter**, type **`iotbutton/`*`your-button-DSN`***. If you are not using an AWS IoT button, type **`my/topic`**.

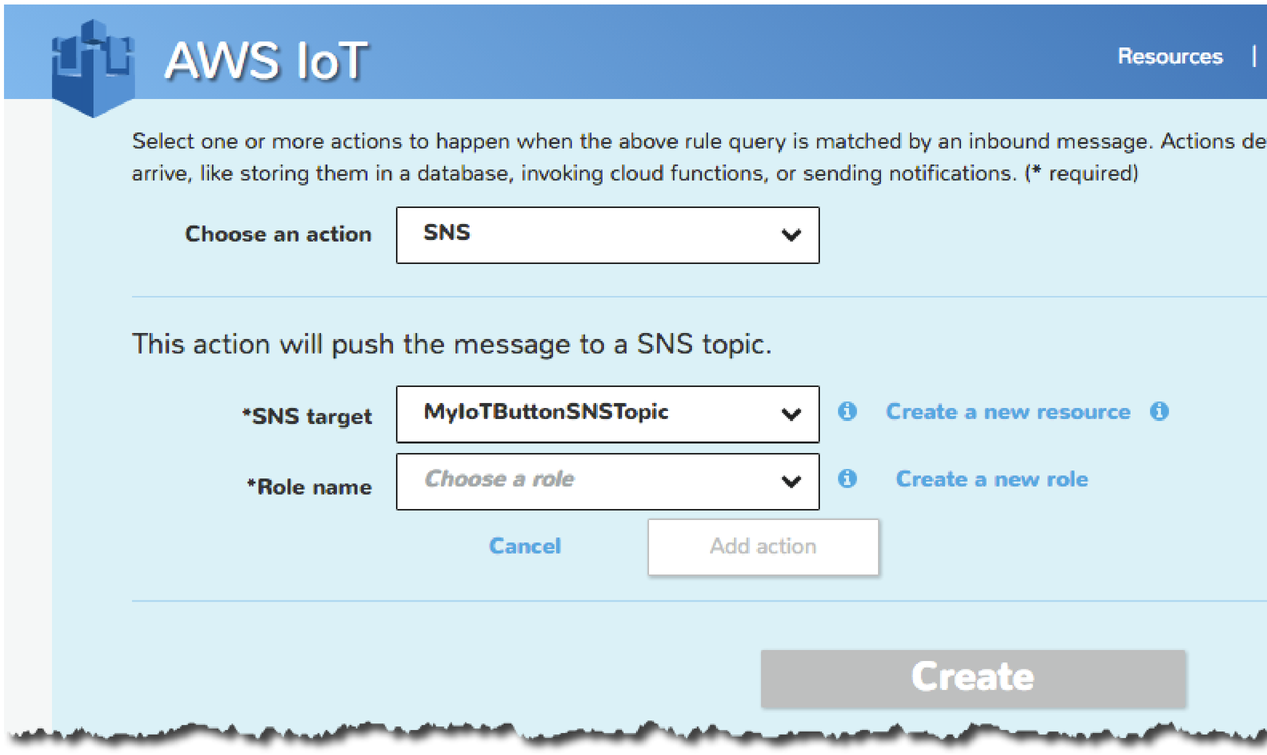
   Note
   You can find the DSN on the bottom of the button.
6. Leave **Condition** blank.



Indicate the source of the messages you want to process with this rule.

Rule query statement	`SELECT * FROM 'my/topic'`
Attribute	*
Topic filter	my/topic
Condition	e.g. temperature > 75

7. From the **Choose an action** drop-down list, choose **Send message as a push notification (SNS)**.
8. From the **SNS target** drop-down list, choose the Amazon SNS topic you created earlier.



Select one or more actions to happen when the above rule query is matched by an inbound message. Actions de
arrive, like storing them in a database, invoking cloud functions, or sending notifications. (* required)

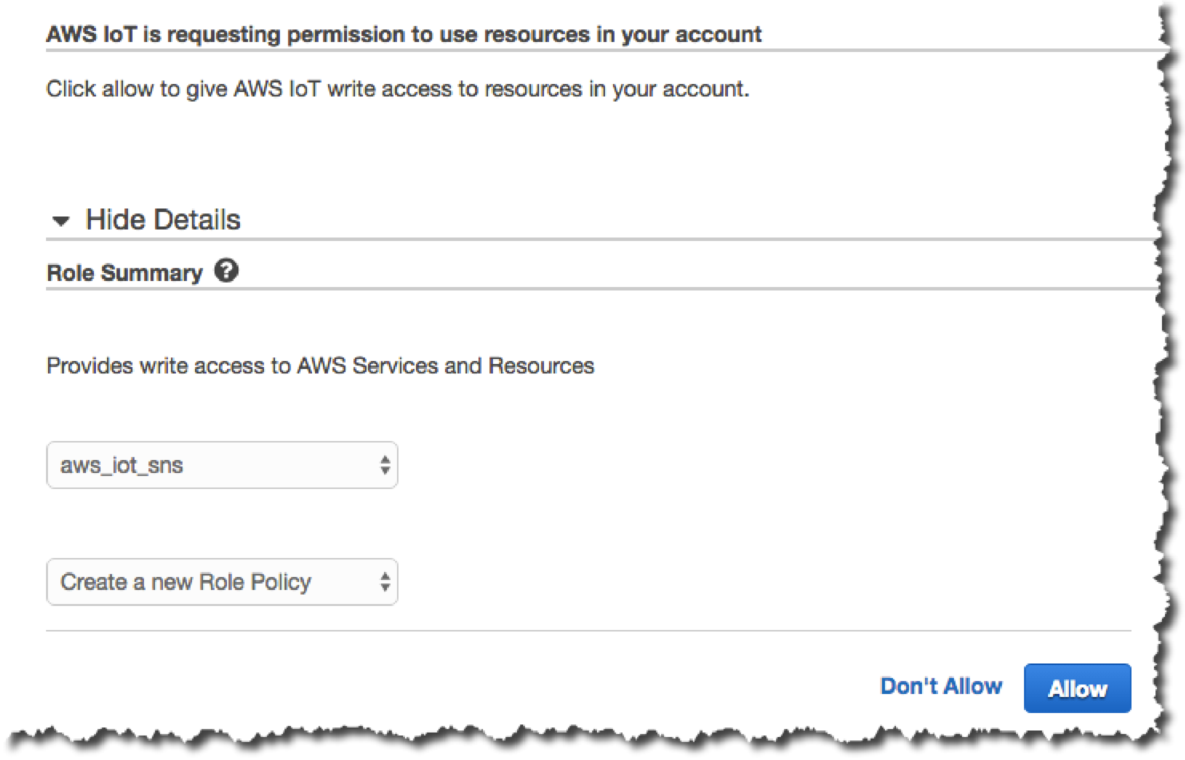
Choose an action: SNS

This action will push the message to a SNS topic.

*SNS target: MyIoTButtonSNSTopic

*Role name: Choose a role

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 the **Create a new role** link. This will open a web page in the IAM console.
10. Accept the default values, and then choose **Allow**.


AWS IoT is requesting permission to use resources in your account

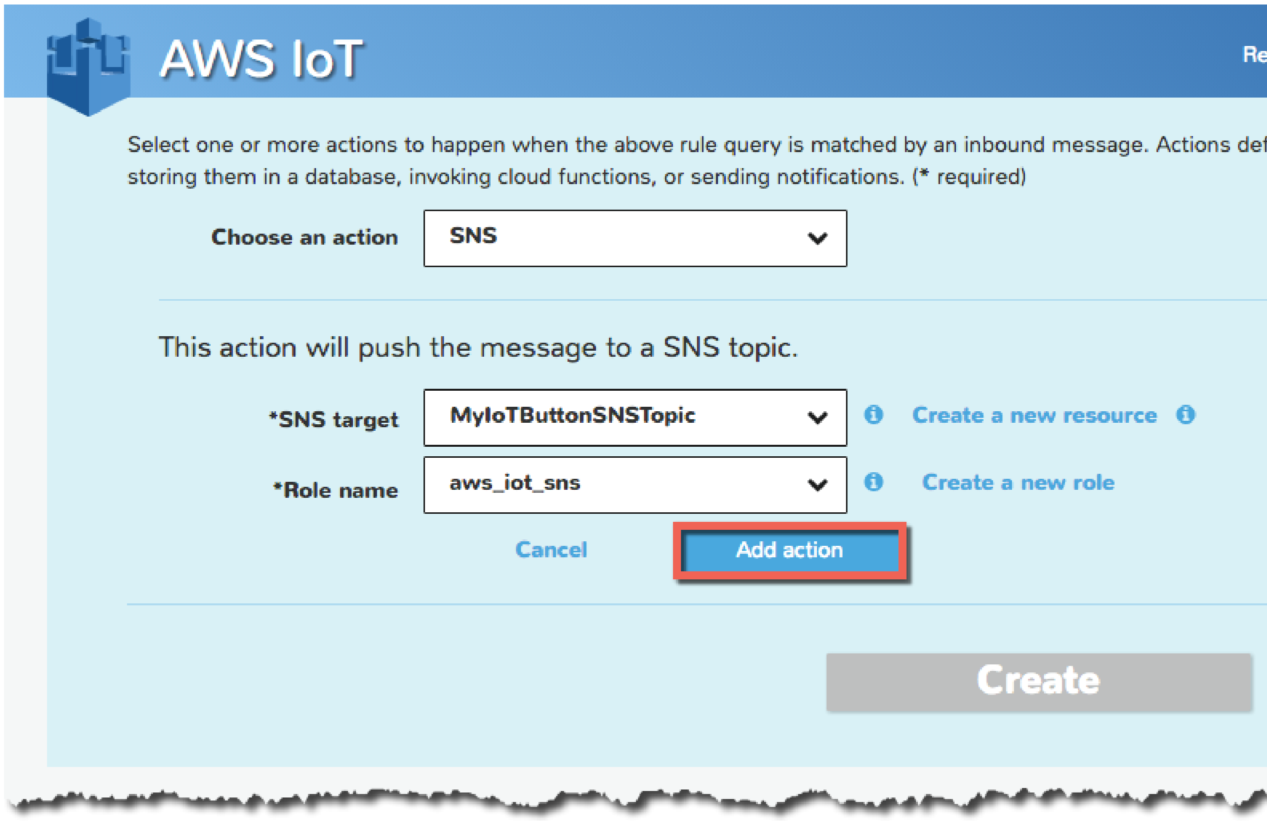
Click allow to give AWS IoT write access to resources in your account.

Provides write access to AWS Services and Resources

aws_iot_sns

Create a new Role Policy

11. Choose **Add action** to add the action to the rule.



Select one or more actions to happen when the above rule query is matched by an inbound message. Actions def storing them in a database, invoking cloud functions, or sending notifications. (* required)

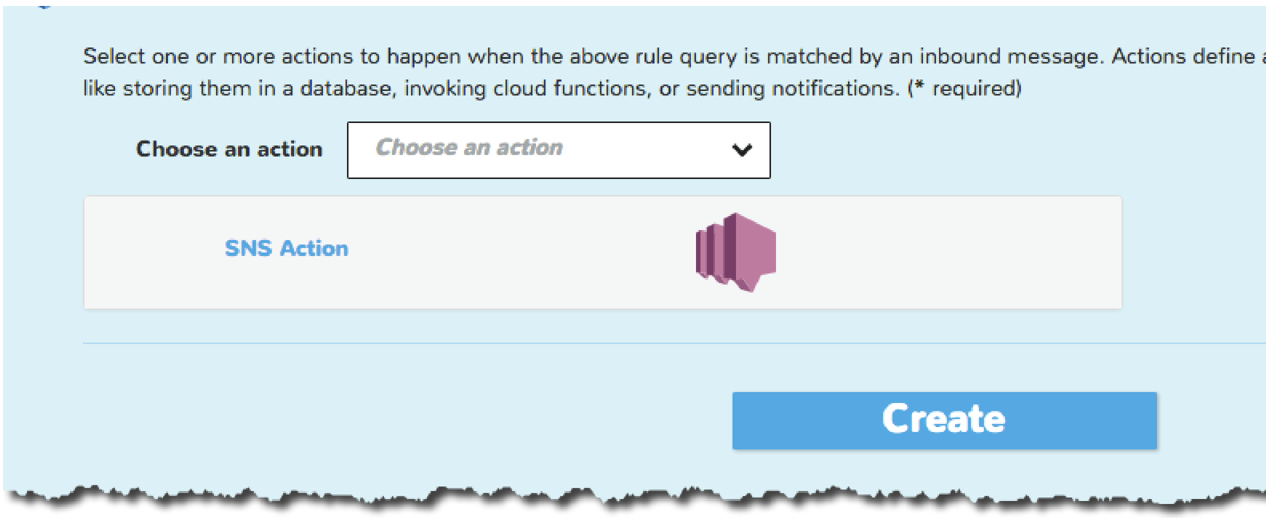
Choose an action: SNS

This action will push the message to a SNS topic.

*SNS target: MyIoTButtonSNSTopic

*Role name: aws_iot_sns

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



Select one or more actions to happen when the above rule query is matched by an inbound message. Actions define like storing them in a database, invoking cloud functions, or sending notifications. (* required)

Choose an action: Choose an action

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.

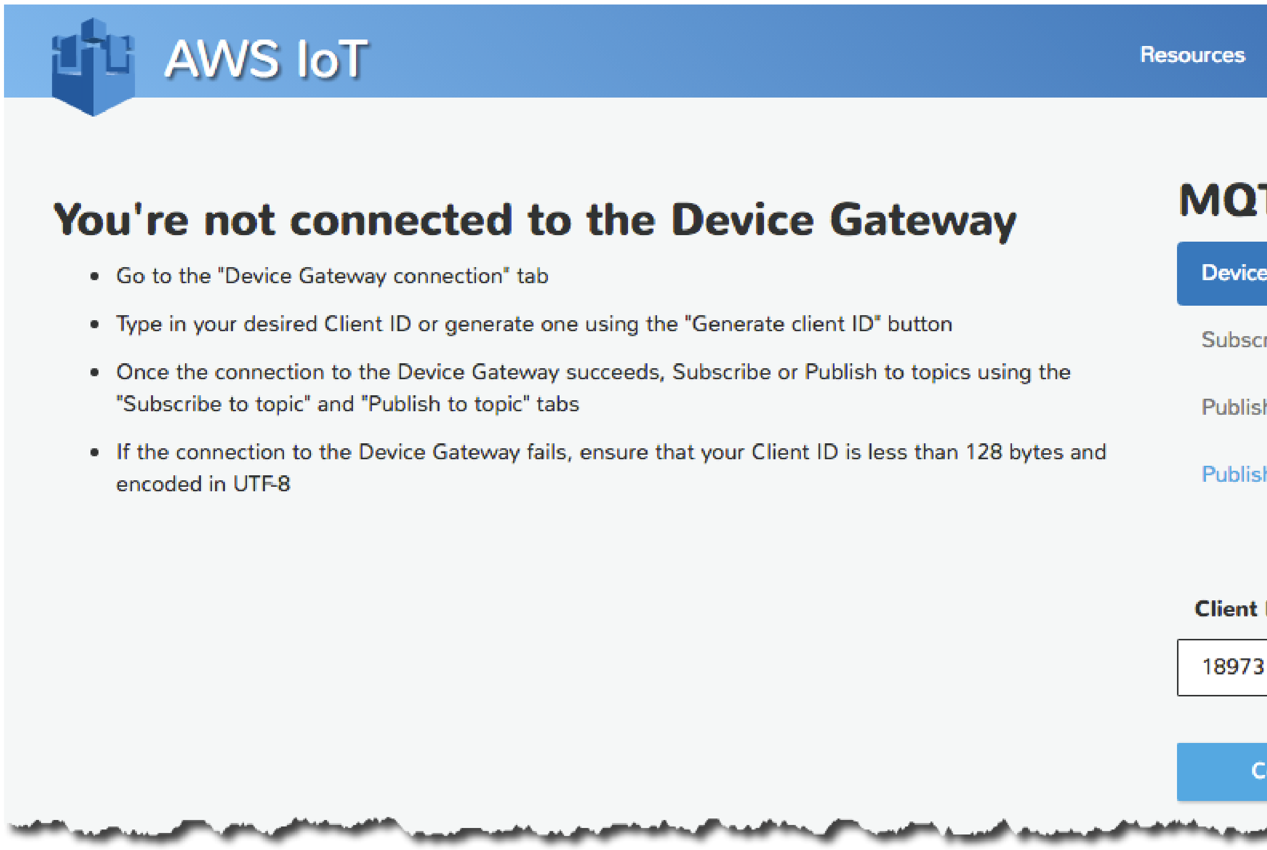
AWS IoT Button

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

AWS IoT MQTT Client

To test your rule with the AWS IoT MQTT client:

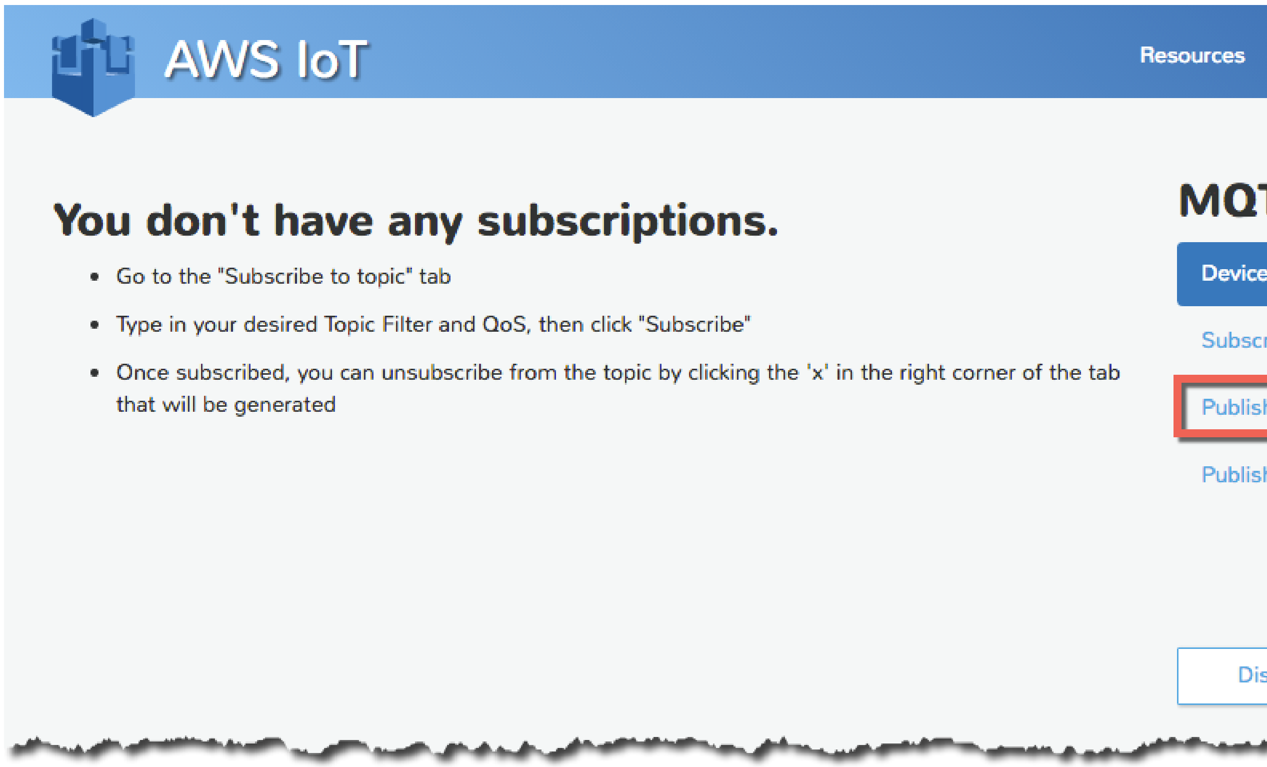
1. In the AWS IoT console, choose **MQTT Client**.
2. Choose **Generate client ID**, and then choose **Connect**.



You're not connected to the Device Gateway

- Go to the "Device Gateway connection" tab
- Type in your desired Client ID or generate one using the "Generate client ID" button
- Once the connection to the Device Gateway succeeds, Subscribe or Publish to topics using the "Subscribe to topic" and "Publish to topic" tabs
- If the connection to the Device Gateway fails, ensure that your Client ID is less than 128 bytes and encoded in UTF-8

3. On the MQTT client page, choose **Publish to topic**.

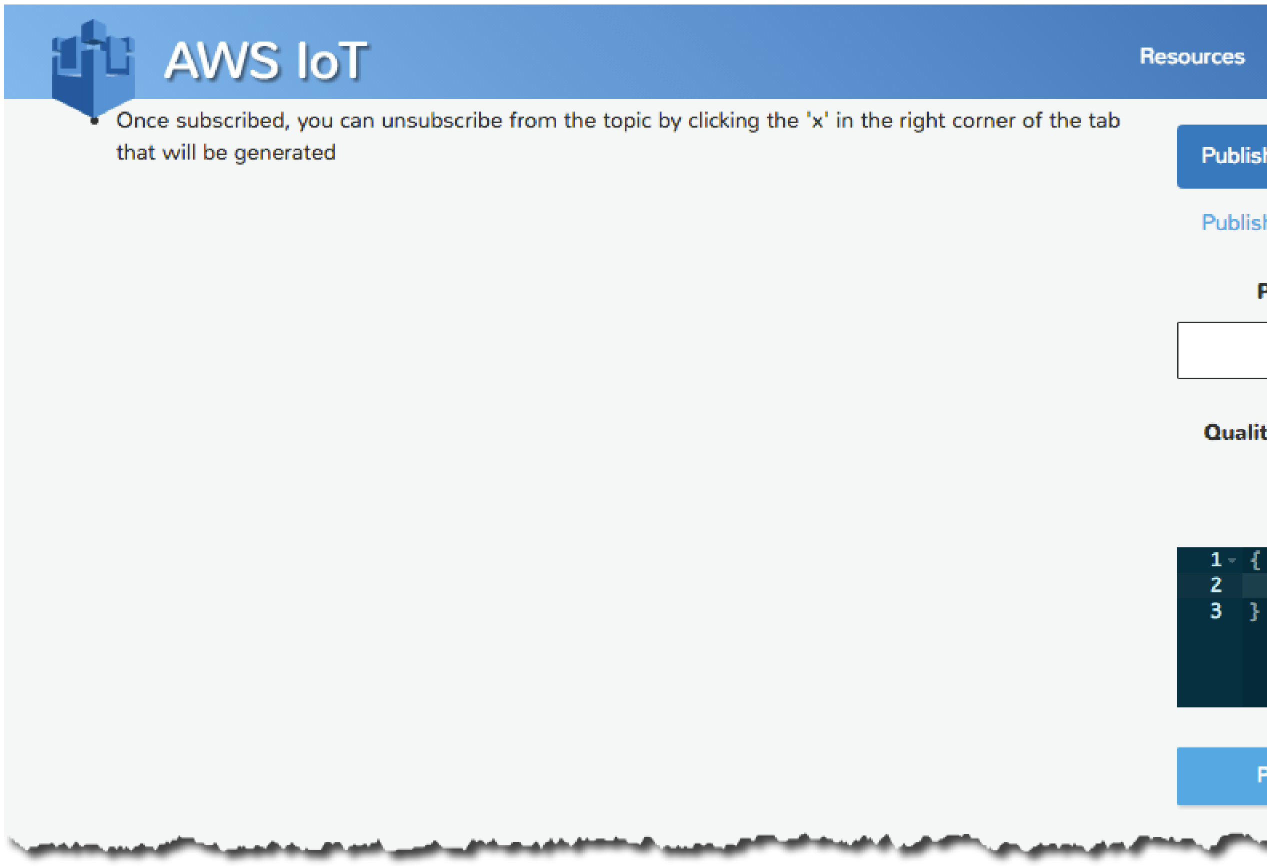


You don't have any subscriptions.

- Go to the "Subscribe to topic" tab
- Type in your desired Topic Filter and QoS, then click "Subscribe"
- Once subscribed, you can unsubscribe from the topic by clicking the 'x' in the right corner of the tab that will be generated

4. In the **Publish topic** field, type **`my/topic`**.
5. In **Payload**, type the following JSON:

```
{
    "message": "Hello, world from  AWS IoT!"
}
```



- Once subscribed, you can unsubscribe from the topic by clicking the 'x' in the right corner of the tab that will be generated

```
{
}
```

6. Choose **Publish**. You should receive an Amazon SNS message on your cell phone.

Next Steps

For more information about AWS IoT rules, see AWS IoT Rule Tutorials (p. 41) and AWS IoT Rules (p. 108).

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. 18), 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. 117).

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.

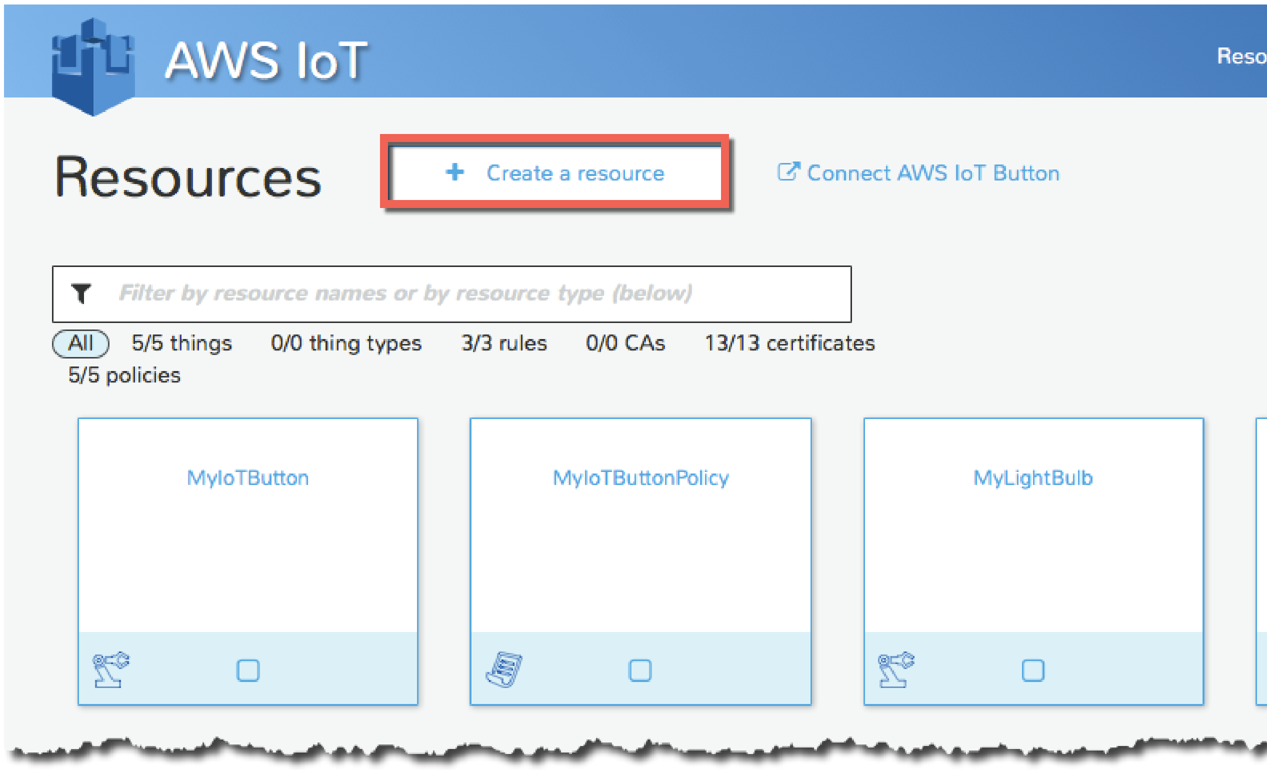
You can use your own device, but you will need to know on which MQTT topic your device publishes so you can specify it as the topic filter in the rule. For more information, see AWS IoT Rules (p. 108).

Creating a DynamoDB Rule

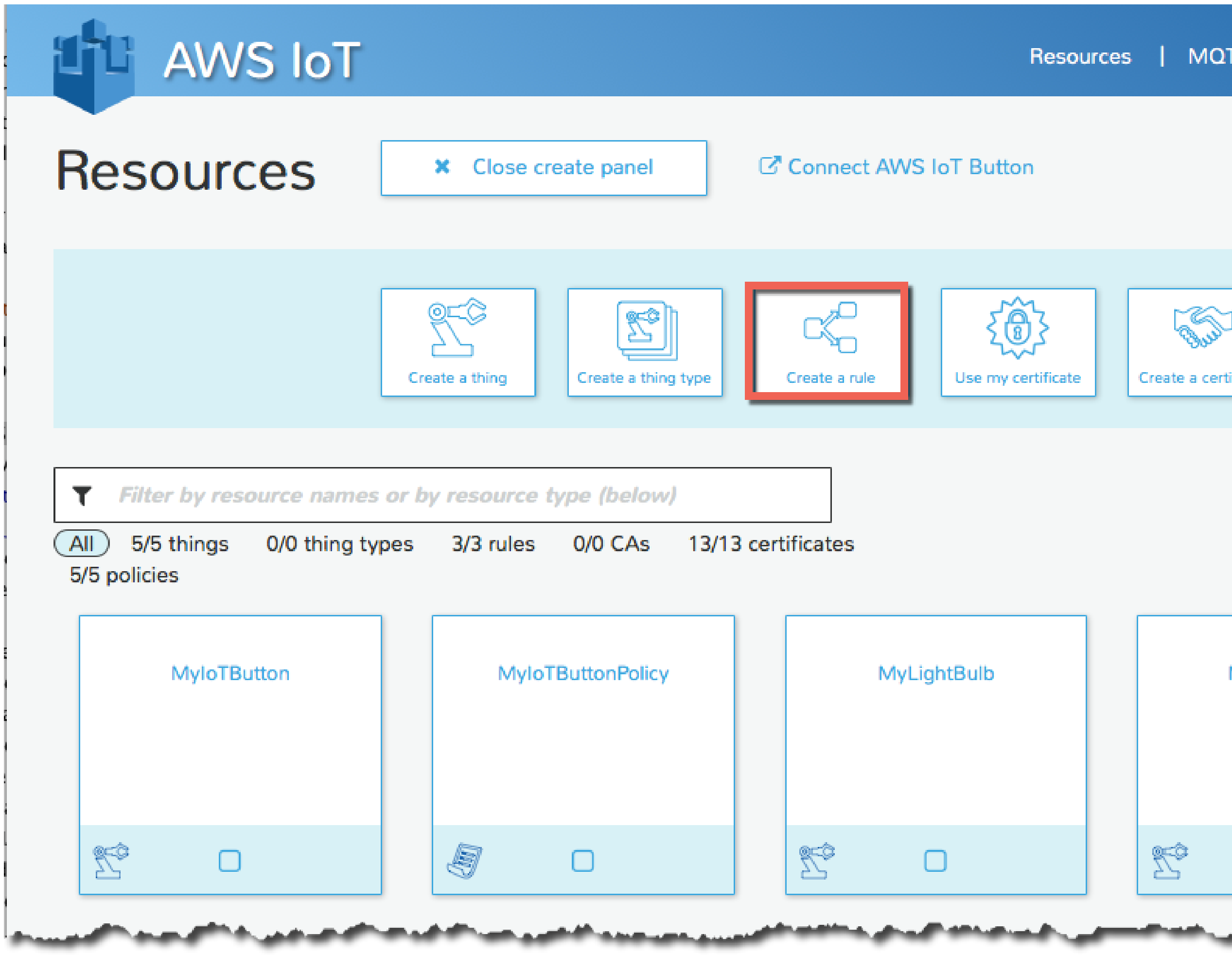
DynamoDB rules allow you to take information from an incoming MQTT message and write it to a DynamoDB table.

To create a DynamoDB rule:

1. In the AWS IoT console, choose **Create a resource**.



2. Choose **Create a rule**.



3. On the **Create a rule** page:

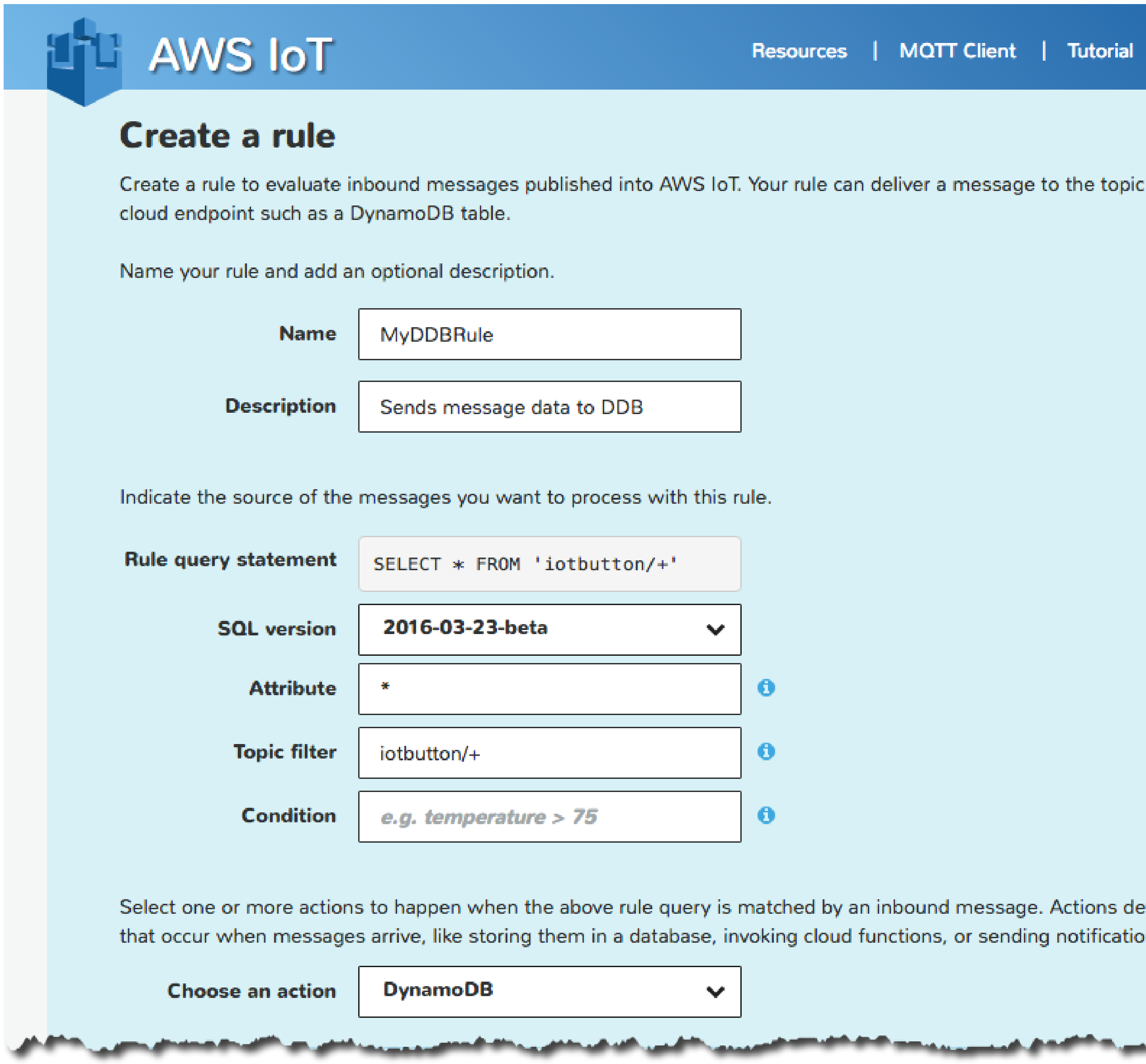
   Type a rule name and description in **Name** and **Description**.

   The **Rule query statement** field will be populated automatically when you enter data into the fields below it.

   In **Attribute**, type `*`. This determines which part of the incoming message will be sent to the rule action. Using `*` sends the entire message.

   In **Topic filter**, type **`iotbutton/+`**. If you are using a different device, type a topic filter that will match the MQTT topic on which your device publishes.

   From **Choose an action**, choose **Insert message into a database table (DynamoDB)**.



Create a rule

Create a rule to evaluate inbound messages published into AWS IoT. Your rule can deliver a message to the topic
cloud endpoint such as a DynamoDB table.

Name your rule and add an optional description.

Name	MyDDBRule
Description	Sends message data to DDB

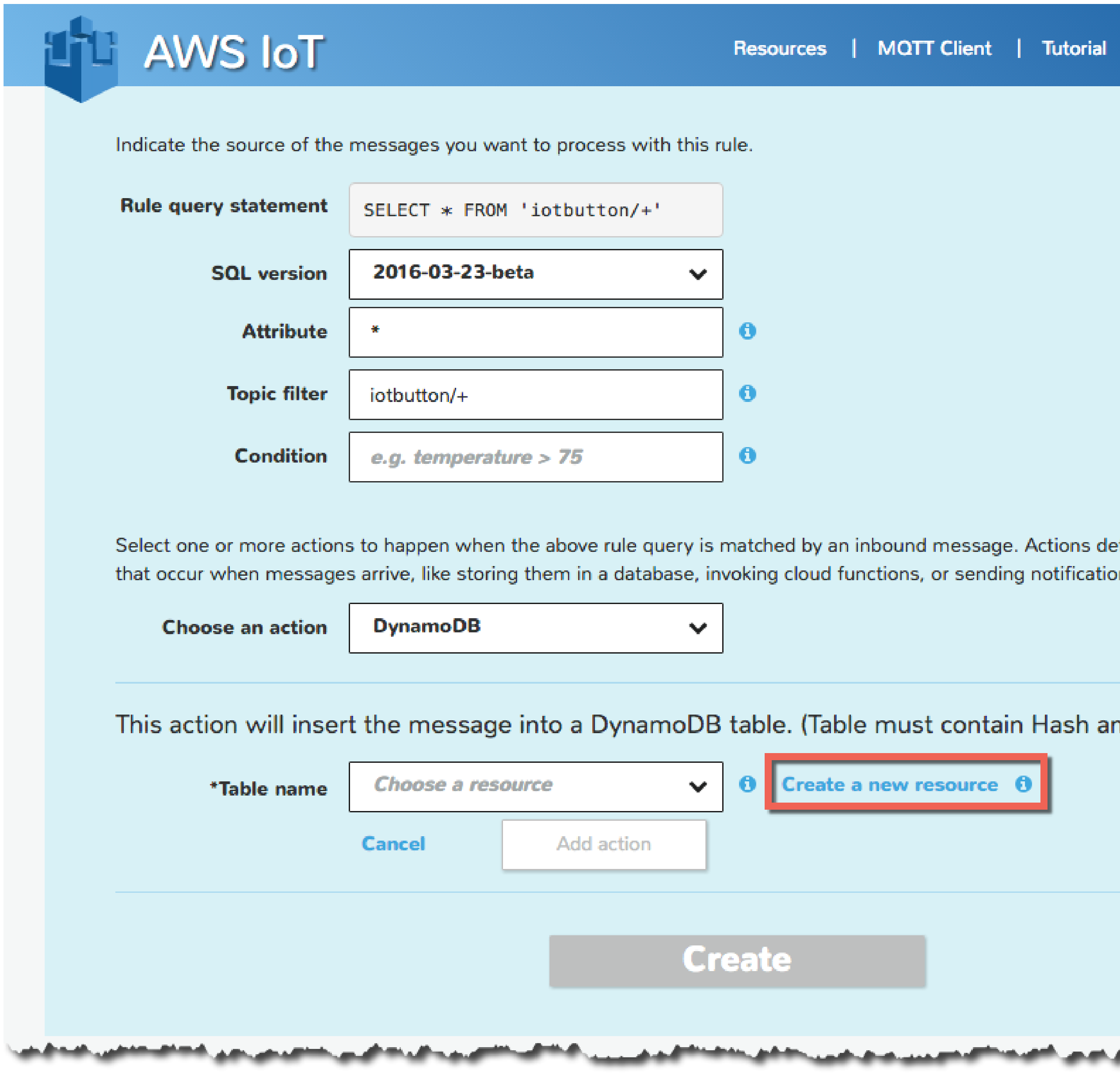
Indicate the source of the messages you want to process with this rule.

Rule query statement	`SELECT * FROM 'iotbutton/+'`
SQL version	2016-03-23-beta
Attribute	*
Topic filter	iotbutton/+
Condition	e.g. temperature > 75

Select one or more actions to happen when the above rule query is matched by an inbound message. Actions de
that occur when messages arrive, like storing them in a database, invoking cloud functions, or sending notificatio

Choose an action	DynamoDB

4. The **Create a rule** page will expand. Next to the **Table name** drop-down list, choose **Create a new resource**. This will open the Amazon DynamoDB console where you can create a DynamoDB table.



Indicate the source of the messages you want to process with this rule.

Rule query statement	SELECT * FROM 'iotbutton/+'
SQL version	2016-03-23-beta
Attribute	*
Topic filter	iotbutton/+
Condition	e.g. temperature > 75

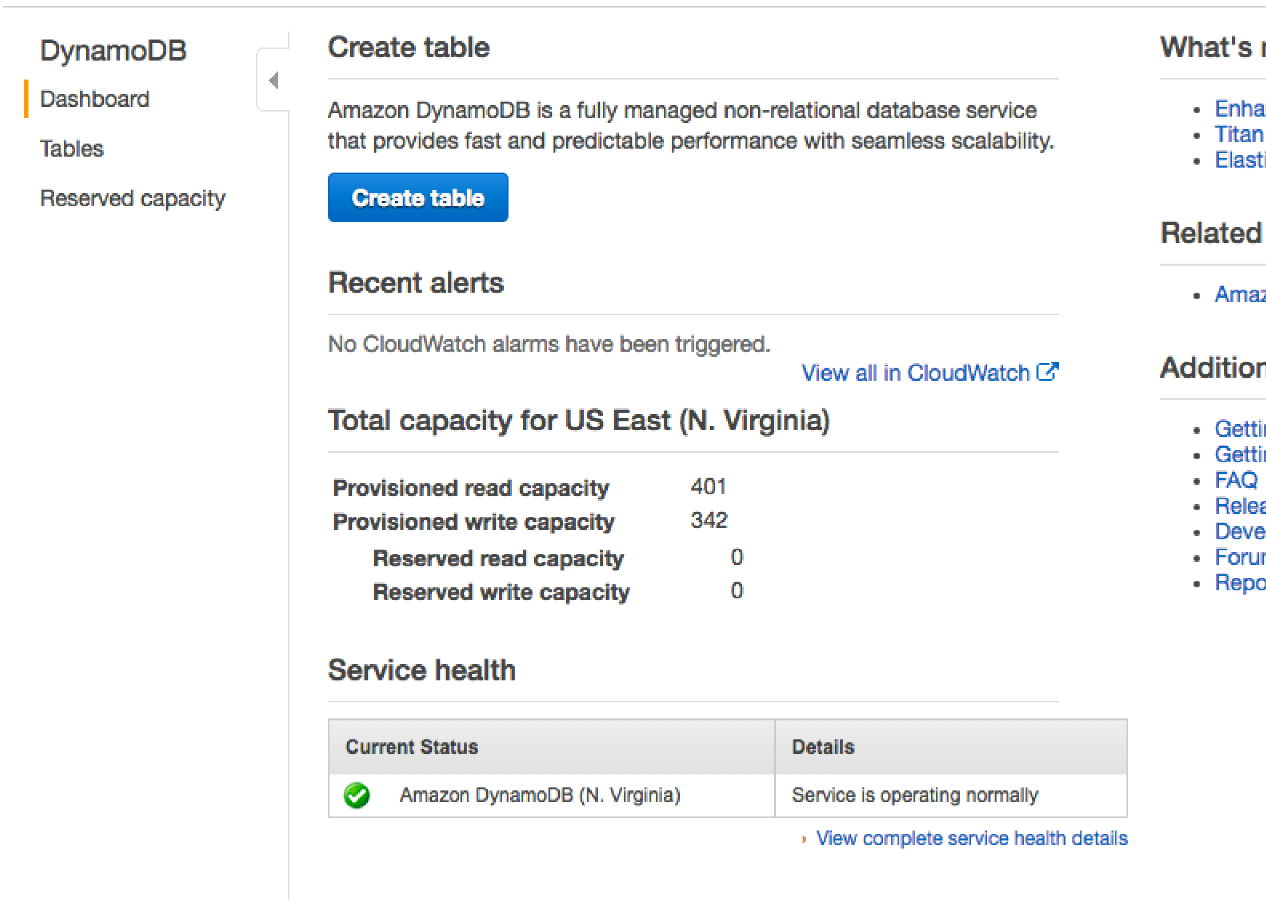
Select one or more actions to happen when the above rule query is matched by an inbound message. Actions de
that occur when messages arrive, like storing them in a database, invoking cloud functions, or sending notificatio

Choose an action	DynamoDB

This action will insert the message into a DynamoDB table. (Table must contain Hash ar

*Table name	Choose a resource

5. Choose **Create table**.



Amazon DynamoDB is a fully managed non-relational database service that provides fast and predictable performance with seamless scalability.

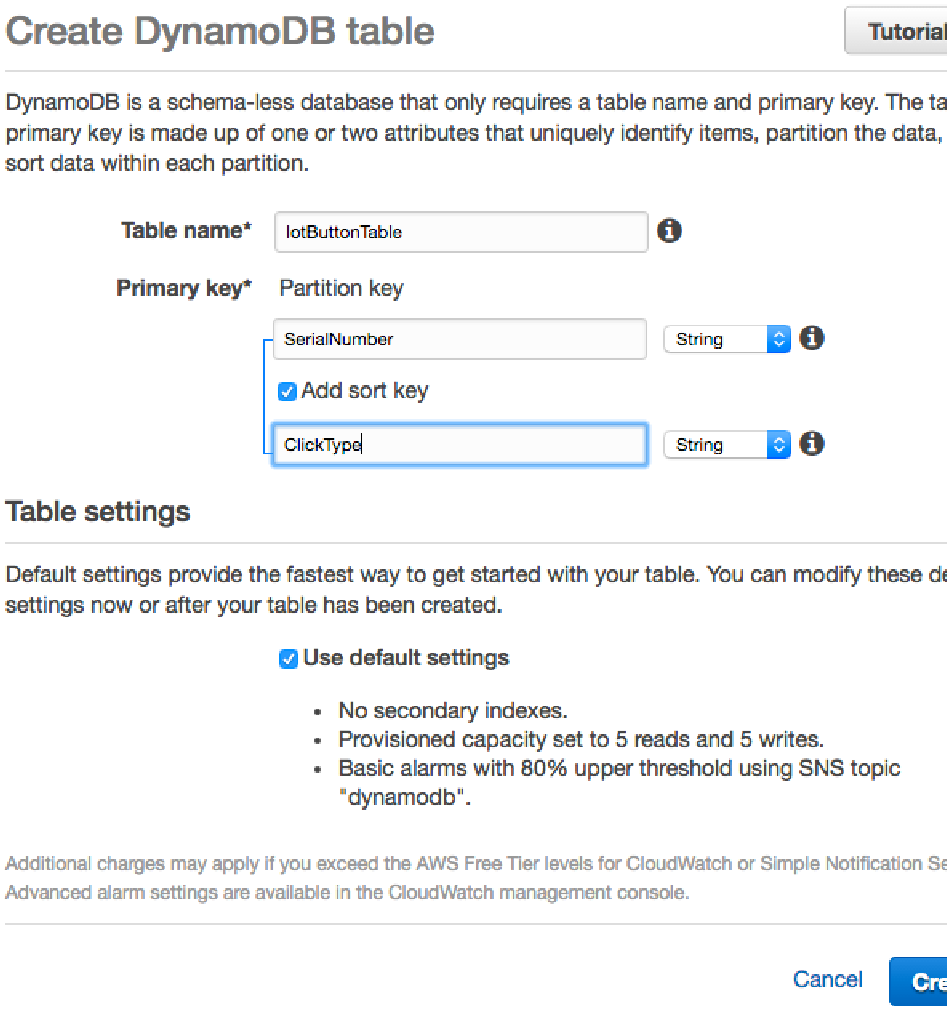
No CloudWatch alarms have been triggered.

Provisioned read capacity	401
Provisioned write capacity	342
Reserved read capacity	0
Reserved write capacity	0

Current Status	Details
✓ Amazon DynamoDB (N. Virginia)	Service is operating normally

6. In **Table name**, type a name for the table. The partition and sort keys are combined to create a primary key for your DynamoDB table. For the **Partition key**, type **`SerialNumber`**, and then select **Add sort key**. For the **Sort key**, type `ClickType`. Both the partition and sort keys should be of type **String**.

   Your screen should now look like the following:



Create DynamoDB table

DynamoDB is a schema-less database that only requires a table name and primary key. The ta
primary key is made up of one or two attributes that uniquely identify items, partition the data,
sort data within each partition.

Table name* IotButtonTable

Primary key* Partition key

SerialNumber String

✓ Add sort key

ClickType String

Table settings

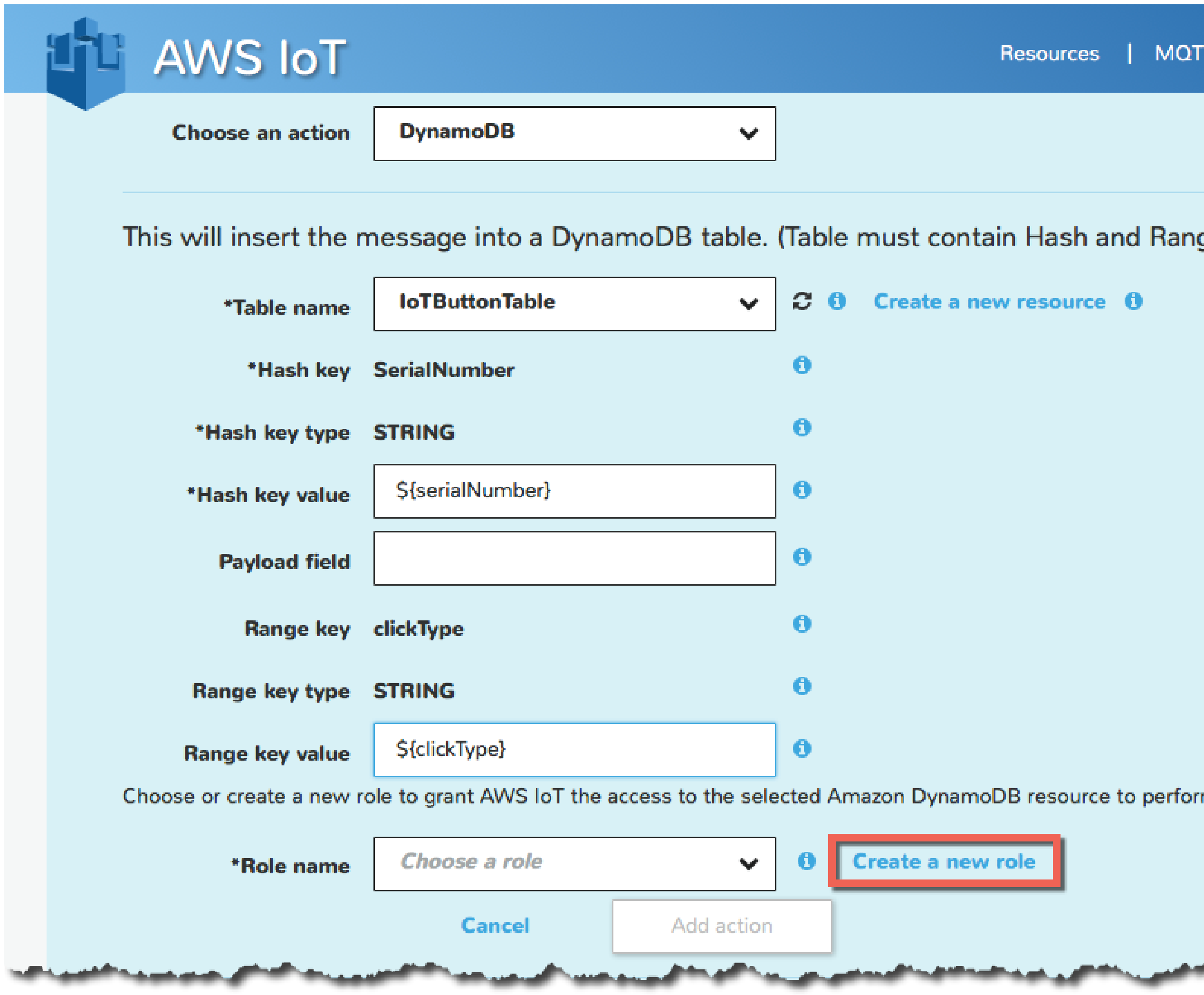
Default settings provide the fastest way to get started with your table. You can modify these de
settings now or after your table has been created.

✓ Use default settings

- No secondary indexes.
- Provisioned capacity set to 5 reads and 5 writes.
- Basic alarms with 80% upper threshold using SNS topic "dynamodb".

Additional charges may apply if you exceed the AWS Free Tier levels for CloudWatch or Simple Notification Se
Advanced alarm settings are available in the CloudWatch management console.

7. Choose **Create**. It will take a few seconds to create your DynamoDB table. Close the browser tab that contains the Amazon DynamoDB console. If you do not close the tab, your DynamoDB table will not be displayed in the **Table name** drop-down list in the AWS IoT console. In the AWS IoT console, choose your new table.
8. 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 **Payload field** blank. By default, the entire message will be written to a column in the table called Payload. Select **Create a new role**.



Choose an action	DynamoDB

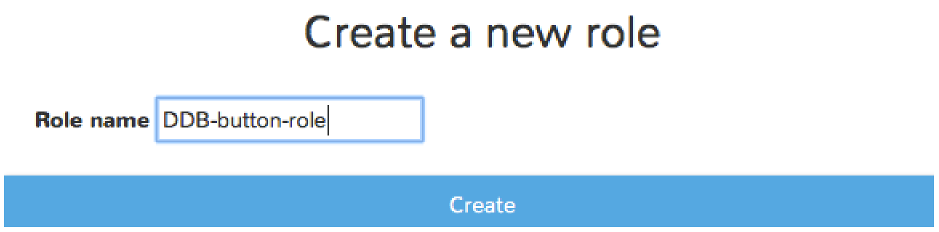
This will insert the message into a DynamoDB table. (Table must contain Hash and Rang

*Table name	IoTButtonTable
*Hash key	SerialNumber
*Hash key type	STRING
*Hash key value	${serialNumber}
Payload field	
Range key	clickType
Range key type	STRING
Range key value	${clickType}

Choose or create a new role to grant AWS IoT the access to the selected Amazon DynamoDB resource to perfor

*Role name	Choose a role

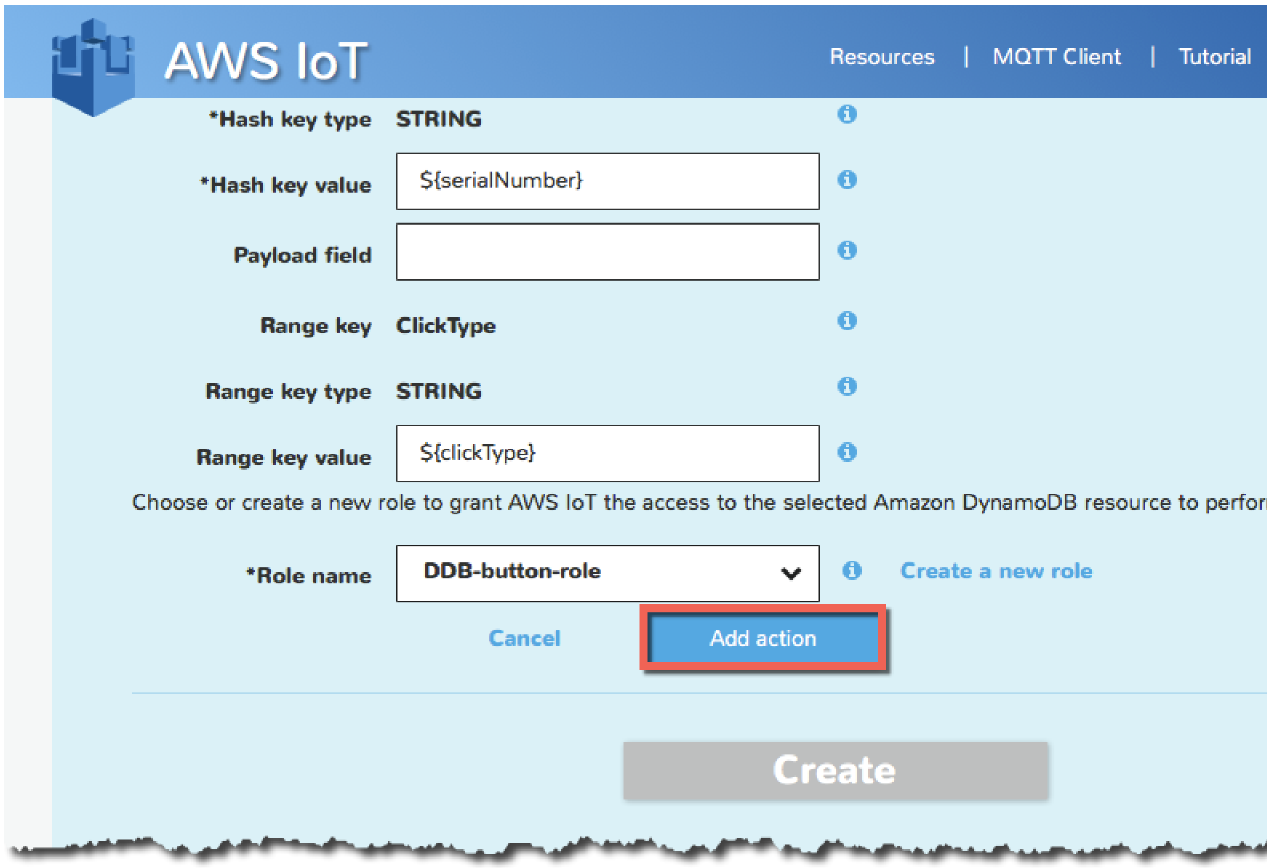
9. Type a unique role name in the **Create a new role** dialog box, and then choose the **Create** button.



Create a new role

Role name	DDB-button-role

10. Choose **Add action** to add the action to the rule.



*Hash key type STRING

*Hash key value ${serialNumber}

Payload field

Range key ClickType

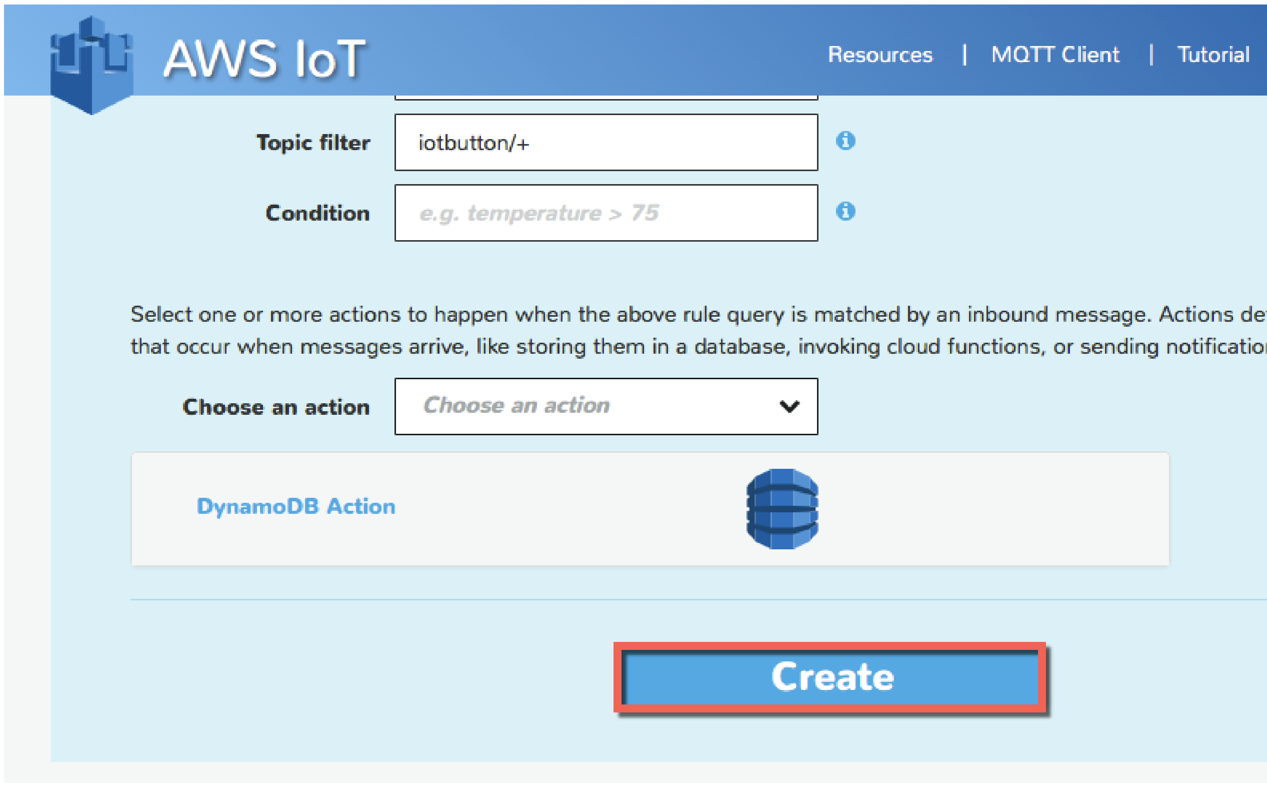
Range key type STRING

Range key value ${clickType}

Choose or create a new role to grant AWS IoT the access to the selected Amazon DynamoDB resource to perfor

*Role name DDB-button-role

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



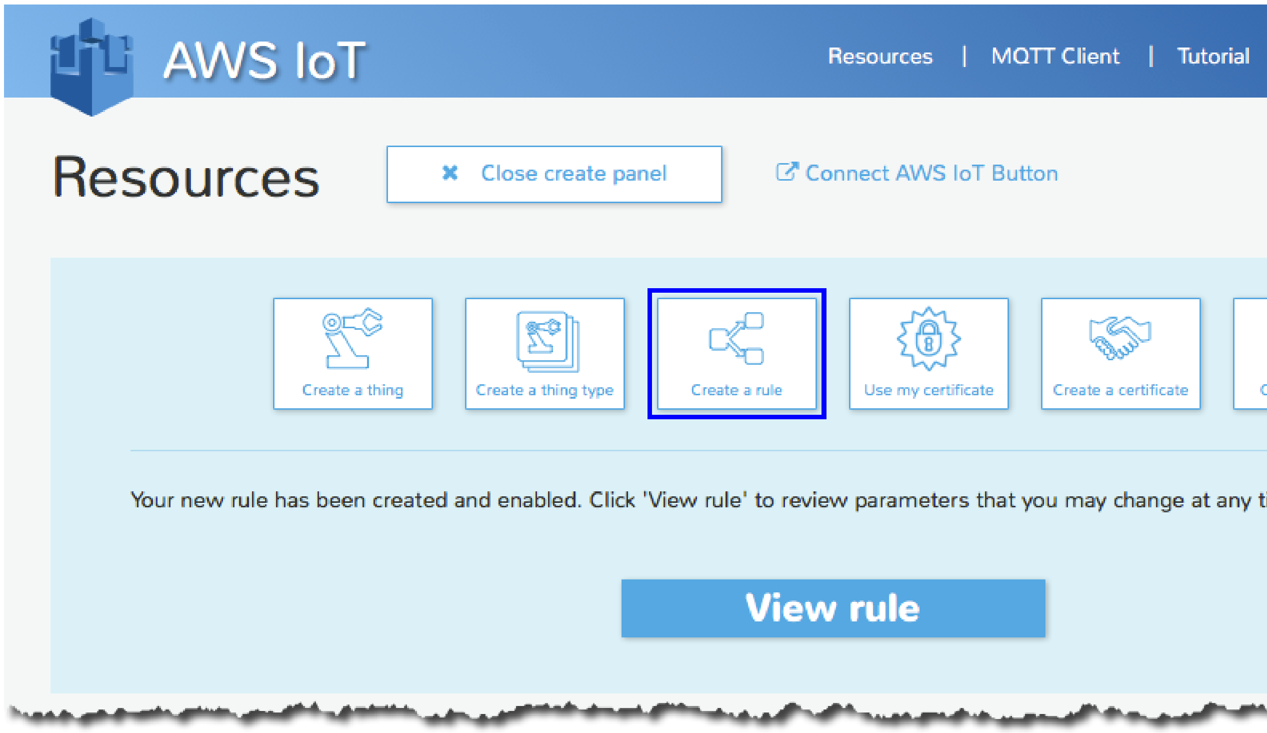
Topic filter iotbutton/+

Condition *e.g. temperature > 75*

Select one or more actions to happen when the above rule query is matched by an inbound message. Actions de
that occur when messages arrive, like storing them in a database, invoking cloud functions, or sending notificatio

Choose an action *Choose an action*

12. A confirmation message shows the rule has been created.



Your new rule has been created and enabled. Click 'View rule' to review parameters that you may change at any t

13. Test the rule by either 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.

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.

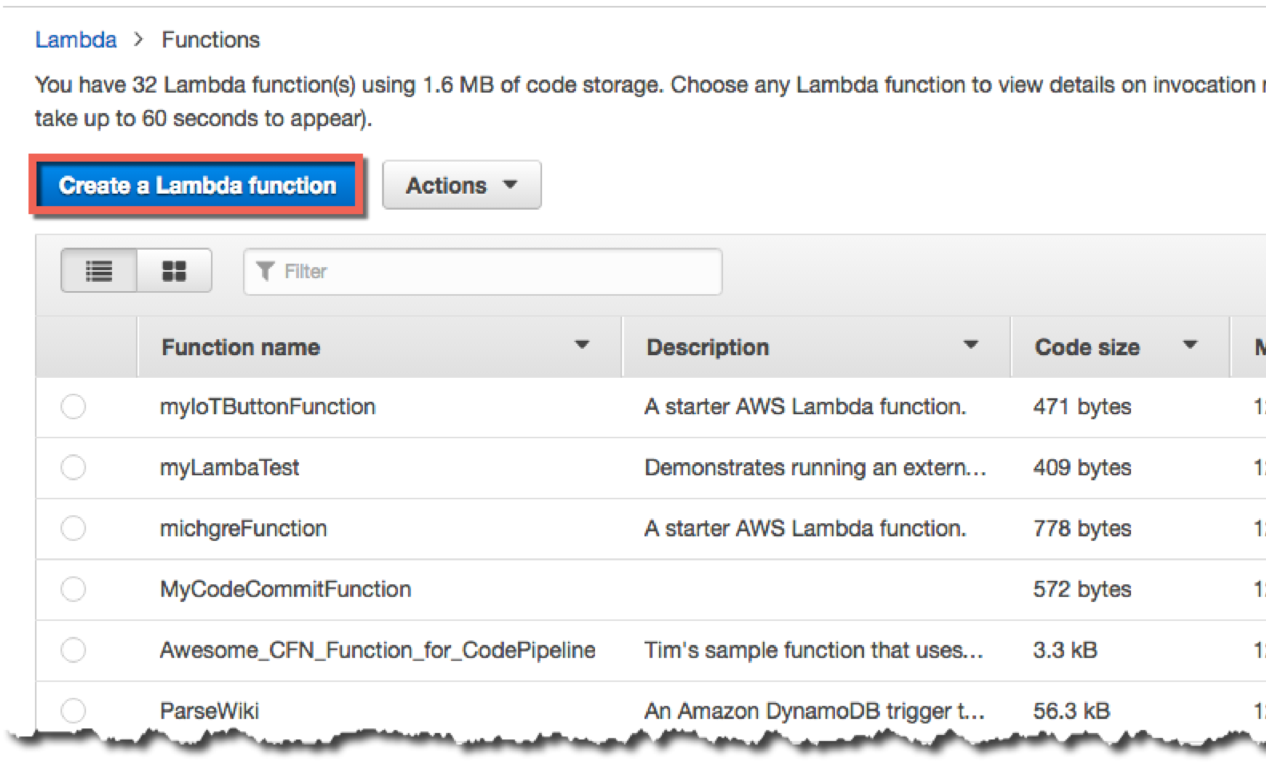
In this tutorial, we assume you have completed the AWS IoT Getting Started Tutorial (p. 18) in which you create and subscribe to an Amazon SNS topic using your cell phone number. You will create a Lambda function that publishes a message to the Amazon SNS topic you created in the AWS IoT Getting Started Tutorial (p. 18). 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 purchase 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:

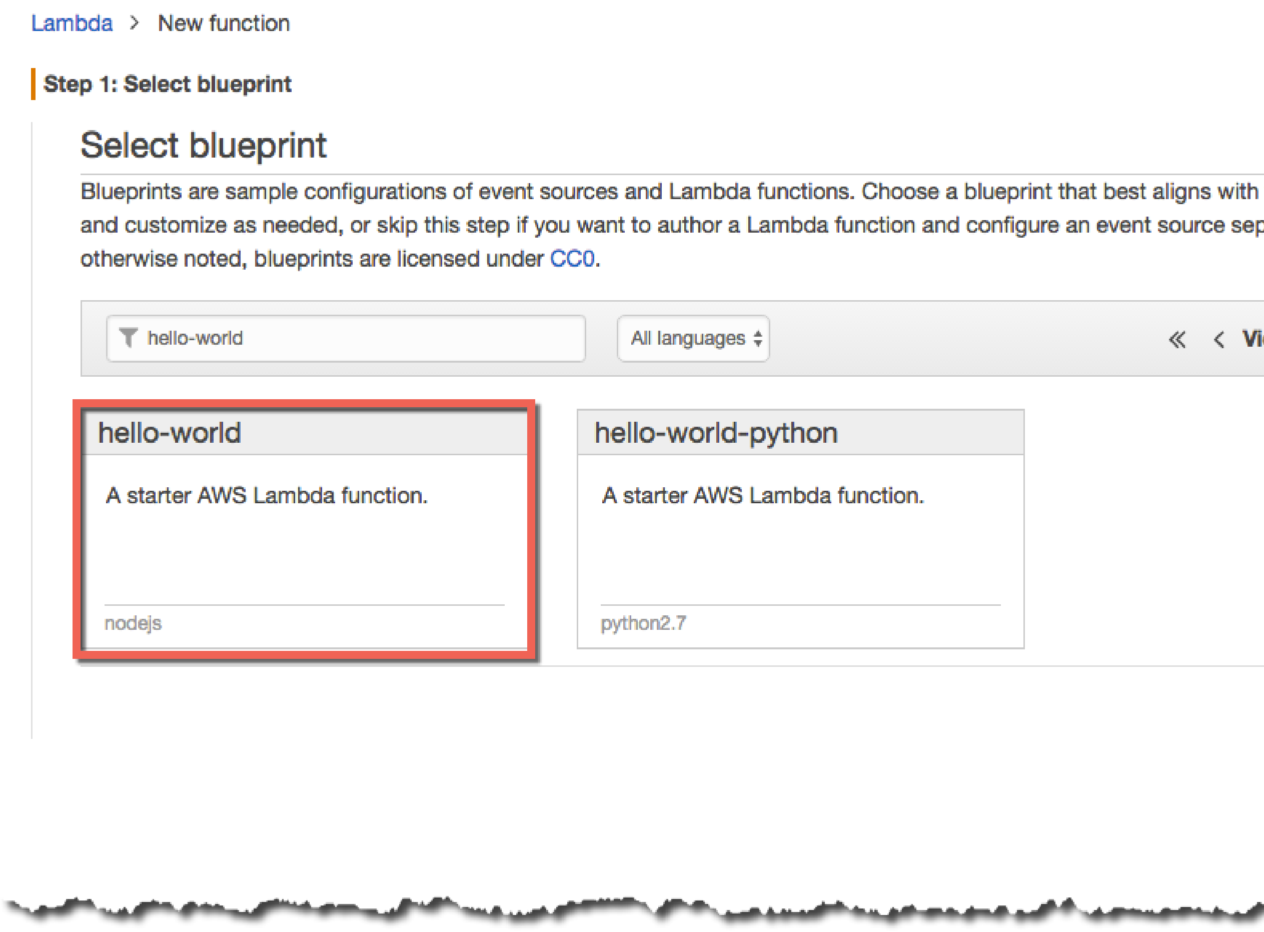
1. In the AWS Lambda console, choose **Create a Lambda function**.



You have 32 Lambda function(s) using 1.6 MB of code storage. Choose any Lambda function to view details on invocation
take up to 60 seconds to appear).

	Function name	Description	Code size
○	myIoTButtonFunction	A starter AWS Lambda function.	471 bytes
○	myLambaTest	Demonstrates running an extern...	409 bytes
○	michgreFunction	A starter AWS Lambda function.	778 bytes
○	MyCodeCommitFunction		572 bytes
○	Awesome_CFN_Function_for_CodePipeline	Tim's sample function that uses...	3.3 kB
○	ParseWiki	An Amazon DynamoDB trigger t...	56.3 kB

2. For the filter, type **`hello-world`**, and then choose the **hello-world** blueprint.



Lambda > New function

Step 1: Select blueprint

Select blueprint

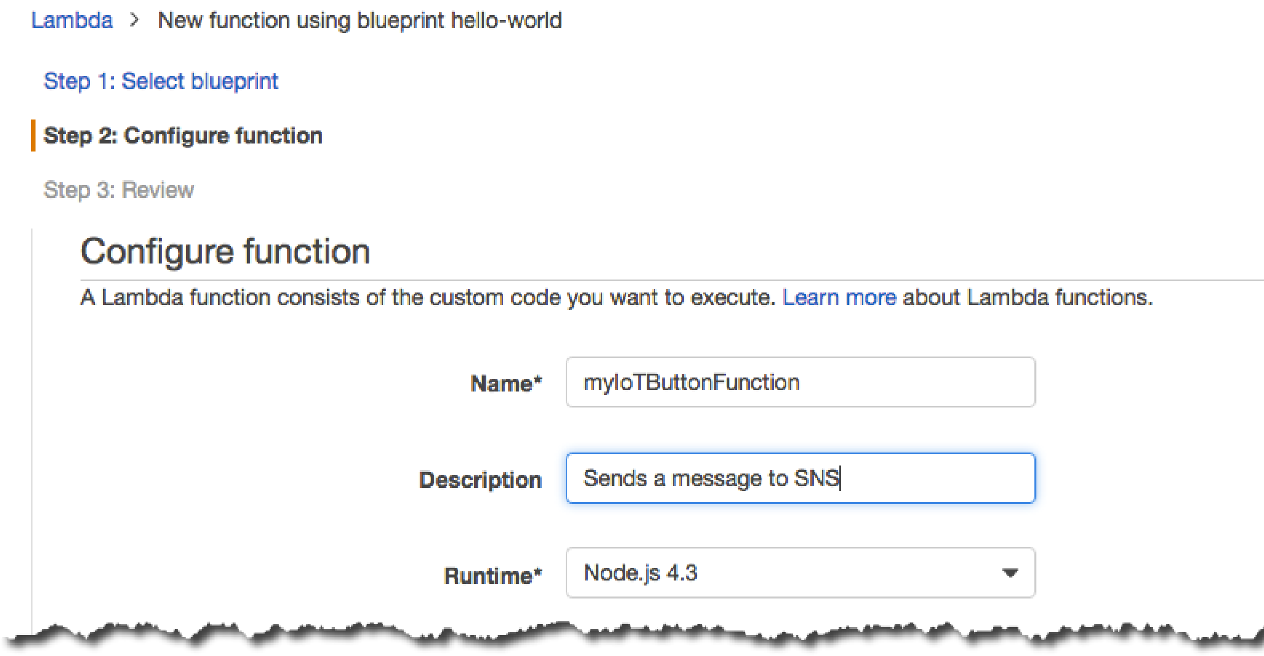
Blueprints are sample configurations of event sources and Lambda functions. Choose a blueprint that best aligns with
and customize as needed, or skip this step if you want to author a Lambda function and configure an event source sep
otherwise noted, blueprints are licensed under CC0.

hello-world | All languages

hello-world
A starter AWS Lambda function.
nodejs

hello-world-python
A starter AWS Lambda function.
python2.7

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



Lambda > New function using blueprint hello-world

Step 1: Select blueprint

Step 2: Configure function

Step 3: Review

Configure function

A Lambda function consists of the custom code you want to execute. Learn more about Lambda functions.

Name*	myIoTButtonFunction
Description	Sends a message to SNS
Runtime*	Node.js 4.3

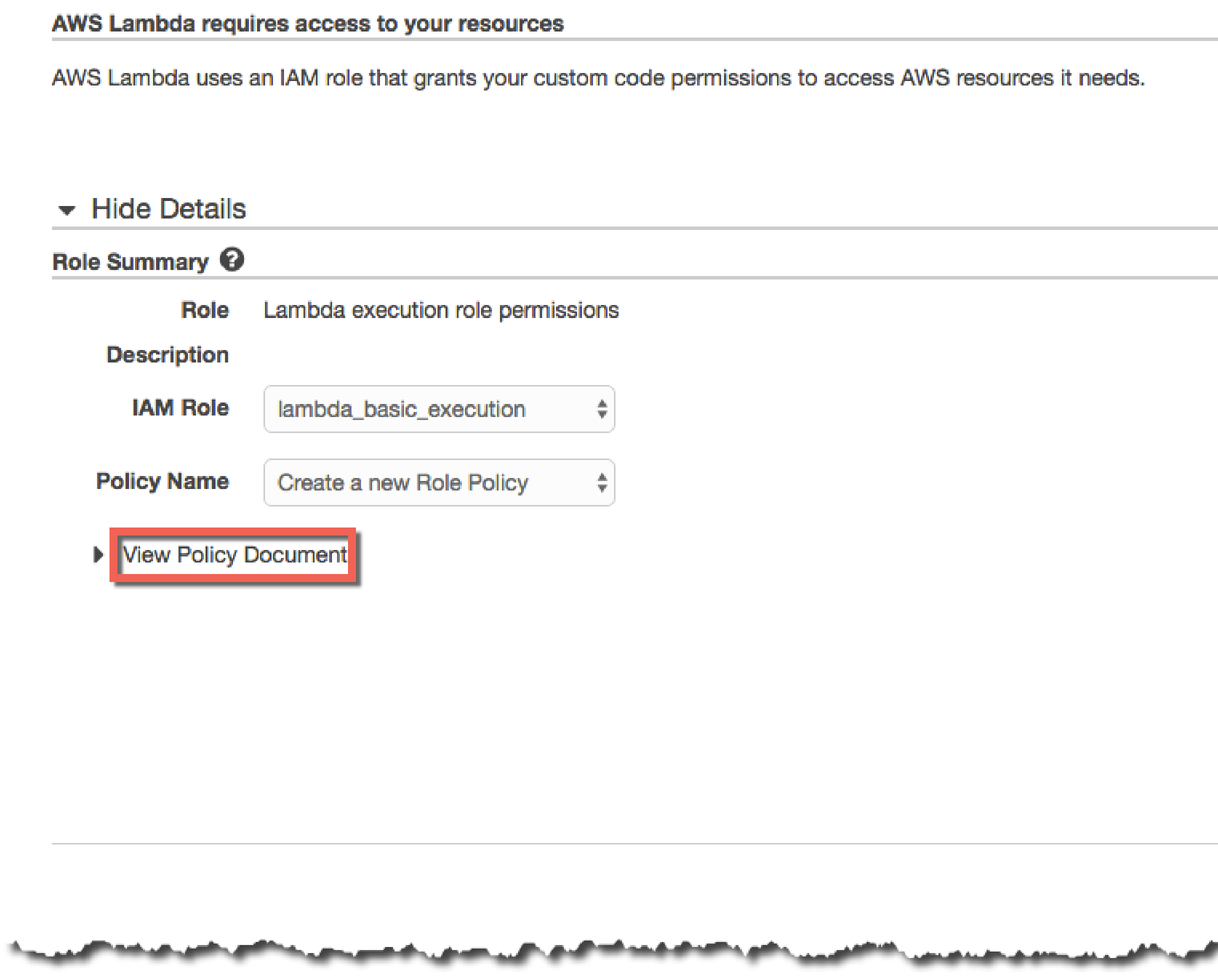
4. Scroll down to the **Lambda function code** section of the page. Replace the existing code with the following:

```
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 Mon
itoring 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);

        // Crewate 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);
    };
```

5. Scroll down to the **Lambda function handler and role** section of the page. For **Role**, choose **Basic execution 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:

   1. Choose **View Policy Document**.


AWS Lambda requires access to your resources

AWS Lambda uses an IAM role that grants your custom code permissions to access AWS resources it needs.

Hide Details

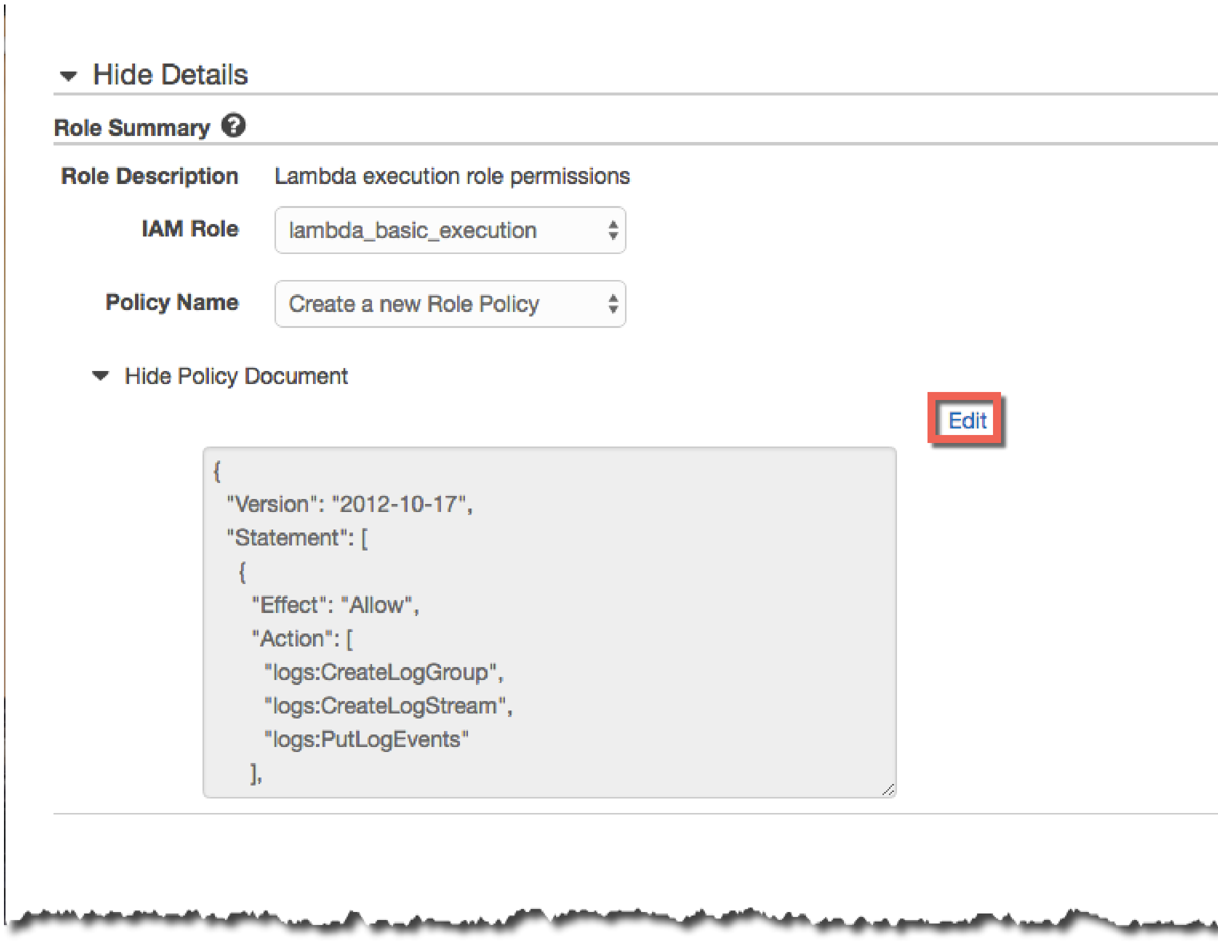
Role Summary

Role Description: Lambda execution role permissions

IAM Role: lambda_basic_execution

Policy Name: Create a new Role Policy

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



Role Description: Lambda execution role permissions
IAM Role: lambda_basic_execution
Policy Name: Create a new Role Policy

```
{
  "Version": "2012-10-17",
  "Statement": [
   {
    "Effect": "Allow",
    "Action": [
      "logs:CreateLogGroup",
      "logs:CreateLogStream",
      "logs:PutLogEvents"
    ],
```

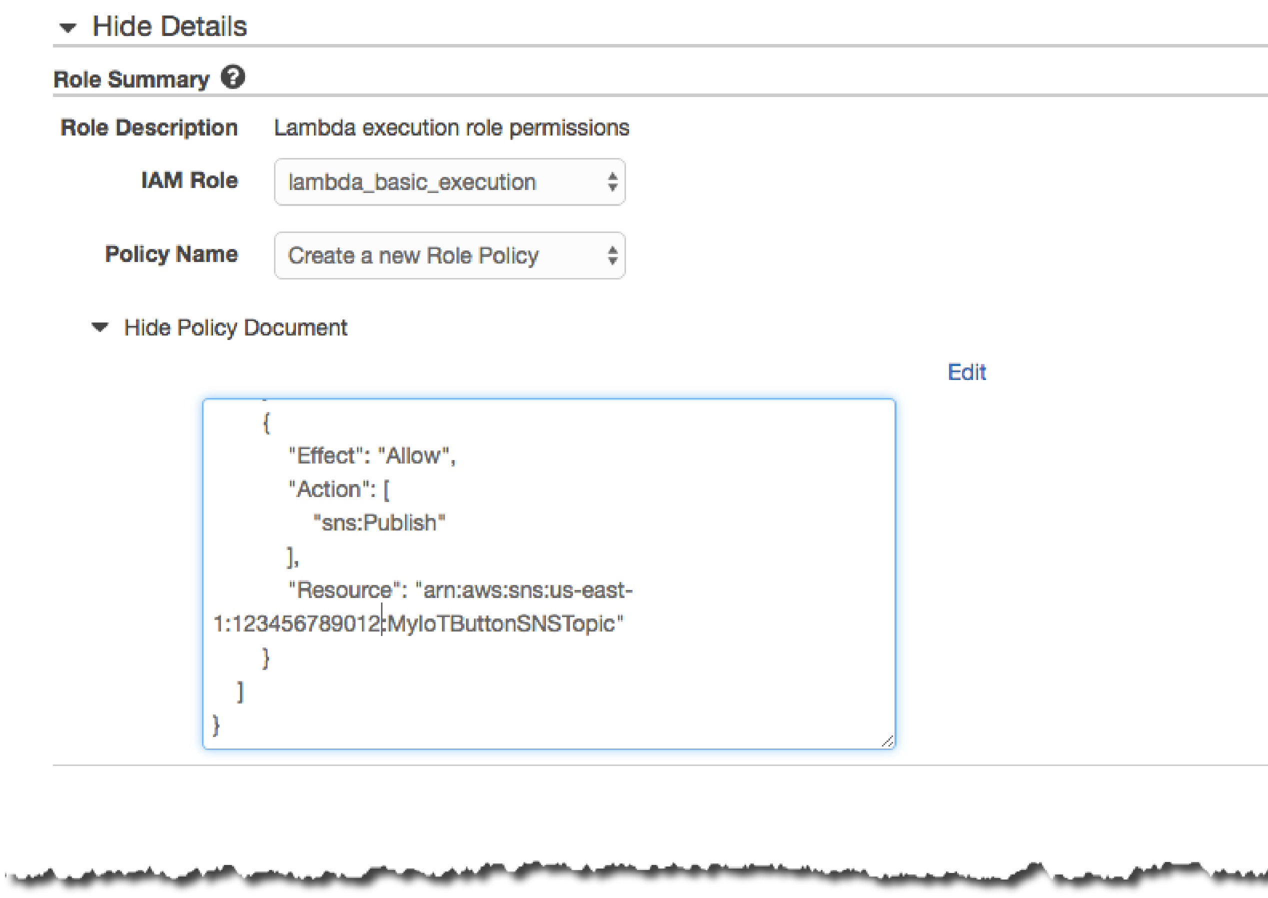
2. Replace the policy document with the following:

```
{
    "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:MyIoTButton
SNSTopic"
        }
    ]
}
```

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

Note
This example uses a fictitious AWS account number in the resource ARN. Make sure to use the ARN for your Amazon SNS topic.

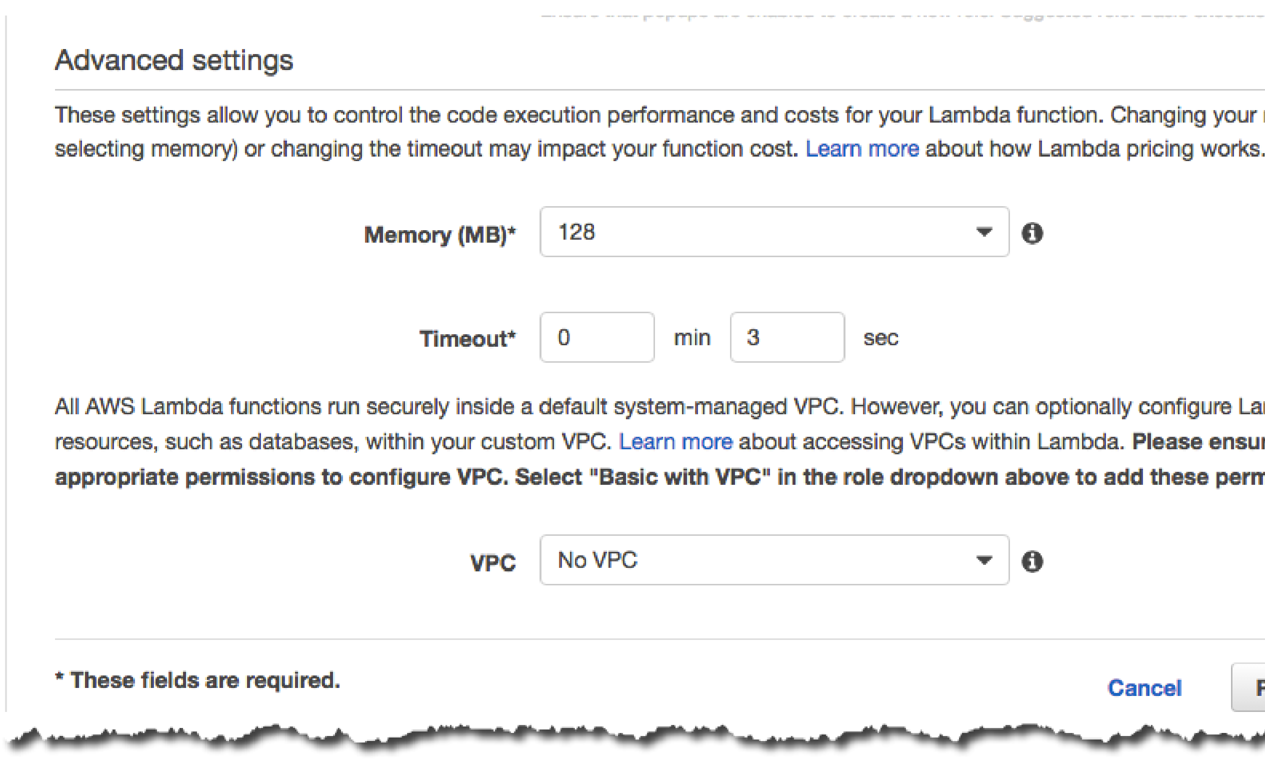
6. Choose **Allow**.



Role Description: Lambda execution role permissions
IAM Role: lambda_basic_execution
Policy Name: Create a new Role Policy

```
        {
            "Effect": "Allow",
            "Action": [
                "sns:Publish"
            ],
            "Resource": "arn:aws:sns:us-east-
1:123456789012:MyIoTButtonSNSTopic"
        }
    ]
}
```

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



Advanced settings

These settings allow you to control the code execution performance and costs for your Lambda function. Changing your
selecting memory) or changing the timeout may impact your function cost. Learn more about how Lambda pricing works.

Memory (MB)* 128

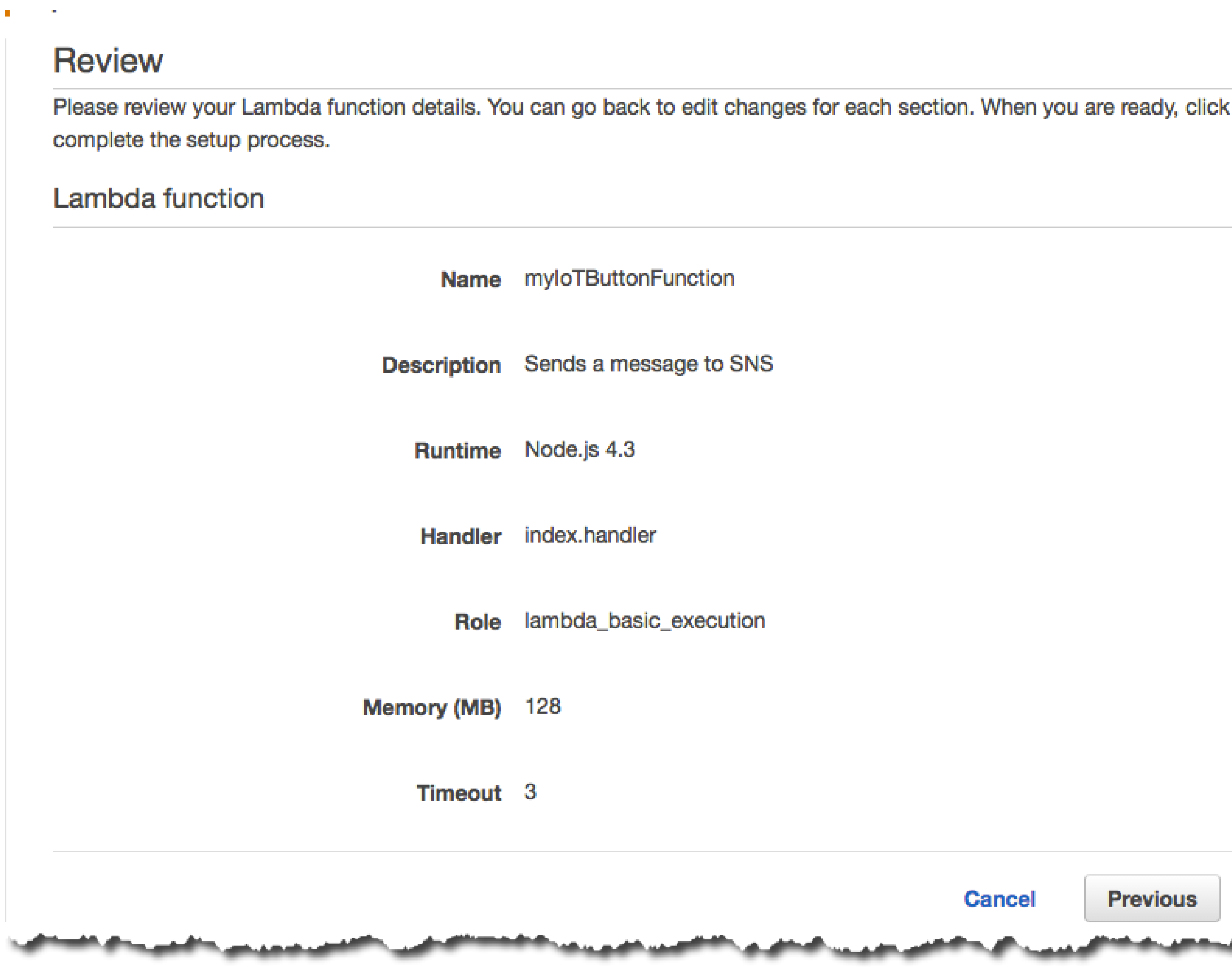
Timeout* 0 min 3 sec

All AWS Lambda functions run securely inside a default system-managed VPC. However, you can optionally configure La
resources, such as databases, within your custom VPC. Learn more about accessing VPCs within Lambda. **Please ensu
appropriate permissions to configure VPC. Select "Basic with VPC" in the role dropdown above to add these perm**

VPC No VPC

*** These fields are required.**

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


Review

Please review your Lambda function details. You can go back to edit changes for each section. When you are ready, click
complete the setup process.

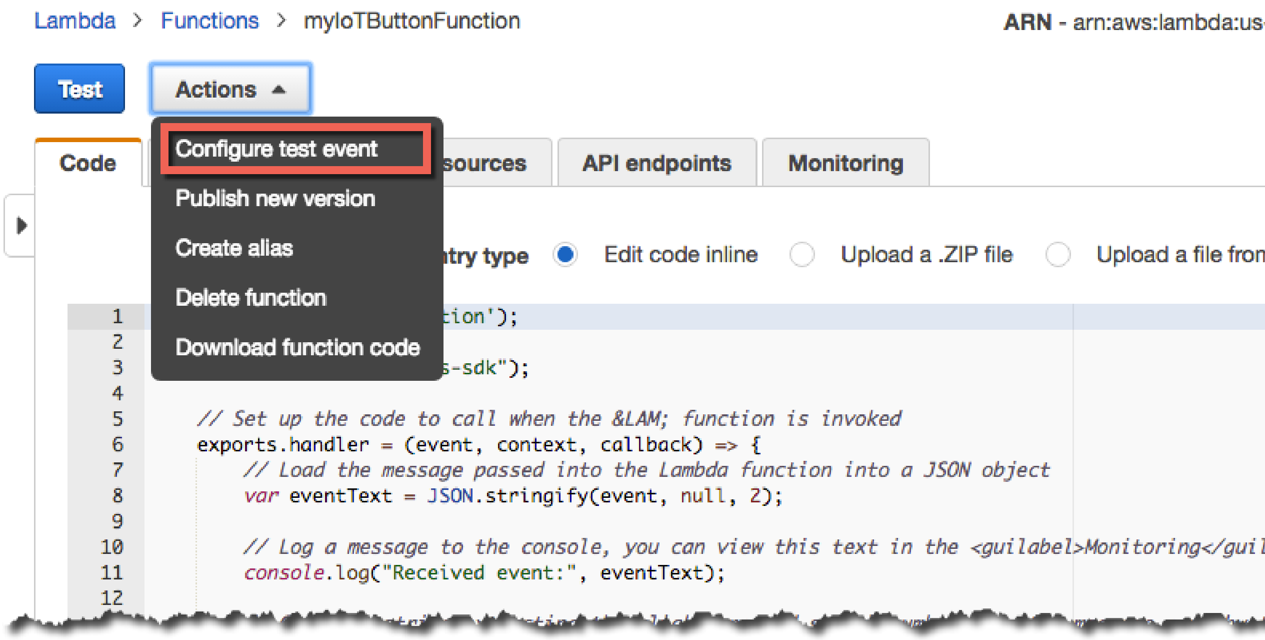
Lambda function

Name	myIoTButtonFunction
Description	Sends a message to SNS
Runtime	Node.js 4.3
Handler	index.handler
Role	lambda_basic_execution
Memory (MB)	128
Timeout	3

Test Your Lambda Function

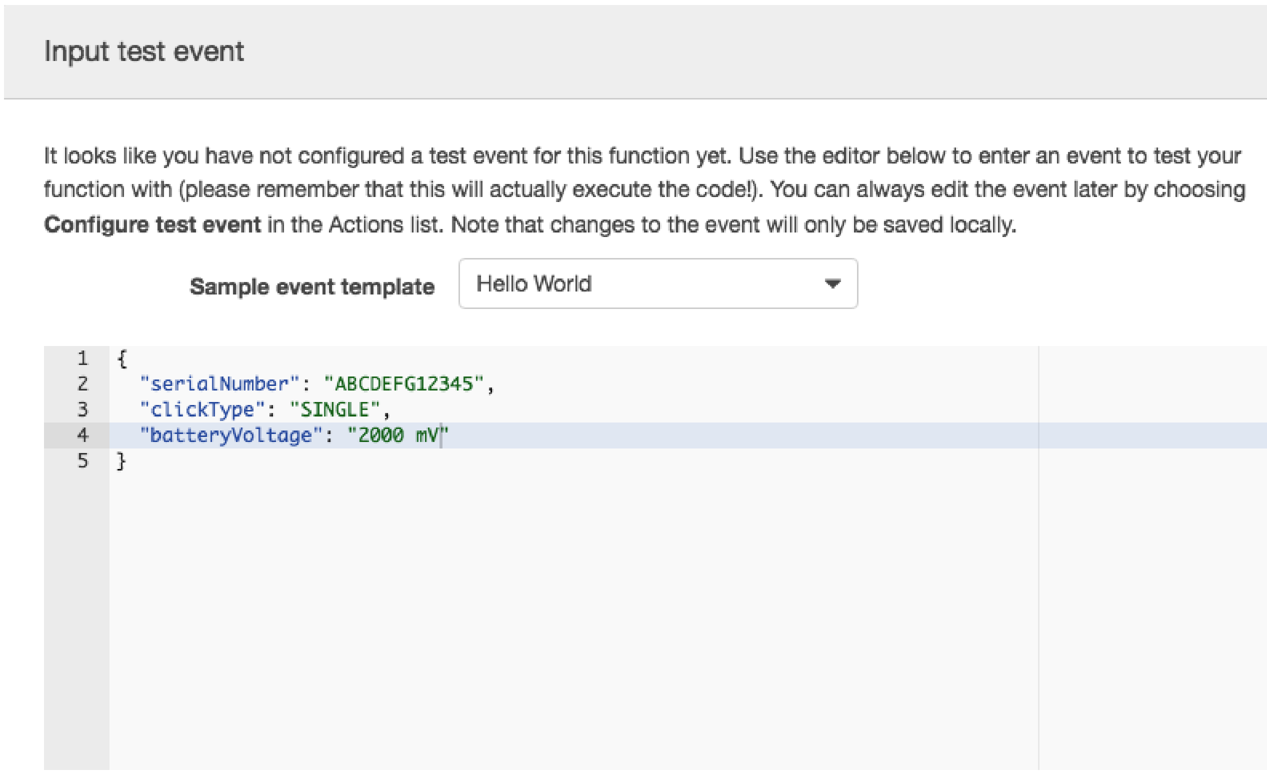
To test the Lambda function:

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



```
                                ion');

                                s-sdk");

// Set up the code to call when the &LAM; 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 <guilabel>Monitoring</guil
    console.log("Received event:", eventText);
```

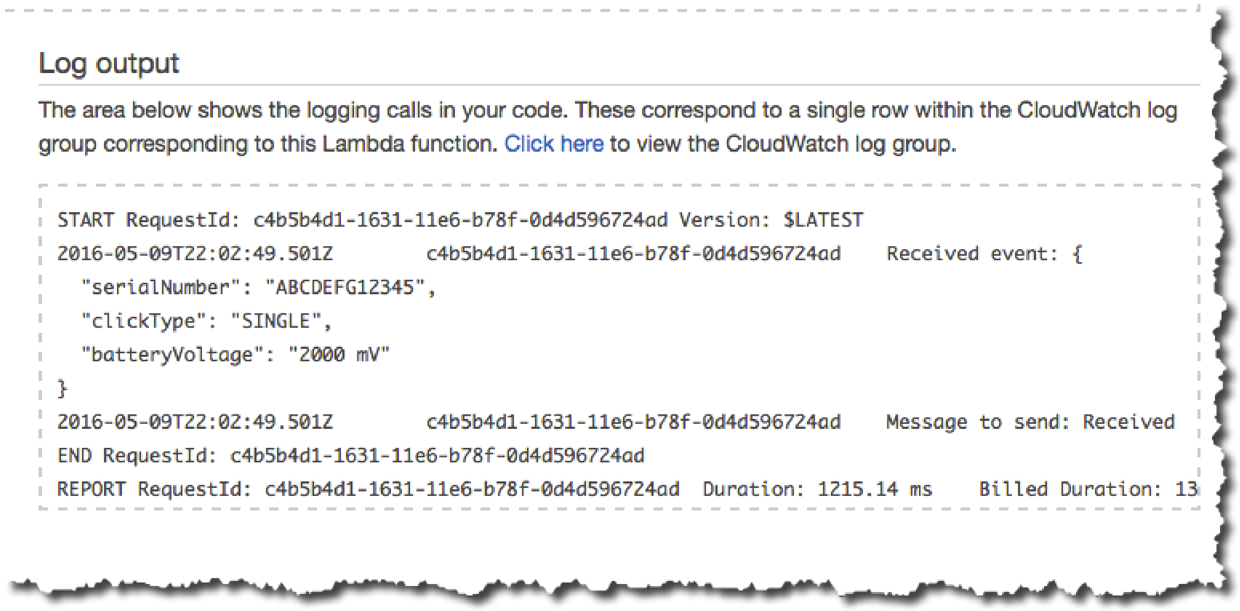
2. Copy and paste the following JSON into the **Input test event** page, and then choose **Save and test**.



It looks like you have not configured a test event for this function yet. Use the editor below to enter an event to test your function with (please remember that this will actually execute the code!). You can always edit the event later by choosing **Configure test event** in the Actions list. Note that changes to the event will only be saved locally.

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

3. In the AWS Lambda console, choose the **Monitoring** tab, and then scroll to the bottom of the screen. The **Log output** section displays the output the Lambda function has written to the console.



Log output

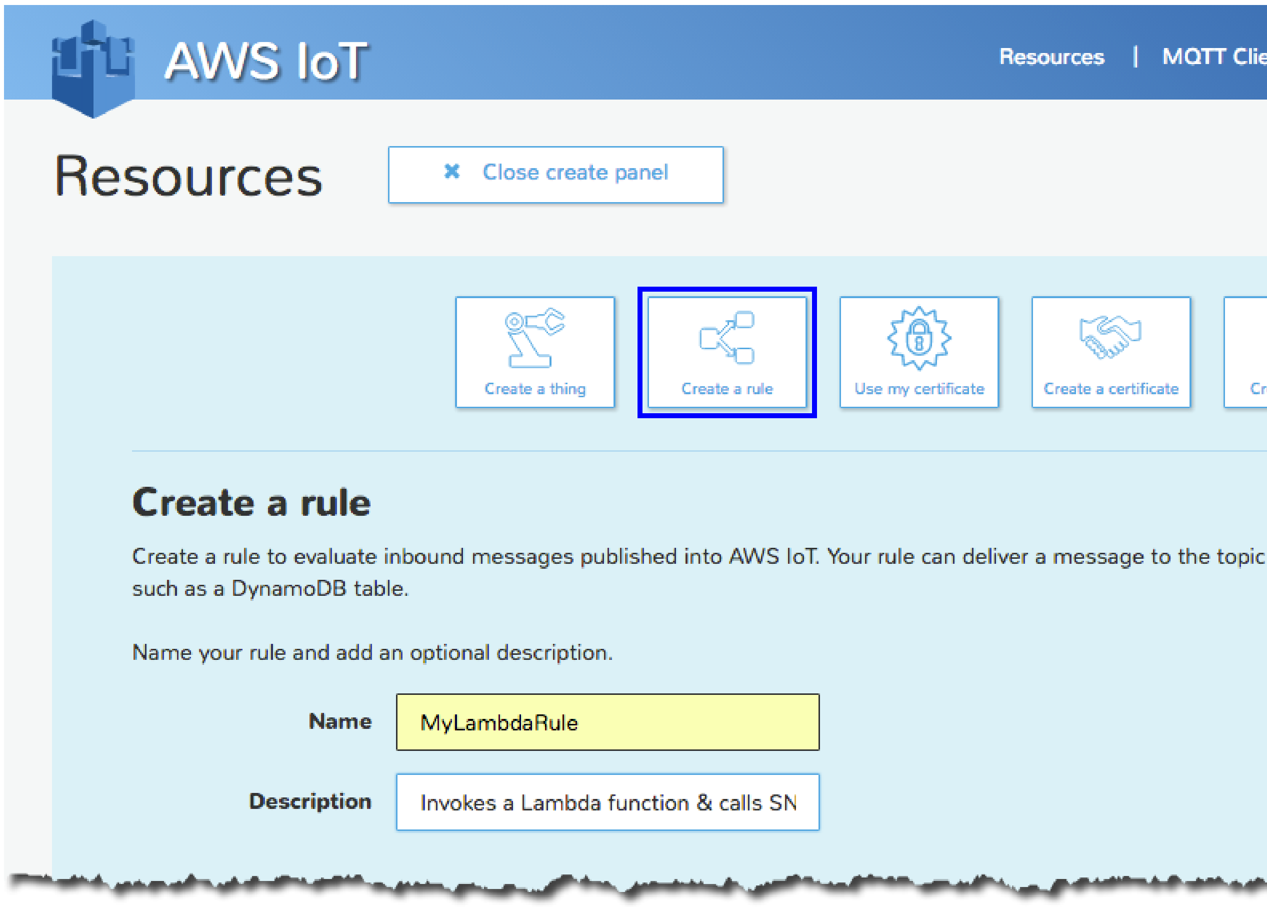
The area below shows the logging calls in your code. These correspond to a single row within the CloudWatch log group corresponding to this Lambda function. Click here to view the CloudWatch log group.

```
START RequestId: c4b5b4d1-1631-11e6-b78f-0d4d596724ad Version: $LATEST
2016-05-09T22:02:49.501Z        c4b5b4d1-1631-11e6-b78f-0d4d596724ad    Received event: {
  "serialNumber": "ABCDEFG12345",
  "clickType": "SINGLE",
  "batteryVoltage": "2000 mV"
}
2016-05-09T22:02:49.501Z        c4b5b4d1-1631-11e6-b78f-0d4d596724ad    Message to send: Received
END RequestId: c4b5b4d1-1631-11e6-b78f-0d4d596724ad
REPORT RequestId: c4b5b4d1-1631-11e6-b78f-0d4d596724ad  Duration: 1215.14 ms    Billed Duration: 13
```

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, choose **Create a resource**.
2. Choose **Create a rule**.
3. Type a name and description for the rule.



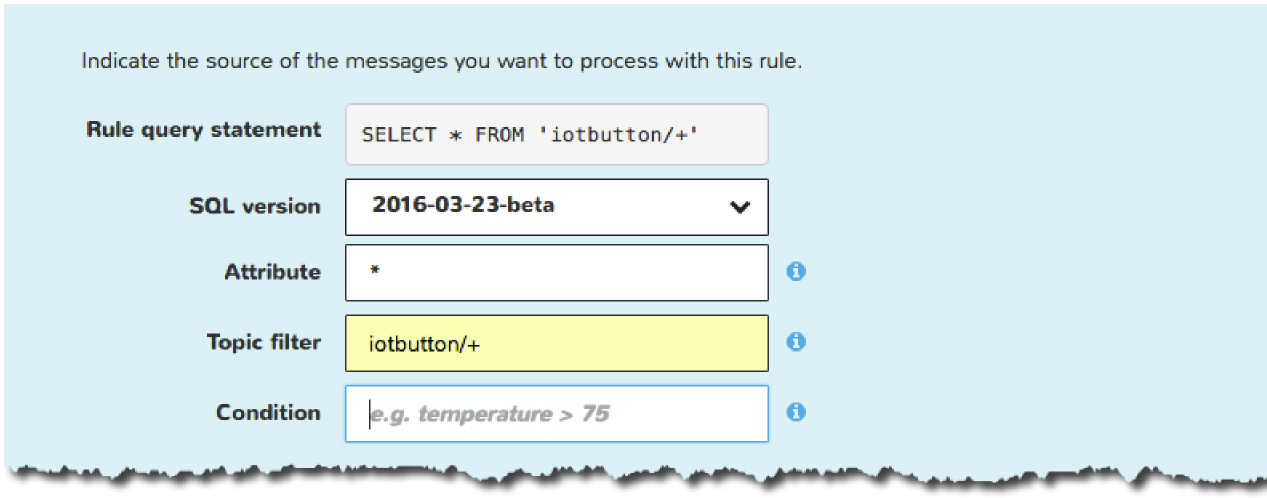
Create a rule

Create a rule to evaluate inbound messages published into AWS IoT. Your rule can deliver a message to the topic
such as a DynamoDB table.

Name your rule and add an optional description.

Name	MyLambdaRule
Description	Invokes a Lambda function & calls SN

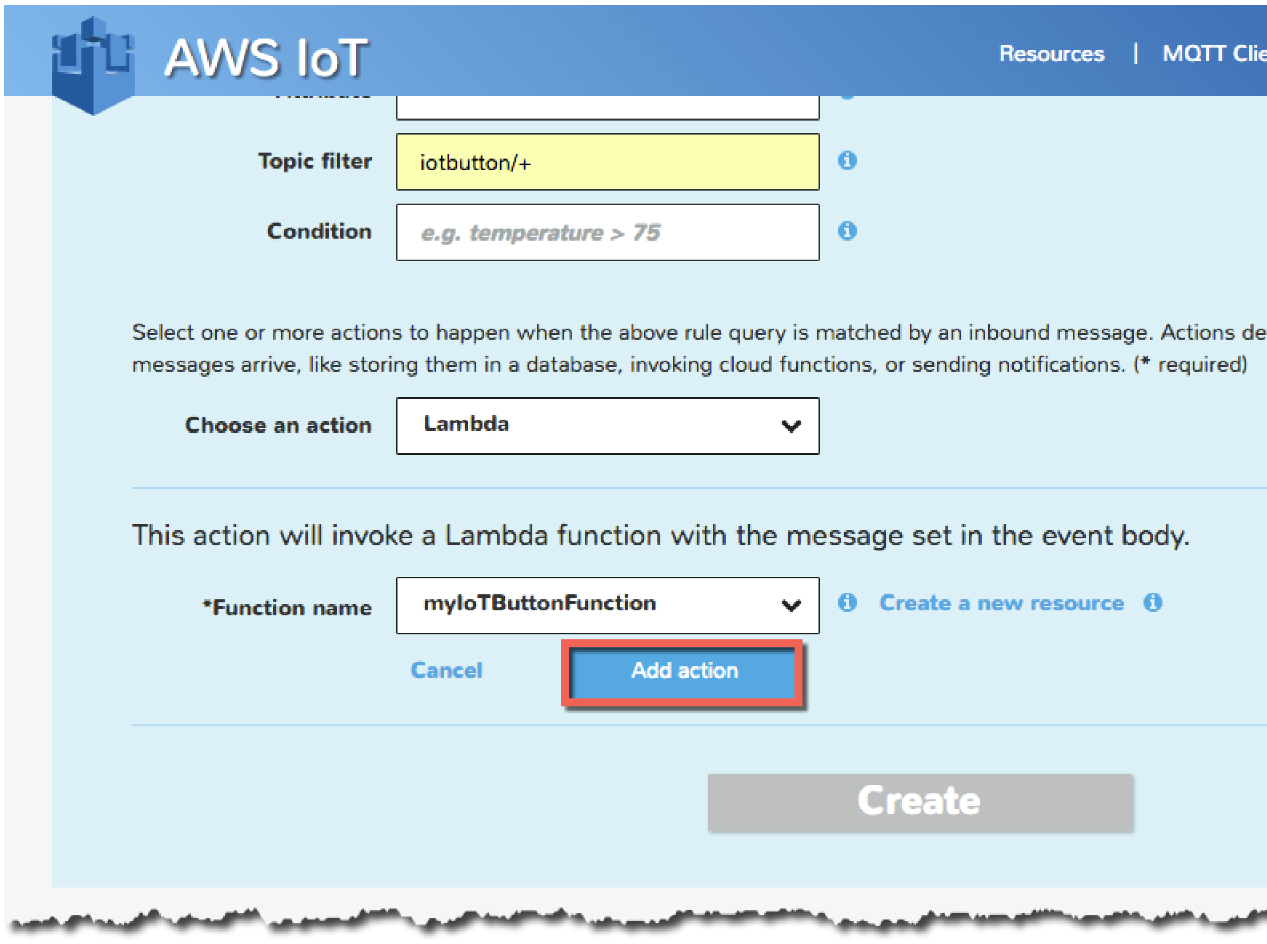
4. Enter the following settings for the rule:



Indicate the source of the messages you want to process with this rule.

Rule query statement	SELECT * FROM 'iotbutton/+'
SQL version	2016-03-23-beta
Attribute	*
Topic filter	iotbutton/+
Condition	e.g. temperature > 75

5. For **Choose an action**, choose **Insert this message into a code function and execute it (Lambda)**.
6. From **Function name**, choose your Lambda function name, and then choose **Add action**.



Topic filter: iotbutton/+

Condition: e.g. temperature > 75

Select one or more actions to happen when the above rule query is matched by an inbound message. Actions de messages arrive, like storing them in a database, invoking cloud functions, or sending notifications. (* required)

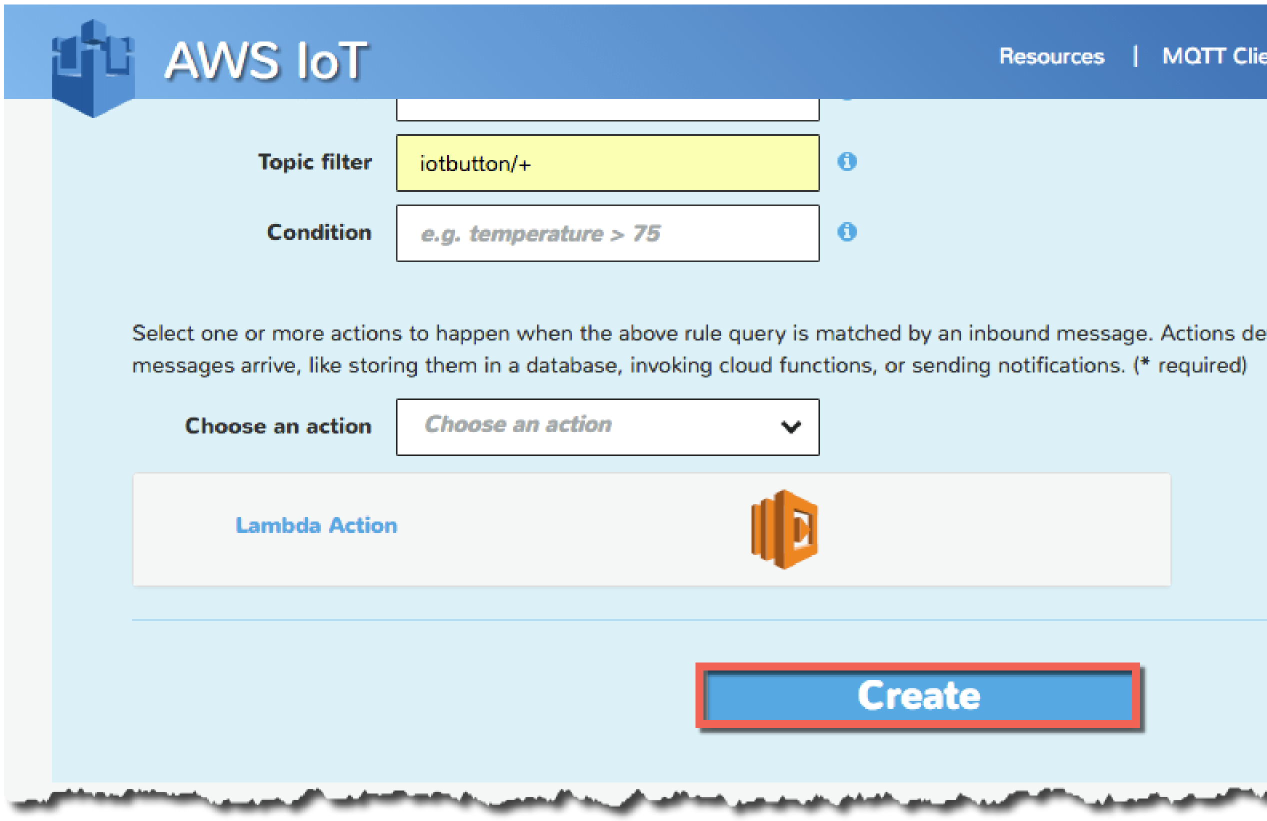
Choose an action: Lambda

This action will invoke a Lambda function with the message set in the event body.

*Function name: myIoTButtonFunction

Create a new resource

7. Choose **Create** to create your Lambda function.



Topic filter: iotbutton/+

Condition: e.g. temperature > 75

Select one or more actions to happen when the above rule query is matched by an inbound message. Actions de
messages arrive, like storing them in a database, invoking cloud functions, or sending notifications. (* required)

Choose an action: Choose an action

Test Your Lambda Rule

In this tutorial, we assume you have completed the AWS IoT Getting Started Tutorial (p. 18), which covers:

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

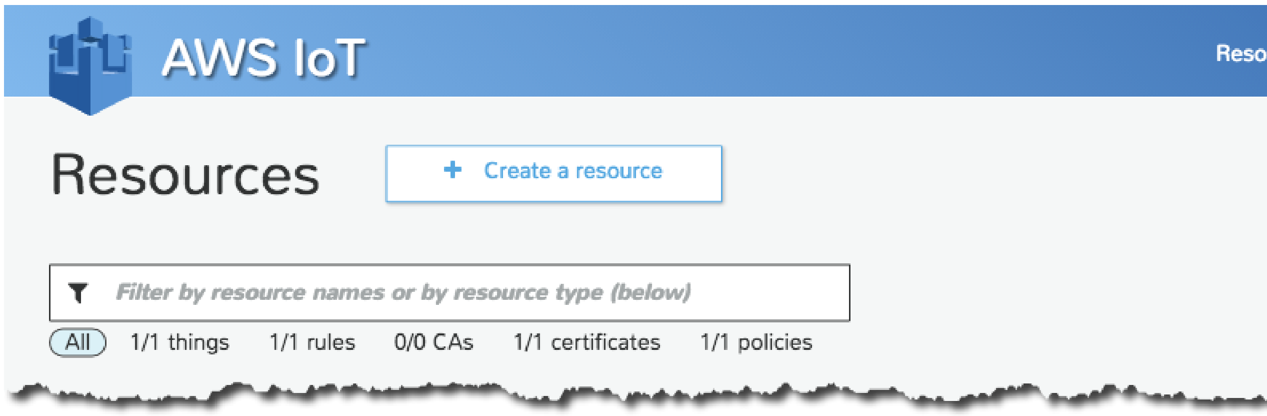
Now that your button is configured and connected to Wi-Fi and you have configured an Amazon SNS topic, you can press the button to test your Lambda rule. You should receive an SMS text message on your phone that contains the serial number of your button, the type of button press (SINGLE or DOUBLE), and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

The message should look like the following:

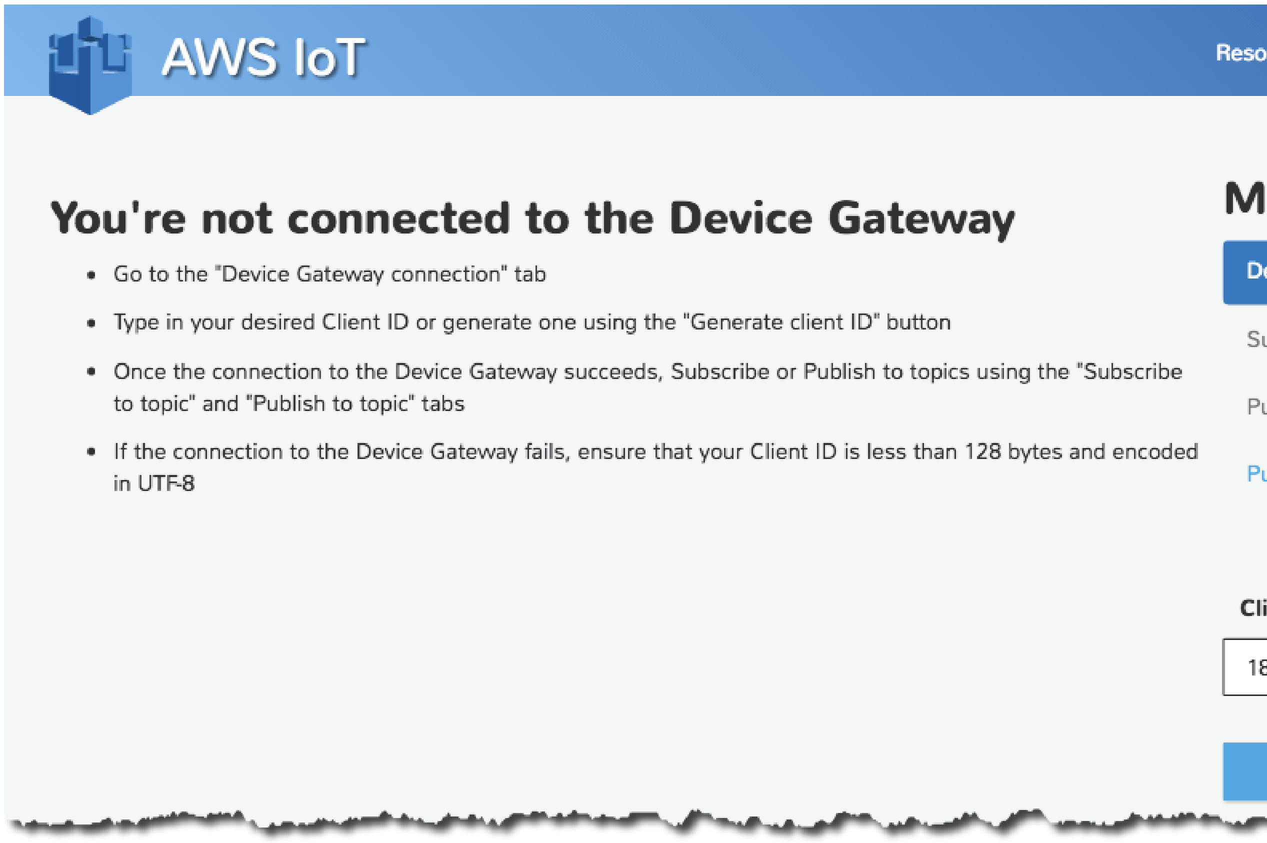
```
IOT BUTTON> {
    "serialNumber" : "ABCDEFG12345",
    "clickType" : "SINGLE",
    "batteryVoltage" : "2000 mV"
}
```

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 **MQTT Client**.



2. Type a client ID or choose **Generate client ID**, and then choose **Connect**.

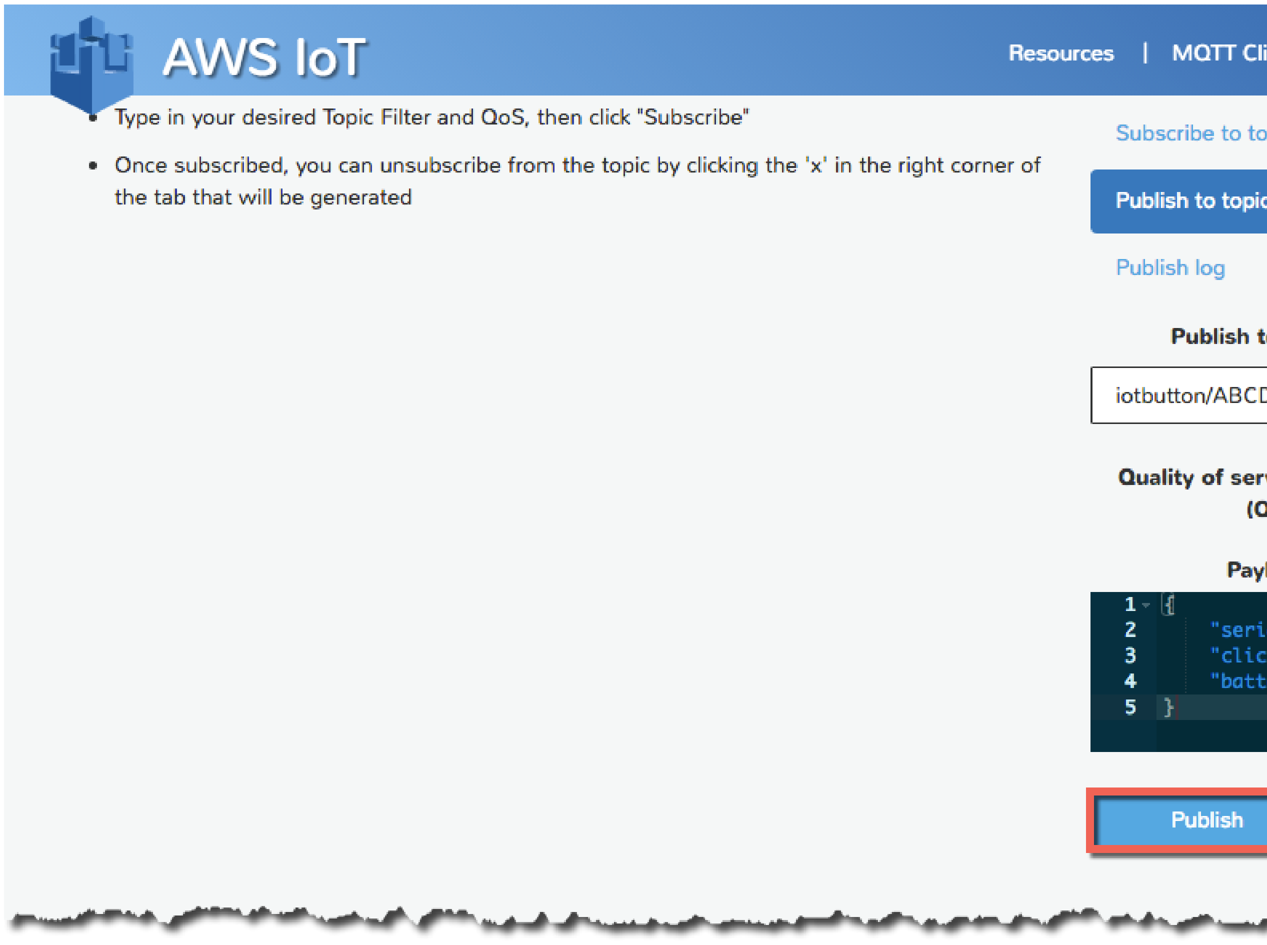


You're not connected to the Device Gateway

- Go to the "Device Gateway connection" tab
- Type in your desired Client ID or generate one using the "Generate client ID" button
- Once the connection to the Device Gateway succeeds, Subscribe or Publish to topics using the "Subscribe to topic" and "Publish to topic" tabs
- If the connection to the Device Gateway fails, ensure that your Client ID is less than 128 bytes and encoded in UTF-8

3. Choose **Publish to topic**.
4. In **Publish topic**, type `iotbutton/ABCDEFG12345`.
5. In **Payload**, type the following JSON, and then choose **Publish**.

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



- Type in your desired Topic Filter and QoS, then click "Subscribe"
- Once subscribed, you can unsubscribe from the topic by clicking the 'x' in the right corner of the tab that will be generated

iotbutton/ABCD

```
{
    "seri
    "clic
    "batt
}
```

You should receive a message on your cell phone.

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 "{\"at
tributes\": {\"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 "{\"at
tributes\": {\"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 "thingTy
peDescription=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\", \"mod
el\":\"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

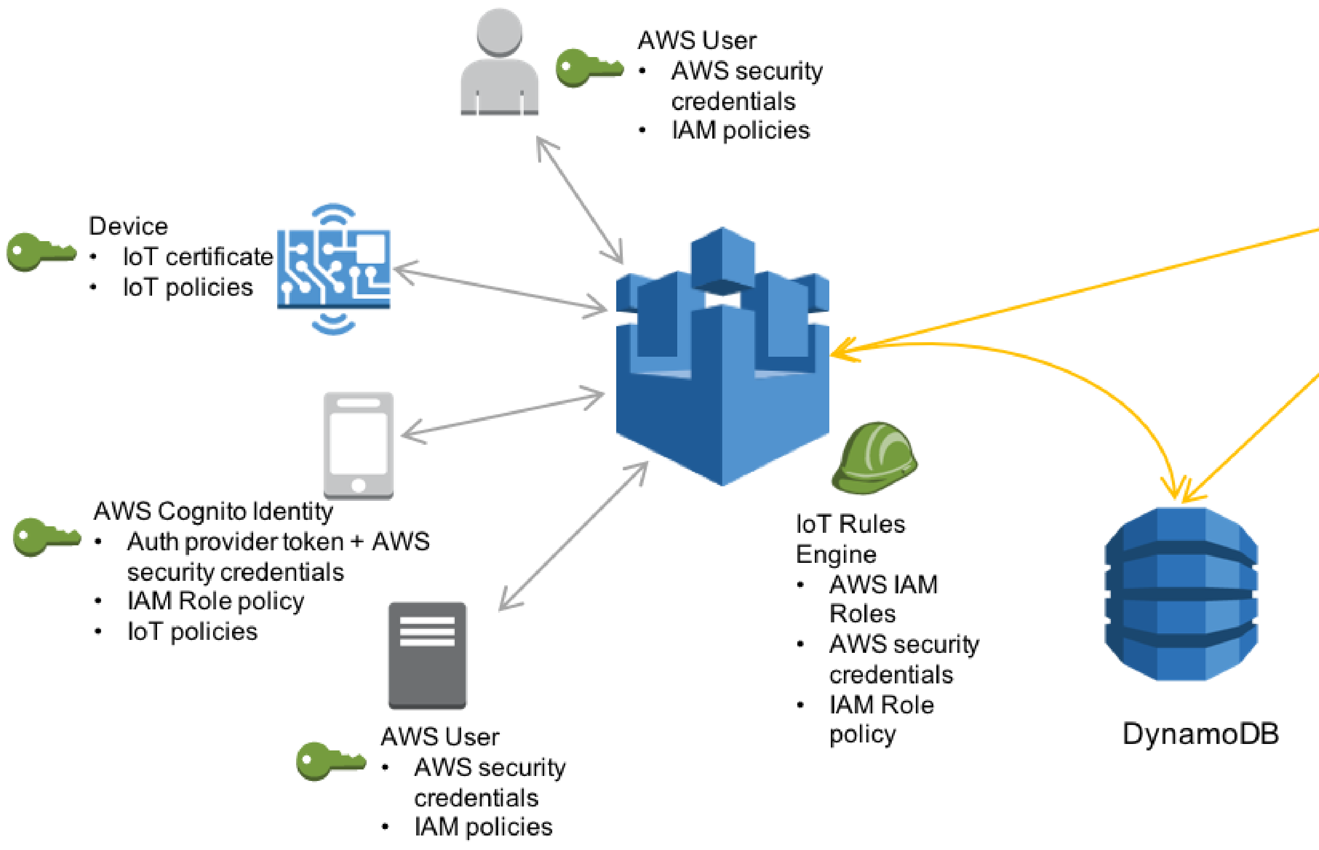
You can delete thing types only after they have been deprecated. To delete a thing type, use the `delete-thing-type` API:

```
$ 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. After data reaches 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 is responsible for leveraging AWS access management systems to securely transfer data to its final destination.

Authentication in AWS IoT

AWS IoT supports three types of identity principals for authentication:

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

Each identity type supports different use cases for accessing the AWS IoT message broker and Thing Shadows service.

The identity type you use depends on your choice of application protocol. If you use HTTP, use IAM (users, groups, roles) or Amazon Cognito identities. If you use MQTT, use X.509 certificates.

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.

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

X.509 certificates provide several benefits over other identification and authentication mechanisms. X.509 certificates enable asymmetric keys to be used with devices. Your manufacturing process and devices can be in control of keys. You do not need to rely on AWS for generating security credentials. This means you can burn private keys into secure storage on a device without ever allowing the sensitive cryptographic material to leave 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.

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 create and manage certificates. The following operations are available:

- 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 ensure your device is communicating with AWS IoT and not another server impersonating AWS IoT, copy the VeriSign root CA certificate onto your device. Reference the CA root certificate in your device code when connecting to AWS IoT. For more information, see the AWS IoT Device SDKs (p. 167).

> **Note**
> You cannot use your own CA certificate to authenticate the AWS IoT server, only the VeriSign 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)

You can use the UpdateCertificate API to revoke a certificate at any time. For more information about managing device certificates, see the AWS Command Line Interface User Guide.

1. Sign in to the AWS Management Console and open the AWS IoT console at https://console.aws.amazon.com/iot.
2. Choose **Create a resource**, and then choose **Create a certificate**.
3. Choose **1-Click certificate create**. Alternatively, to generate a certificate with a certificate signing request (CSR), choose the **Create with CSR** button.
4. Use the links to the public key, private key, and certificate to download each to a secure location.
5. The newly created certificate will be displayed as **INACTIVE**. Choose it, and from the **Actions** drop-down list, 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 ten CA certificates with the same subject field and public key per AWS account. 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 will be using an intermediate certificate to sign your device certificates, you must register the intermediate CA certificate. You should use the AWS IoT root CA certificate when connecting 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 create your own by using OpenSSL tools.

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, and pass both 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 privateKeyVerificationCert.key 2048
```

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

```
openssl req -new -key privateKeyVerificationCert.key -out privateKeyVerific
ationCert.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) []:XXXXXXXXXXXXMYREGISTRATIONCO
DEXXXXXX
Email Address []:
```

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

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

5. Register the CA certificate with AWS IoT, passing 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 -–veri
fication-cert file://privateKeyVerificationCert.crt
```

6. Activate the CA certificate using the `update-certificate` CLI command.

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

Creating a Device 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
```

2. Create a CSR for the device certificate.

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

You will be prompted for some additional 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.crt -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 registering your device certificate.

4. Register a device certificate.

   ```
   aws iot register-certificate -–certificate file://deviceCert.crt --caCerti
   ficate file://caCert.crt
   ```

5. Activate the device certificate using the `update-certificate` CLI command.

   ```
   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 registering 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 file://deviceCert.crt --caCertificate
 file://caCert.crt
```

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

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

Enable Auto Registration

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

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

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

```
aws iot register-ca-certificate -–ca-certificate file://rootCA.pem -–verifica
tion-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 will recognize the CA certificate as a registered CA certificate and will automatically register the device certificate and set 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 additional 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 attempt to register a device certificate, AWS will check if the associated CA certificate is `ACTIVE`. If the CA certificate is `INACTIVE`, AWS IoT will not allow the device certificate to be registered. By marking the CA certificate as INACTIVE, you are preventing any new device certificates issued by the compromised CA to be registered in your account. You can deactivate the CA certificate using the `update-ca-certificate` API:

```
$ aws iot update-ca-certificate --certificate-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 the device certificate.

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 any suspicious activity with a registered device certificate, you can revoke it by using the `update-certificate` API:

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

If any error or exception occurs during the auto-registration of the device certificates, AWS IoT will send the appropriate events or messages to your logs in CloudWatch Logs. For more information about setting up the CloudWatch 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. As with any other AWS service, you can use them to connect to AWS IoT HTTP interfaces using the AWS SDK and CLI.

IAM roles are also the basis for AWS IoT security in the cloud. Roles allow AWS IoT to issue calls to other AWS resources in your account on your behalf. If you want to have a device publish its state to a DynamoDB table, for example, IAM roles allow AWS IoT to do the heavy lifting securely. For more information, see IAM Roles.

For message broker connections, AWS IoT authenticates IAM users, groups, and roles using the Signature Version 4 signing process. For information about authentication with AWS security credentials, see Signing AWS API Requests.

When using AWS Signature Version 4 with AWS IoT, clients must support the following in their TLS implementation:

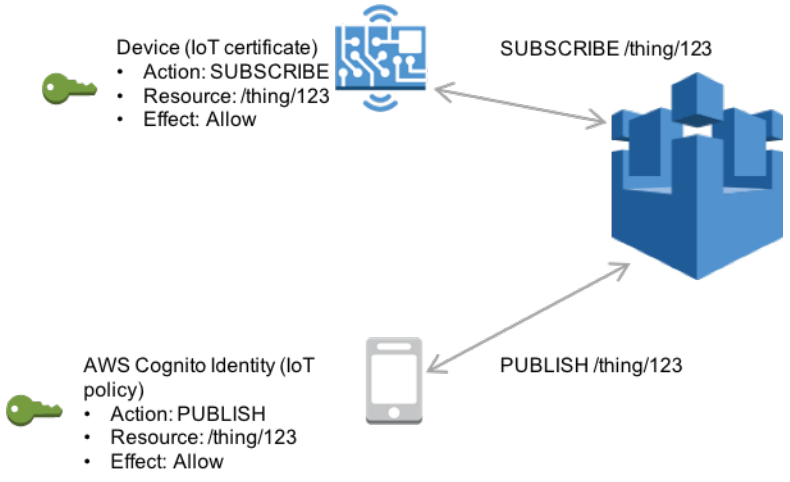
- TLS 1.2, TLS 1.1, TLS 1.0.
- SHA-256 RSA certificate signature validation.
- One of the cipher suites from the TLS cipher suite support section.

For information, see the IAM User Guide.

Amazon Cognito Identities

Amazon Cognito Identity allows you to use your own identity provider or leverage 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. This can be used to assign permissions between specific customers and their devices. For more information, see Amazon Cognito Identity.



Authorization

Communication with AWS IoT follows the principle of least privilege. An identity can execute AWS IoT operations only if you grant the appropriate permission. You create AWS IoT and IAM policies to give permissions to authenticated identities in AWS IoT.

Policies give permissions to AWS IoT clients regardless of the authentication mechanism they use to connect to AWS IoT. To control which resources a device can access, attach one or more AWS IoT policies to the certificate associated with the device. To control which resources a web or mobile application can access, attach one or more AWS IoT policies to the Amazon Cognito identity pool associated with the application. AWS IoT policies control access to AWS IoT resources (MQTT topics, devices, thing shadows, and so on). IAM policies control access to other AWS services and are attached to IAM users, groups, and roles.

Policy-based authorization is a powerful tool. It gives you complete control over the topics and topic filters in your AWS account. For example, consider a device connecting to AWS IoT with a certificate. You can open its access to all topics, or you can restrict its access to a single topic. The latter example allows you to assign a topic per device. For example, the device ID 123ABC can subscribe to `/device/123ABC` and you can grant other identities permission to subscribe to this topic, effectively opening a communication channel to this device.

AWS IoT Policies

AWS IoT policies are JSON documents. They follow the same conventions as IAM policies. For more information, see Overview of IAM Policies.

An AWS IoT policy looks like the following:

```
{
    "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": ["*"]
        }]
}
```

This policy allows the principal to connect and publish messages to AWS IoT.

Managing AWS IoT 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 Policy Actions

The following actions are available for use with AWS IoT:

iot:Publish
: Checked every time a PUBLISH request is sent to the broker. Used to allow clients to publish to specific topic patterns.

iot:Subscribe
: Checked every time a SUBSCRIBE request is sent to the broker. Used to allow clients to subscribe to topics that match specific topic patterns.

iot:Receive
: Checked every time a message is delivered to a client. Because the Receive permission is checked on every delivery, it can be used to revoke permissions to clients that are currently subscribed to a topic.

iot:Connect
: 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 Connect permission can be used to ensure only authorized clients can connect using a specific client ID.

iot:UpdateThingShadow
: Checked every time a request is made to update the state of a thing shadow document.

iot:GetThingShadow
: Checked every time a request is made to get the state of a thing shadow document.

iot:DeleteThingShadow
: Checked every time a request is made to delete the thing shadow document.

Action Resources

The following table shows the resource to specify for each action type:

Action	Resource
iot:DeleteThingShadow	thing ARN
iot:Connect	client ID ARN
iot:Publish	topic ARN
iot:Subscribe	topic filter ARN
iot:Receive	topic ARN
iot:UpdateThingShadow	thing ARN
iot:GetThingShadow	thing ARN

AWS IoT Policy Variables

AWS IoT defines two policy variables that can be used in AWS IoT policies: `iot:ClientId` and `aws:SourceIp`. When a policy is evaluated, the variables will be replaced by the actual values. `iot:ClientId` is replaced by the client ID that sent an MQTT or HTTP message. `aws:SourceIp` is replaced by the IP address from which the message originated.

The following AWS IoT policy illustrates 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}"

        ]
    }]
}
```

When you use policy variables like `${iot:ClientId}`, you can inadvertently open access to topics you do not want to be accessible. For example, if you use a policy that uses `${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 `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"
        ]
    }]
}
```

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>*:*<accountId>*:client/*<clientId>*

Topic ARN - arn:aws:iot:*<region>*:*<accountId>*:topic/*<topicName>*

Topic filter ARN - arn:aws:iot:*<region>*:*<accountId>*:topicfilter/*<topicFilter>*

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 ignature ersion 4 and Amazon Cognito.

Policies for MQTT Clients

The following policy allows the certificate holder using any client ID 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 using any client ID to publish to all topics in the AWS account:

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

The following policy allows the certificate holder using any client ID 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 using any client ID 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 using any client ID 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 using any client ID to publish on topic `foo` and subscribe to topic filter `foo/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": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topicfilter/foo/bar/*"
            ]
        }
    ]
}
```

The following policy allows the certificate holder using any client ID to publish on topic `foo` and prevents the certificate holder using any client ID 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 using any client ID 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 using any client ID to publish on the `arn:aws:iot:us-east-1:123456789012:topic/iotmonitor/provisioning/8050373158915119971` topic and allows the certificate holder using any client ID 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/provision
ing/8050373158915119971"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
               "arn:aws:iot:us-east-1:123456789012:topicfilter/iotmonitor/pro
visioning/8050373158915119971"
            ]
        }
    ]
}
```

Policies for HTTP and WebSocket Clients

For the following operations, AWS IoT uses 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 role policy attached to the pool and the 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 and allow identities from that pool to make a publish call. The second policy is attached to an Amazon Cognito user using the AWS IoT AttachPrincipalPolicy API and 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"
            ]
        }
    ]
}
```

IAM IoT Policies

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. When MQTT over the WebSocket protocol is used, only policies stored in IAM will apply to the WebSocket connection.
AWSIoTFullAccess	Allows the associated identity full access to all AWS IoT configuration and data operations.
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.

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"`.

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(creden
tials.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(cre
dentials.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_re
quest';
var canonicalQuerystring = 'X-Amz-Algorithm=' + algorithm;
canonicalQuerystring += '&X-Amz-Credential=' + encodeURIComponent(creden
tials.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(cre
   dentials.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 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/mois-ture/room1`, and so on.

Reserved Topics

Any topics beginning with $ are considered reserved and are not supported for publishing and subscribing except when working with the Thing Shadows service. For more information, see Thing Shadows.

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/ABCDE
FGHIJKLMNOPQRSTU: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/ABCDE
FGHIJKLMNOPQRSTU: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

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. 111).

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

When creating or replacing a rule, you must pass a role that controls the AWS resources to which the rule has access. The role must be defined in the same AWS account as the rule. The AWS IoT rules engine checks to make sure you have `iam:PassRole` permission to pass the role to the `create-topic-rule` API. To ensure you have this access, you need to create a policy that grants this access 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 also 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. 125).

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-beta",
  "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.

```
{
    "rule": {
        "awsIotSqlVersion": "2016-03-23-beta",
        "sql": "SELECT * FROM 'iot/test'",
        "ruleDisabled": false,
        "actions": [
            {
                "s3": {
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_s3",
                    "bucketName": "my-bucket",
                    "key": "myS3Key"
                }
            }
        ],
        "ruleName": "MyS3Rule"
    }
}
```

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-beta",
    "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-beta",
  "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-beta",
  "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-beta",
  "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-beta",
    "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-beta",
    "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. 111). 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-beta",
    "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-beta`, 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-beta 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, see Troubleshooting AWS IoT (p. 204).

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.
- `elasticsearch` to write data to a Amazon Elasticsearch Service domain.
- `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.
- `firehose` to write data to an Amazon Kinesis Firehose stream.
- `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 alarm.
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.

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.

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:ac
count_id:rule/rule_name --source-account "account_id" --statement-id "unique_id"
 --action "lambda:InvokeFunction"
```

The following are the parameters 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`.

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:func
tion:myLambdaFunction"
             }
        }]
    }
}
```

For more information, see the AWS Lambda Developer Guide.

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.

key
: The path to the file where the data is written. For example, if the value of this parameter 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.

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.

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.amazon
aws.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.

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"
            }
        }]
    }
}
```

AWS IoT SQL Reference

This reference focuses on the differences between ANSI SQL and AWS IoT SQL. If you are not familiar with ANSI SQL, see the W3Schools SQL Tutorial.

All rules include a SQL statement that consists of a SELECT clause and an optional WHERE clause. The SELECT clause allows you to extract one or more JSON objects or attributes from the MQTT message payload. The WHERE clause allows you to filter the JSON objects or attributes extracted by the SELECT clause.

All data processed by a SELECT query is assumed to be in JSON format unless certain functions are used within the query. If the data is in JSON format, it may contain either a single root object or a single root object with nested objects. For more information about functions applicable to non-JSON data, see SQL Functions (p. 128).

As with ANSI SQL, white space is insignificant and keywords are not case-sensitive. Strings and JSON properties are case-sensitive. In our examples, all keywords are capitalized following common practice for SQL.

Expressions

Select, FROM, and WHERE clauses are all composed of expressions. The following expressions are allowed in AWS IoT.

Token	Meaning	Example
=	Equal, comparison	color = 'red'
<>	Not equal, comparison	color <> 'red'
AND	Logical AND	color = 'red' AND siren = 'on'
OR	Logical OR	color = 'red' OR siren = 'on'
( )	Parenthesis, grouping	color = 'red' AND (siren = 'on' OR isTest)
$	Dollar sign, used in reserved topic names. When specifying a reserved topic name in a substitution template, the '$' must be escaped using another '$'	"$$aws/things/mything/shadow/update/accepted"
+	Addition, arithmetic	4 + 5
-	Subtraction, arithmetic	5 - 4
/	Division, arithmetic	20 / 4
*	Multiplication, arithmetic	5 * 4
%	Modulo division, arithmetic	20 % 6
<	Less than, comparison	5 < 6
<=	Less than or equal, comparison	5 <= 6
>	Greater than, comparison	6 > 5

Token	Meaning	Example
>=	Greater than or equal, comparison	6 >= 5
Function call	A invocation of an SQL function	clientId()
A JSON extension expression	An expression that selects a specific value in a JSON document.	state.desired.color

CASE ... WHEN ...
 	
THEN ... ELSE ...
 	
END
	Case statement	CASE location WHEN 'home'
		
		THEN 'off' WHEN 'work'
		
		THEN 'on' ELSE 'silent' END

SELECT Clause

The AWS IoT SELECT clause is essentially the same as the ANSI SQL SELECT clause, with a few minor differences. The "AS" keyword is required with AWS IoT SELECT clauses unless you are using JSON extensions ('.' syntax), for example:

```
SELECT state.temperature FROM 'mydevices/device1'
```

For more information about JSON extensions, see JSON Extensions (p. 132).

Unlike querying relational databases, you can use `SELECT *` without the concern of retrieving too much information. The information returned by the `SELECT * AS some_name` clause contains only the JSON payload of the MQTT message. The message can be returned without parsing, which possibly improves performance of the query. SELECT queries for a single object JSON document are of the form: SELECT <object> AS <object-name>. The <object> can be any object or attribute name in the JSON document. <object-name> provides a name for the result of the SELECT query. SELECT queries for JSON documents with nested objects are of the form: SELECT <object> where <object> is the root object, any level of nested objects, and an optional attribute. For example:

- object
- object.nestedObject
- object.nestedObject1.nestedObject2
- object.nestedObject1.nestedObject2.attribute

You can specify as many layers of nested objects as there are in the JSON document.

FROM Clause

In ANSI SQL, you select data from tables. In AWS IoT SQL, you select data from JSON properties in MQTT messages.

The data source, which is the topic to which MQTT messages are sent, is specified using the `MQTT()` function, as shown in this example:

```
SELECT * FROM mqtt('com.example/sensors/+')
```

However, the FROM clause assumes you are querying an MQTT message, so you can omit the call to the `mqtt()` function and specify only the topic name, as shown in the following example:

```
SELECT * FROM 'com.example/sensors'
```

WHERE Clause

The WHERE clause filters the message data returned by the SELECT clause based on JSON attribute values. For example, suppose you have a topic filter from an MQTT topic, `iot/thing/#`. 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
    }
}
```

You could use the following SQL statement in your rule to query the `iot/thing/#` topic and extract the sensor data when the `temp` field is above 50.

```
SELECT * FROM 'iot/thing/#' WHERE temp > 50
```

Functions

You can use the following built-in functions in the SELECT or WHERE clauses of your SQL expressions.

Function	Description	SQL Version
abs(*number*)	Returns the absolute value.	2015-10-08 and later.
accountId()	Returns the account ID of the MQTT client sending the message, or undefined if the message didn't come through MQTT.	2015-10-08 and later.
asin(*number*)	Returns the arcsine.	2015-10-08 and later.
atan(*number*)	Returns the arctangent.	2015-10-08 and later.
bitand(*number1*, *number2*)	Returns the result of a bitwise AND operation.	2015-10-08 and later.

Function	Description	SQL Version
cast(value as type)	Converts the value to the specified data type. Supported data types are: double Converts the value to a double. float Converts the value to a double. int Converts the value to an integer. integer Converts the value to an integer. ntext Converts the value to a string. num Converts the value to a double. number Converts the value to a double. nvarchar Converts the value to a string. string Converts the value to a string. text Converts the value to a string. varchar Converts the value to a string.	2015-10-08 and later.
ceil(*number*)	Returns the result of rounding up to the nearest integer.	2015-10-08 and later.
chr(*number*)	Returns the ASCII character represented by *number*.	2015-10-08 and later.
clientId()	Returns the client ID of the MQTT client sending the message, or undefined if the message didn't come through MQTT.	2015-10-08 and later.
concat(*string1*, *string2*)	Returns the concatenation of two strings.	2015-10-08 and later.
cos(*number*)	Returns the cosine.	2015-10-08 and later.
cosh(*number*)	Returns the hyperbolic cosine.	2015-10-08 and later.
encode(*value*, *encoding-Scheme*)	Encodes the value based on the provided encoding scheme. The function returns a string representation of the encoded payload.	2016-03-23-beta and later.
endswith(*input*, *suffix*)	Returns true if *input* ends with *suffix*.	2015-10-08 and later.
exp(*number*)	Returns e to the power of the specified number.	2015-10-08 and later.
floor(*number*)	Returns the result of rounding down to the nearest integer.	2015-10-08 and later.

Function	Description	SQL Version
get_thing_shadow(*thingName*, *roleArn*	Returns the thing shadow of the specified thing. thingName The name of the thing whose state you want to retrieve. roleArn A role with `iot:GetThingShadow` permission.	2016-03-23-beta and later.
ln(*number*)	Returns the natural logarithm.	2015-10-08 and later.
log(*n*, *m*)	Returns the logarithm of *n* base *m*.	2015-10-08 and later.
lower(*string*)	Returns the result of converting all characters to lowercase.	2015-10-08 and later.
lpad(*string*, *n*)	Adds *n* spaces to the left side of *string*.	2015-10-08 and later.
ltrim(*string*)	Removes all white space from the left side of *string*.	2015-10-08 and later.
machinelearning_predict(*modelId*, *roleArn*, *record*)	Runs a prediction against the specified model, role ARN, and record.	2015-10-08 and later.
md2(*string*)	Returns the MD2 hash value.	2015-10-08 and later.
md5(*string*)	Returns the MD5 hash value.	2015-10-08 and later.
mod(*m*, *n*)	Returns the remainder of *m* divided by *n*.	2015-10-08 and later.
nanvl(*value*, *default*)	Returns *value* if it's non-null, and *default* otherwise.	2015-10-08 and later.
power(*m*, *n*)	Returns *m* raised to the *n*th power.	2015-10-08 and later.
remainder(*m*, *n*)	Returns the remainder of *m* divided by *n*.	2015-10-08 and later.
replace(*source*, *substring*, *replacement*)	Returns *source* with all occurrences of *substring* replaced by *replacement*.	2015-10-08 and later.
round(*number*, *precision*)	Returns the result of rounding *number* to *precision* decimal places. If *precision* is 0, the function rounds to the nearest whole number.	2015-10-08 and later.
rpad(*string*, *n*)	Adds *n* spaces to the right side of *string*.	2015-10-08 and later.
rtrim(*string*)	Removes all white space from the right side of *string*.	2015-10-08 and later.
sign(*number*)	Returns a value indicating the sign of a number. If number < 0, then -1. Else, if number = 0, then 0. Else, if number > 0, then 1.	2015-10-08 and later.
sin(*number*)	Returns the sine.	2015-10-08 and later.
sinh(*number*)	Returns the hyperbolic sine.	2015-10-08 and later.

Function	Description	SQL Version
sqrt(*number*)	Returns the square root.	2015-10-08 and later.
startswith(*input*, *prefix*)	Returns true if *input* starts with *prefix*.	2015-10-08 and later.
tan(*number*)	Returns the tangent.	2015-10-08 and later.
tanh(*number*)	Returns the hyperbolic tangent.	2015-10-08 and later.
traceId()	Returns the trace ID of the MQTT message, or undefined if the message didn't come through MQTT.	2015-10-08 and later.
topic(*number*)	Returns the specified topic segment. For example, if the topic is foo/bar, topic() returns "foo/bar", topic(1) returns "foo" and topic(2) returns "bar".	2015-10-08 and later.
trunc(*number*, *precision*)	Returns the result of truncating *number* to *precision* decimal places.	2015-10-08 and later.
upper(*string*)	Returns the result of converting all characters to uppercase.	2015-10-08 and later.
sha1(*string*)	Returns the SHA-1 hash value.	2015-10-08 and later.
sha224(*string*)	Returns the SHA-224 hash value.	2015-10-08 and later.
sha256(*string*)	Returns the SHA-256 hash value.	2015-10-08 and later.
sha512(*string*)	Returns the SHA-512 hash value.	2015-10-08 and later.
rand()	Returns a random number between 0 and 1.	2015-10-08 and later.
newuuid()	Returns a random 16-byte UUID.	2015-10-08 and later.
timestamp()	Returns the current Unix timestamp, as observed by the current server.	2015-10-08 and later.

Making Predictions with Amazon Machine Learning in an AWS IoT Rule

Use the `machinelearning_predict` function to make predictions using the data from an MQTT message based on an Amazon ML model. The parameters for the `machinelearning_predict` function are:

modelId
: The ID of the model to run the prediction against. 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 Machine Learning 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 Machine Learning to make predictions. The value must be SGD.

predictedScores
: Contains the raw classification score corresponding to each label.

predictedValue
: The value predicted by Amazon Machine Learning.

Encode the Payload Before Further Processing

Use the `encode` function to encode the payload, which potentially might be non-JSON data, into its string representation based on the encoding scheme.

value
: Any of the valid expressions, as defined in SQL Syntax (p. 126). 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.

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.

* 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.

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 Expressions (p. 126).

Substitution templates appear in the SELECT clause within a rule, for example:

```
{
    "sql": "SELECT *, topic() AS topic FROM 'my/iot/topic'",
    "ruleDisabled": false,
    "actions": [{
        "republish": {
            "topic": "${topic()}",
            "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 a number of 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 a thing shadow.

$aws/things/myLightBulb/shadow/update/rejected
: The Thing Shadows service sends messages to this topic when an update to a 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 a thing shadow.

$aws/things/myLightBulb/shadow/get/accepted
: The Thing Shadows service sends messages to this topic when a request for a thing shadow is made successfully.

$aws/things/myLightBulb/shadow/get/rejected
: The Thing Shadows service sends messages to this topic when a request for a thing shadow is rejected.

$aws/things/myLightBulb/shadow/delete/accepted
: The Thing Shadows service sends messages to this topic when a thing shadow is deleted.

$aws/things/myLightBulb/shadow/delete/rejected
: The Thing Shadows service sends messages to this topic when a request to delete a thing shadow is rejected.

To learn more about all of the MQTT topics used by the Thing Shadows service, see Device Shadow MQTT Topics (p. 156).

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"
        }
    }
}
```

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.

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 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"
}
```

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}
```

If you want 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 $) are ignored. To work around this issue, register an LWT message to a non-reserved topic and create a rule that republishes the message on the reserved topic. The following example shows how to create a republish rule that listens for a messages from the `my/things/myLightBulb/update` topic and republishes it to `$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`:

```
{
    "reported":
    {
        "connected":"false"
    }
}
```

It also publishes a message on its update topic (`$aws/things/myLightBulb/shadow/update`), setting its connected state to `true`:

```
{
    "reported":
    {
        "connected":"true"
    }
}
```

When the device disconnects gracefully, it publishes a message on its update topic and sets its connected state to `false`:

```
{
    "reported":
    {
        "connected":"false"
    }
}
```

If the device disconnects due to an error, its LWT message is posted automatically to the update topic.

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:

```
{
  "messageNumber": 2,
  "payload": {
    "version": 3,
    "timestamp": 1469564968
  },
  "qos": 0,
  "timestamp": 1469564968492,
  "topic": "$aws/things/myLightBulb/shadow/delete/accepted"
}
```

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. 162).

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 startup 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. 156) and Device Shadow RESTful API (p. 154).

Updating a Thing Shadow

You can update a thing shadow by using the UpdateThingShadow (p. 155) RESTful API or by publishing to the /update (p. 157) 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. 154) RESTful API or by subscribing and publishing to the /get (p. 159) 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. 157) 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. 156) RESTful API or by publishing to the /delete (p. 161) topic.

Initial state:

```
{
    "state": {
        "desired" : {
            "lights": { "color": "RED" },
            "engine" : "ON"
        },
        "reported" : {
            "lights" : { "color": "GREEN"  },
            "engine" : "OFF"
        }
    }
}
```

A message is sent 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 "+" wildcard 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

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. 163).

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. 163).

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. 163).

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: ignature ersion 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: gnature 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 $aws/things/*thingName*/shadow. Publishing and subscribing on thing shadow topics requires topic-based authorization. AWS IoT reserves the right to add new topics to the existing topic structure. For this reason, we recommend that you avoid wildcard subscriptions to shadow topics. For example, avoid subscribing to topic filters like `$aws/things/thingName/shadow/#` because the number of topics that match this topic filter might increase as AWS IoT introduces new shadow topics.

The following are the MQTT topics used for interacting with thing shadows.

Topics

- /update/documents (p. 158)
- /update/rejected (p. 158)
- /update/delta (p. 159)
- /get (p. 159)
- /get/accepted (p. 160)
- /get/rejected (p. 160)
- /delete (p. 161)
- /delete/accepted (p. 161)
- /delete/rejected (p. 162)

/update

A thing publishes a request state document to this topic to update the thing shadow:

```
$aws/things/thingName/shadow/update
```

AWS IoT responds by publishing to either /update/accepted (p. 157) or /update/rejected (p. 158).

For more information, see Request State Documents (p. 163).

Example Policy

The following is an example policy:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": ["iot:Publish"],
       "Resource": ["arn:aws:iot:region:account:topic/$aws/things/thingName/shad
ow/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. 163).

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/thing
Name/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/thing
Name/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. 164).

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/thing
Name/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. 163).

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 not 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/thing
Name/shadow/update/delta"]
    }]
}
```

/get

A thing publishes to this topic to get the thing shadow:

```
$aws/things/thingName/shadow/get
```

AWS IoT responds by publishing to either /get/accepted (p. 160) or /get/rejected (p. 160).

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/shad
ow/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. 163).

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/thing
Name/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. 164).

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/thing
Name/shadow/get/rejected"]
    }]
}
```

/delete

A thing publishes a document to this topic to delete a thing shadow:

```
$aws/things/thingName/shadow/delete
```

To delete a thing shadow, send a message to the delete topic. The content of the message is ignored.

AWS IoT responds by publishing to either /delete/accepted (p. 161) or /delete/rejected (p. 162).

Example Policy

The following is an example policy:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": [
            "iot:Subscribe",
            "iot:Receive"
        ],
       "Resource": ["arn:aws:iot:region:account:topic filter/$aws/things/thing
Name/shadow/delete"]
    }]
}
```

/delete/accepted

AWS IoT publishes a message to this topic when deleting a thing shadow:

```
$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/thing
Name/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. 164).

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/thing
Name/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. 154) or MQTT Pub/Sub Messages (p. 156). For more information, see Device Shadows Documents (p. 141).

Examples

- Request State Documents (p. 163)
- Response State Documents (p. 163)
- Error Response Documents (p. 164)

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. 165).

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
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.

HTTP Error Code	Error Messages
500 (Internal Server Error)	• Internal service failure

AWS IoT SDKs

Contents

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

Android SDK

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 Android SDK on GitHub
- AWS Android SDK Readme
- AWS Android SDK 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 Embedded C SDK

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:

- Embedded C SDK on GitHub
- Embedded C SDK Readme
- Embedded C SDK 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 Java SDK

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, providing access to thing shadows using HTTP methods, including GET, UPDATE, and DELETE. It 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 SDK for Java on GitHub
- AWS IoT Java SDK readme

AWS IoT JavaScript SDK

The aws-iot-device-sdk.js package allows developers to write JavaScript applications that access AWS IoT using MQTT or MQTT over the secure WebSocket protocol. It can be used in Node.js environments and browser applications. For more information, see the following:

- AWS IoT SDK for JavaScript on GitHub
- AWS IoT 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 SDK for Python on GitHub
- AWS IoT SDK for Python readme

Getting Started with AWS IoT on the Raspberry Pi and the AWS IoT Embedded C SDK

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. After following the steps in this guide, you will be able to get connected to the AWS IoT platform and run sample apps included with the AWS IoT Embedded C SDK.

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:

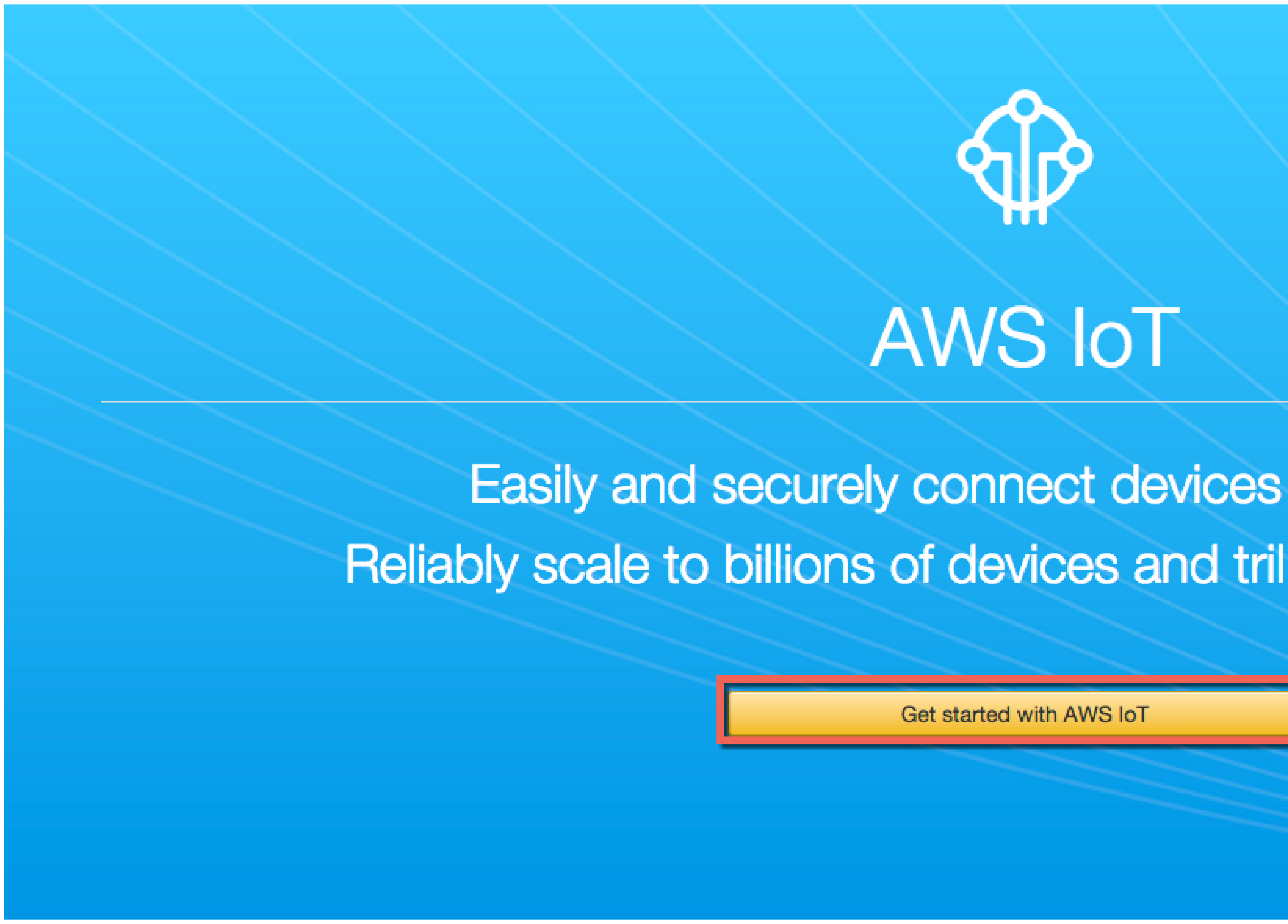
- Raspberry Pi 2 Model B
- Raspbian Wheezy
- Iceweasel browser

Connecting Your Raspberry Pi

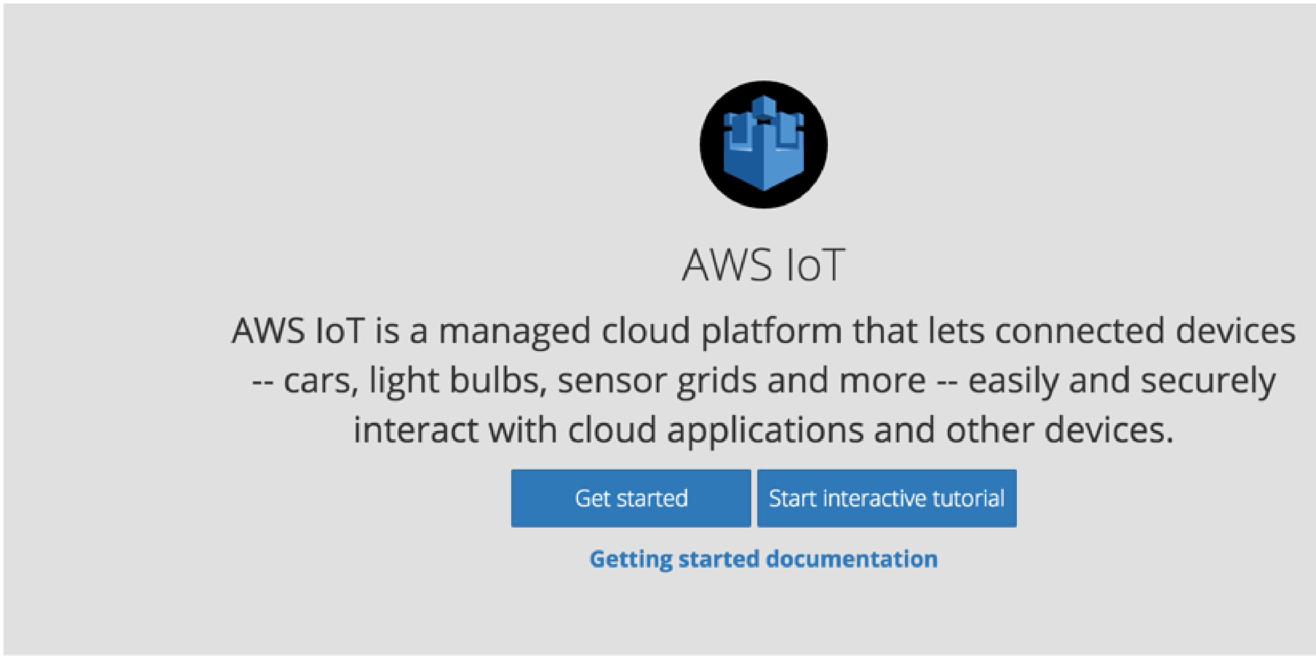
Sign in to the AWS IoT Console

Turn on your Raspberry Pi and confirm you have an Internet connection.

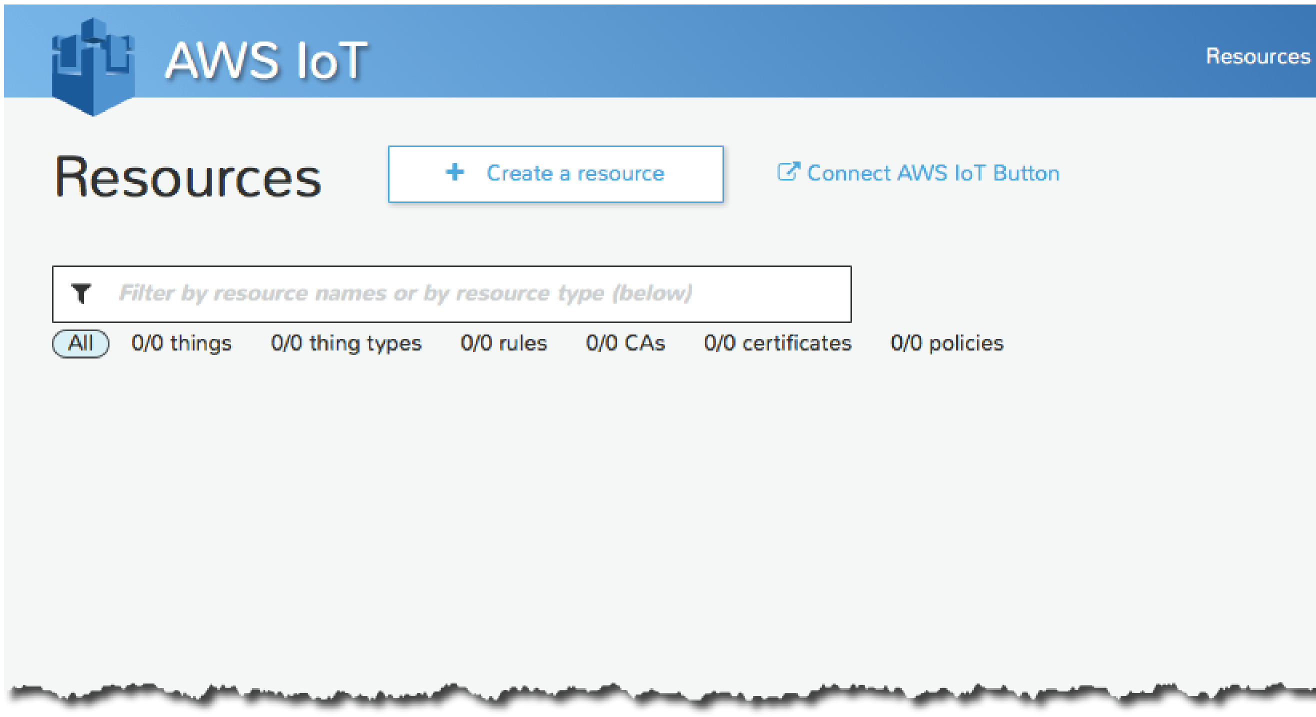
Sign in to the AWS Management Console and open the AWS IoT console at https://aws.amazon.com/iot. On the **Welcome** page, choose **Get started with AWS IoT**.


AWS IoT
Easily and securely connect devices
Reliably scale to billions of devices and tril
Get started with AWS IoT

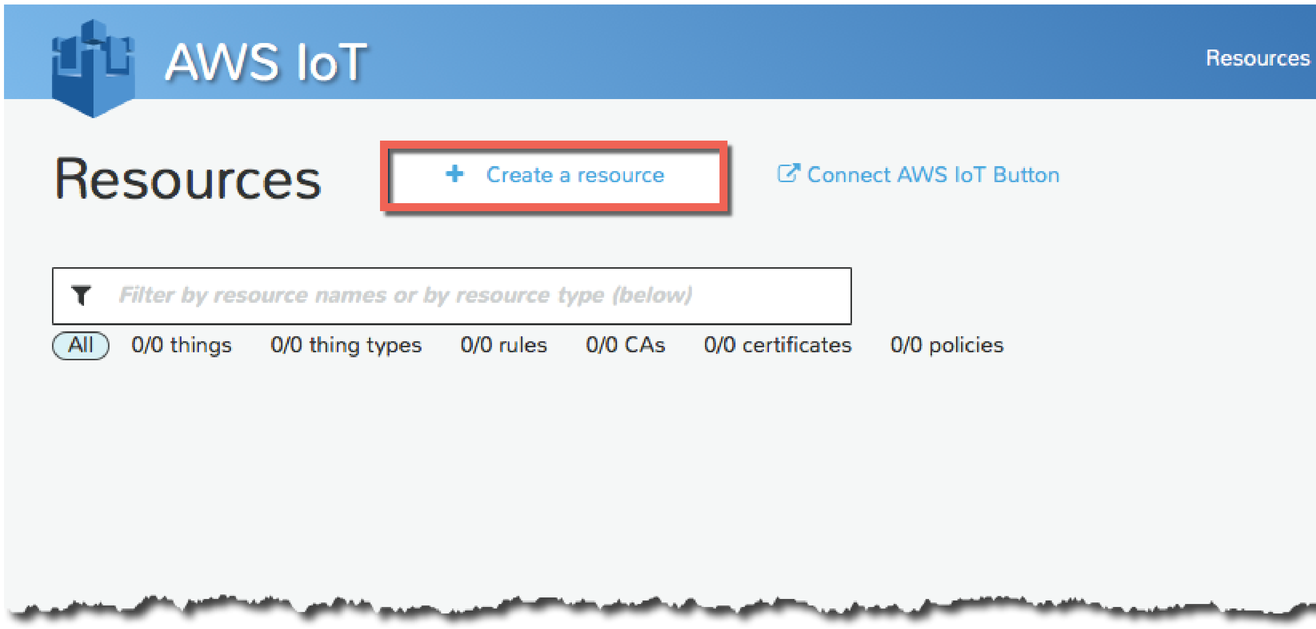
If this is your first time using the AWS IoT console, you will see two buttons: **Get Started** and **Start interactive tutorial**.


AWS IoT
AWS IoT is a managed cloud platform that lets connected devices -- cars, light bulbs, sensor grids and more -- easily and securely interact with cloud applications and other devices.
Get started
Start interactive tutorial
Getting started documentation

Choose **Get Started**. The following page should appear.



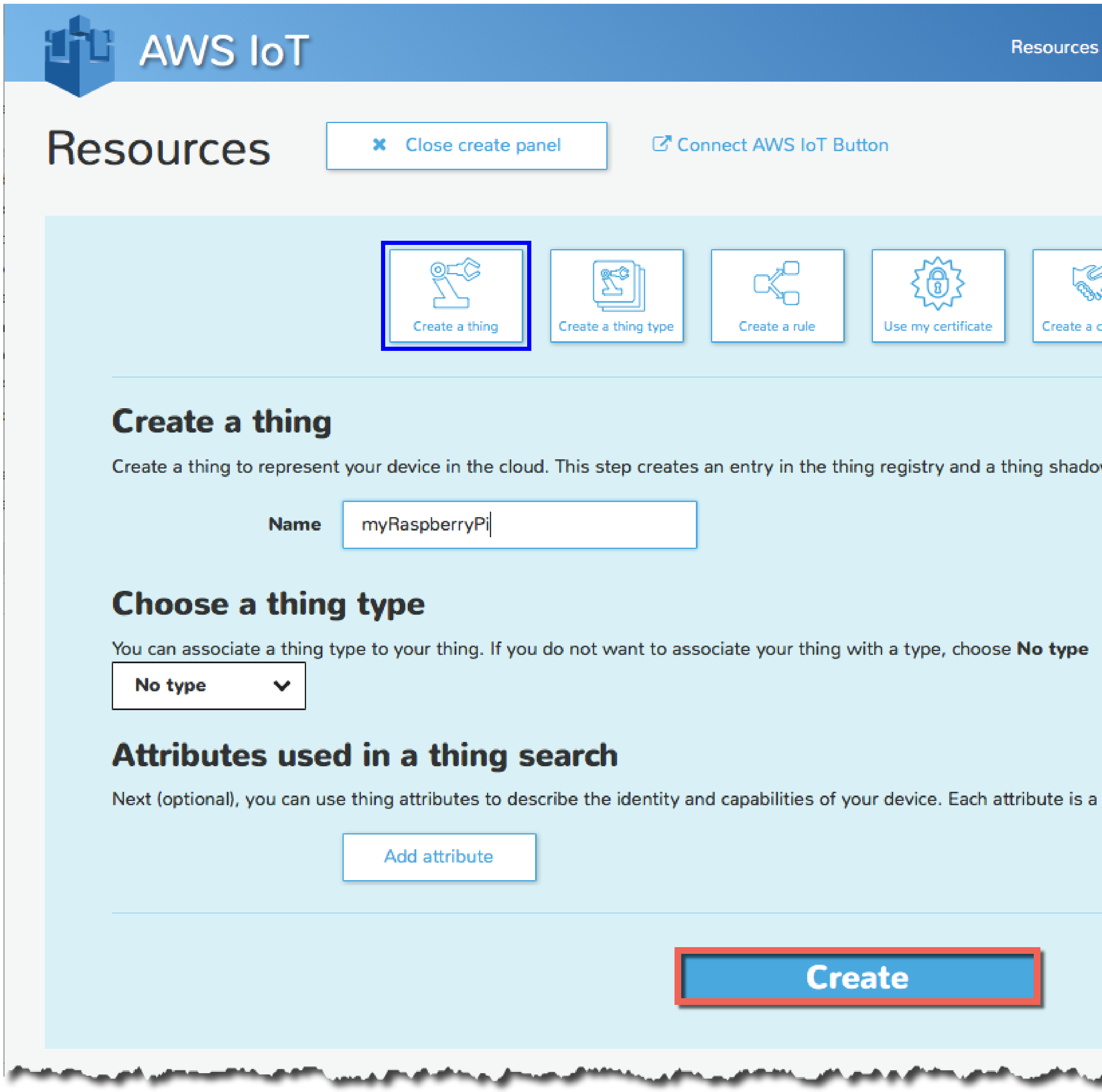
If you don't see a blue banner with **Create a thing**, **Create a rule**, **Create a certificate**, and **Create a policy** buttons, choose the **Create a resource** button:



Create and Attach a Thing (Device)

A thing represents a device whose status or data is stored in the AWS IoT 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.

Choose **Create a thing**, type a name for the thing, and then choose **Create**:



Create a thing

Create a thing to represent your device in the cloud. This step creates an entry in the thing registry and a thing shado

Name myRaspberryPi

Choose a thing type

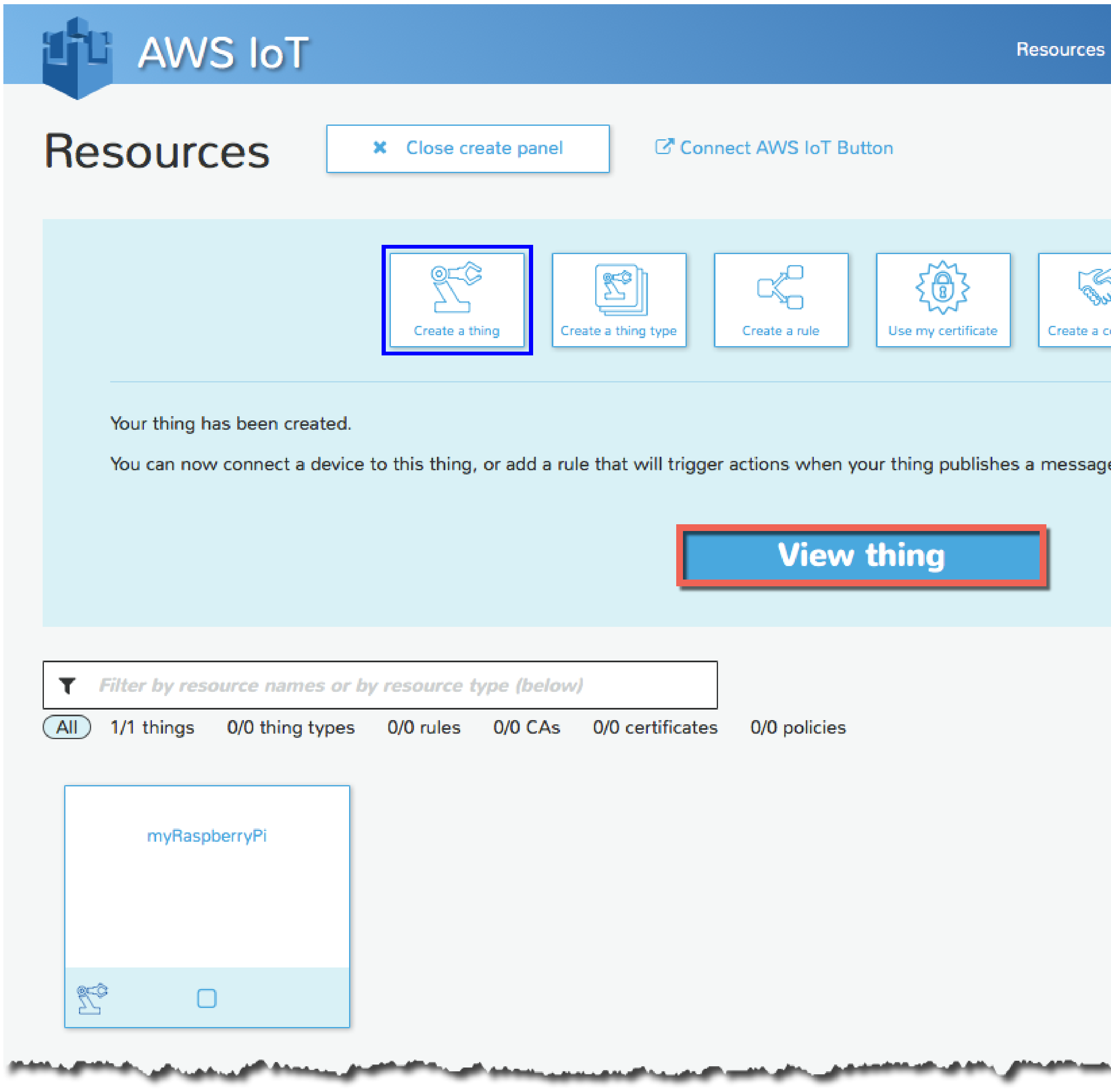
You can associate a thing type to your thing. If you do not want to associate your thing with a type, choose **No type**

No type

Attributes used in a thing search

Next (optional), you can use thing attributes to describe the identity and capabilities of your device. Each attribute is a

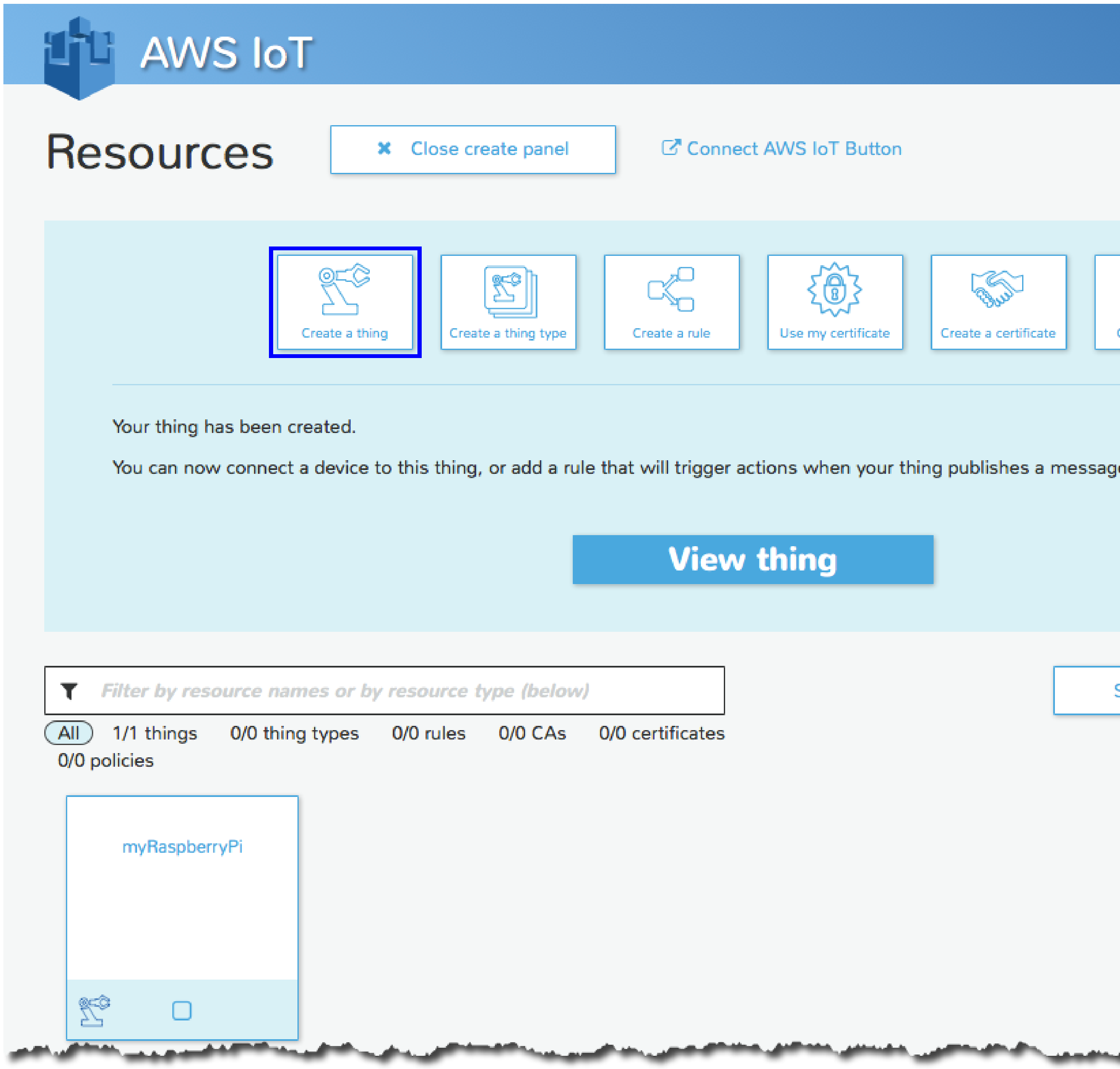
In addition to a confirmation message, the **View thing** button will be displayed:



Your thing has been created.

You can now connect a device to this thing, or add a rule that will trigger actions when your thing publishes a messag

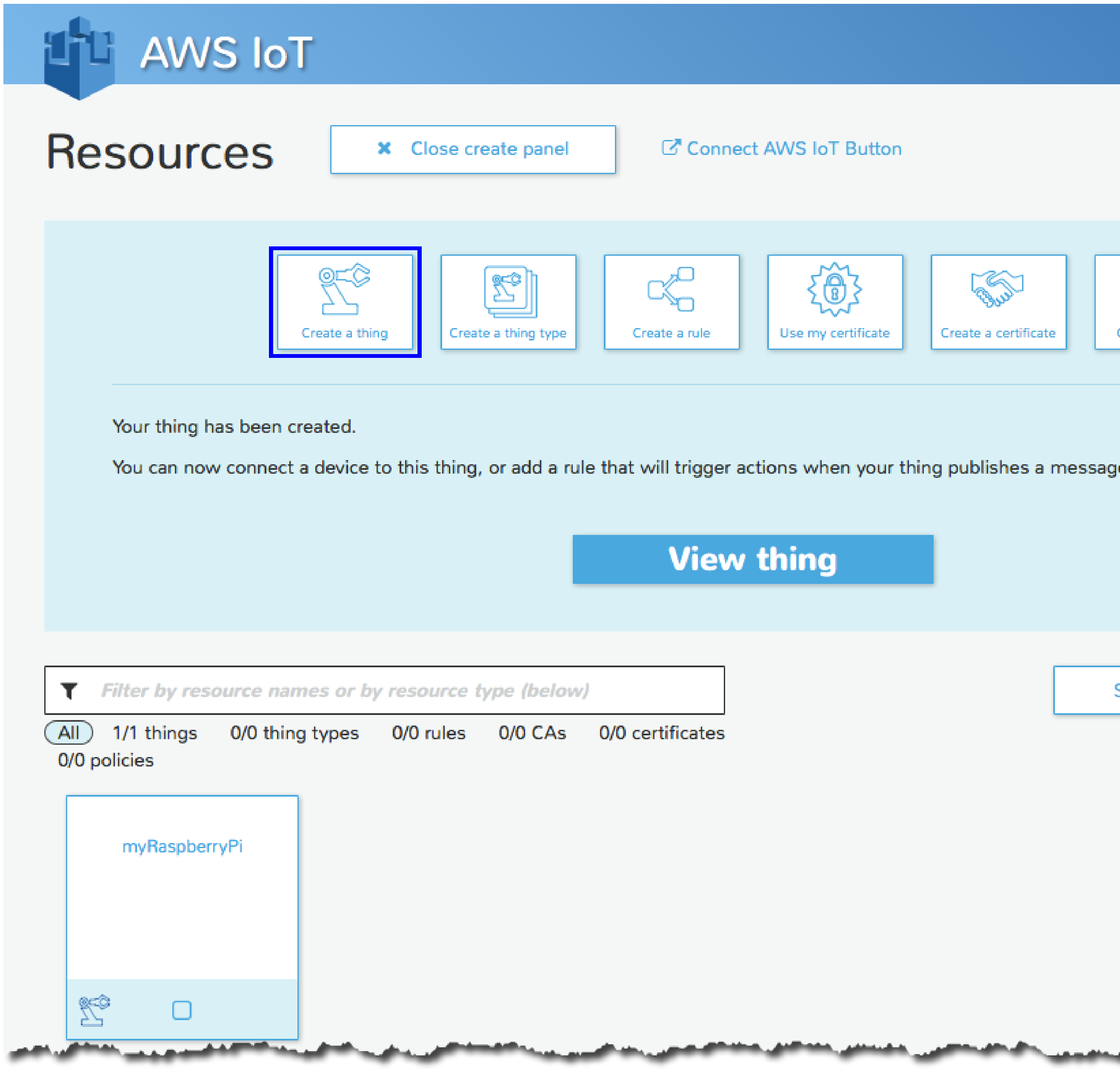
Choose **View thing** to display information about your thing:



Your thing has been created.

You can now connect a device to this thing, or add a rule that will trigger actions when your thing publishes a messag

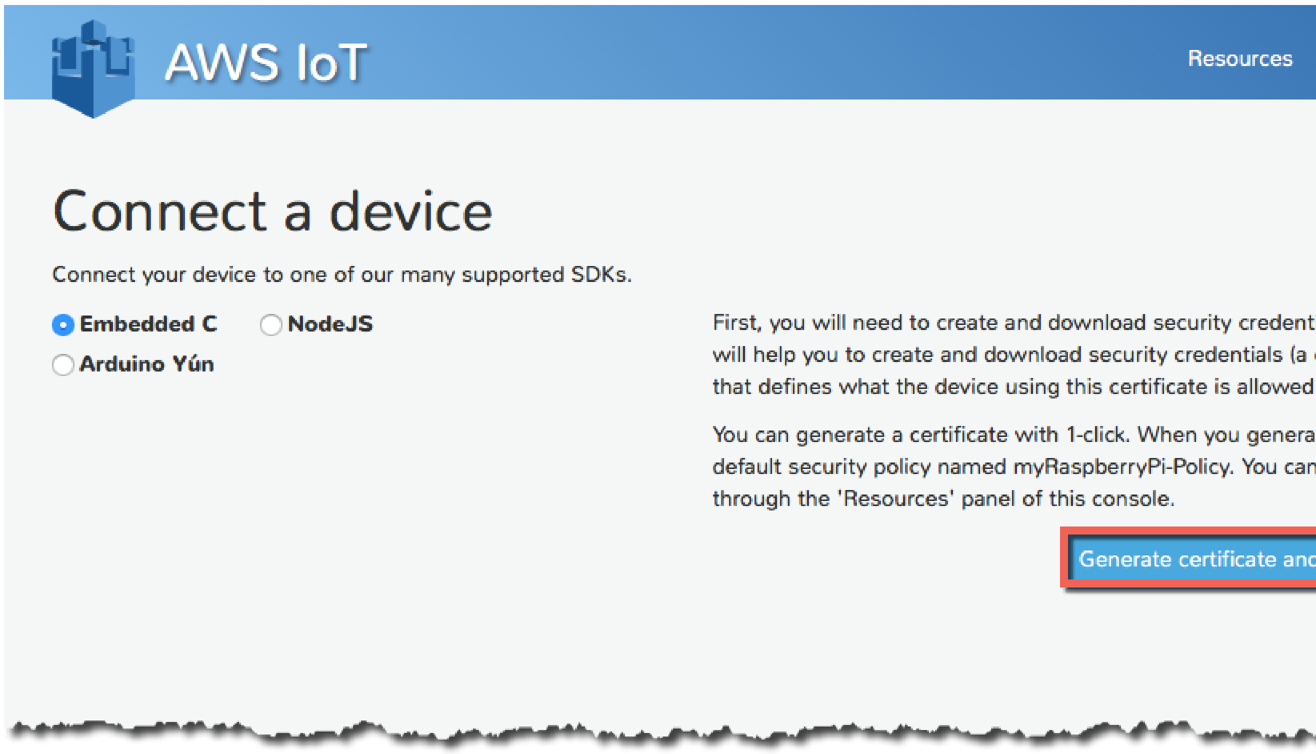
Choose the **Connect a device** button to download a key pair and a certificate generated by AWS IoT:



Your thing has been created.

You can now connect a device to this thing, or add a rule that will trigger actions when your thing publishes a messag

On the **Connect a device** page, select the SDK to use, and then choose **Generate certificate and policy**:



AWS IoT

Resources

Connect a device

Connect your device to one of our many supported SDKs.

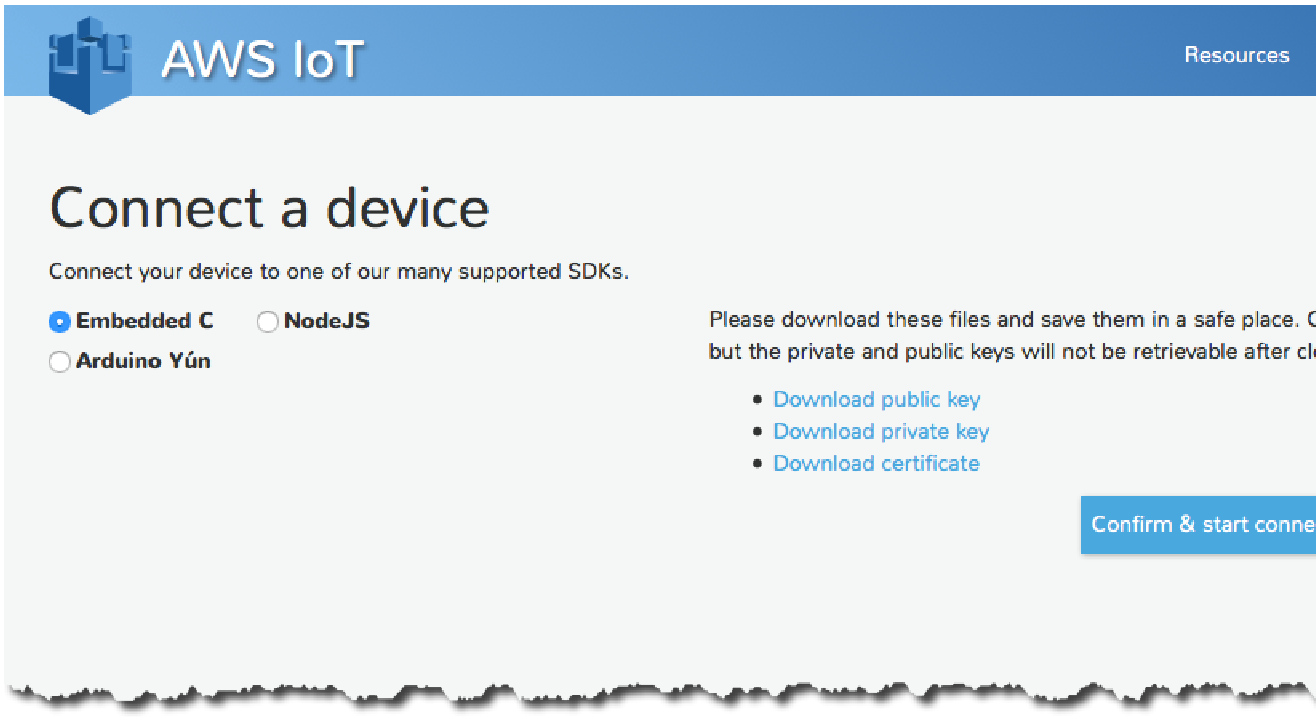
[✓ Embedded C] NodeJS Arduino Yún

First, you will need to create and download security credent
will help you to create and download security credentials (a
that defines what the device using this certificate is allowed

You can generate a certificate with 1-click. When you genera
default security policy named myRaspberryPi-Policy. You can
through the 'Resources' panel of this console.

This will generate an X.509 certificate and key pair; activate the X.509 certificate; and create an AWS IoT policy and attach it to the certificate.

The following page will be displayed:



AWS IoT

Resources

Connect a device

Connect your device to one of our many supported SDKs.

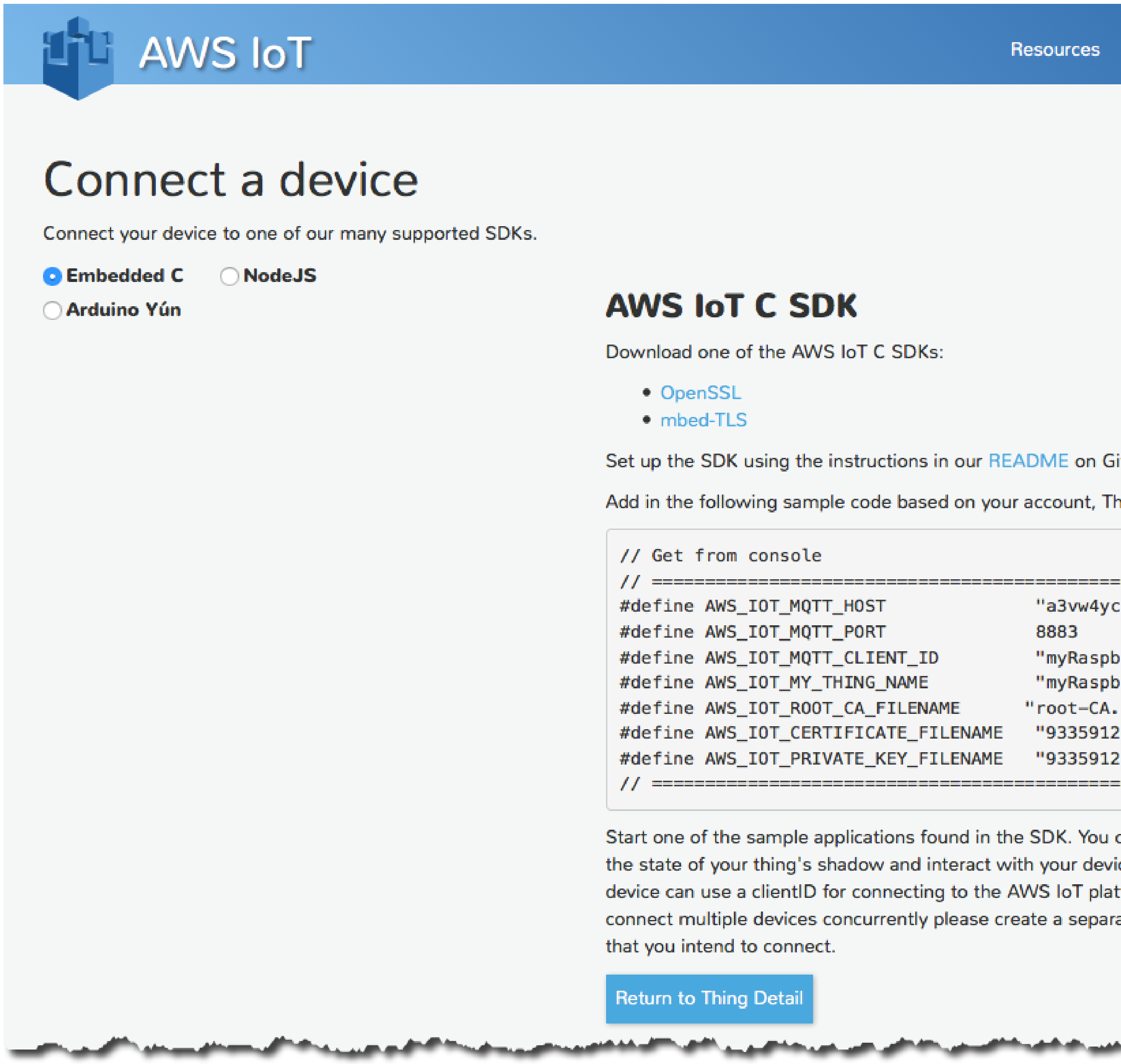
[✓ Embedded C] NodeJS Arduino Yún

Please download these files and save them in a safe place. C
but the private and public keys will not be retrievable after cl

- Download public key
- Download private key
- Download certificate

Create a working directory called `deviceSDK` where your files will be stored. Choose the links to download your public and private keys and certificate and save them in the `deviceSDK` directory.

Choose **Confirm & start connecting**. The following page will be displayed:



Connect a device

Connect your device to one of our many supported SDKs.

Embedded C [✓] NodeJS
Arduino Yún

AWS IoT C SDK

Download one of the AWS IoT C SDKs:

- OpenSSL
- mbed-TLS

Set up the SDK using the instructions in our README on Gi

Add in the following sample code based on your account, Th

```
// Get from console
// ============================================
#define AWS_IOT_MQTT_HOST              "a3vw4yc
#define AWS_IOT_MQTT_PORT              8883
#define AWS_IOT_MQTT_CLIENT_ID         "myRaspb
#define AWS_IOT_MY_THING_NAME          "myRaspb
#define AWS_IOT_ROOT_CA_FILENAME      "root-CA.
#define AWS_IOT_CERTIFICATE_FILENAME   "9335912
#define AWS_IOT_PRIVATE_KEY_FILENAME   "9335912
// ============================================
```

Start one of the sample applications found in the SDK. You c
the state of your thing's shadow and interact with your devic
device can use a clientID for connecting to the AWS IoT plat
connect multiple devices concurrently please create a separa
that you intend to connect.

Return to Thing Detail

There are two versions of the AWS IoT Embedded C SDK: OpenSSL and mbed TLS. Choose the **OpenSSL** link. This will download the AWS IoT AWS IoT Device SDK for C in a tarball (`linux_mqtt_openssl-latest.tar`). Save it in your `deviceSDK` directory. In a terminal window, type the following command to extract the tarball into your `deviceSDK` directory:

```
`tar -xvf linux_mqtt_openssl-latest.tar`
```

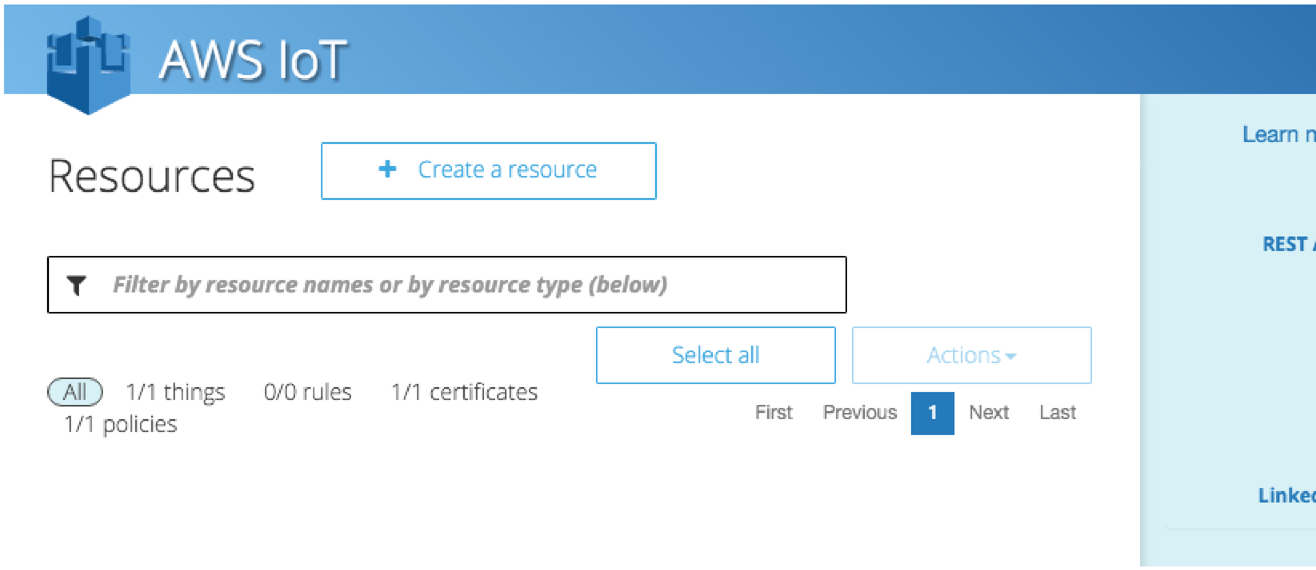
Set Up the Runtime Environment for the AWS IoT Embedded C SDK

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. Copy your certificate and private key into the `deviceSDK/certs` directory. Download a root CA certificate here. Copy the root CA text from the browser, paste it into a file, and then copy it into the `deviceSDK/certs` directory.

Navigate to the `deviceSDK/sample_apps/subscribe_publish_sample` directory. You will need to configure your personal endpoint, private key, and certificate. If 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. Otherwise, go to the AWS IoT console, double-click **MyNewThing**, and copy everything after "https://" including ".com" from **REST API endpoint**.



Open the `aws_iot_config.h` file and 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.

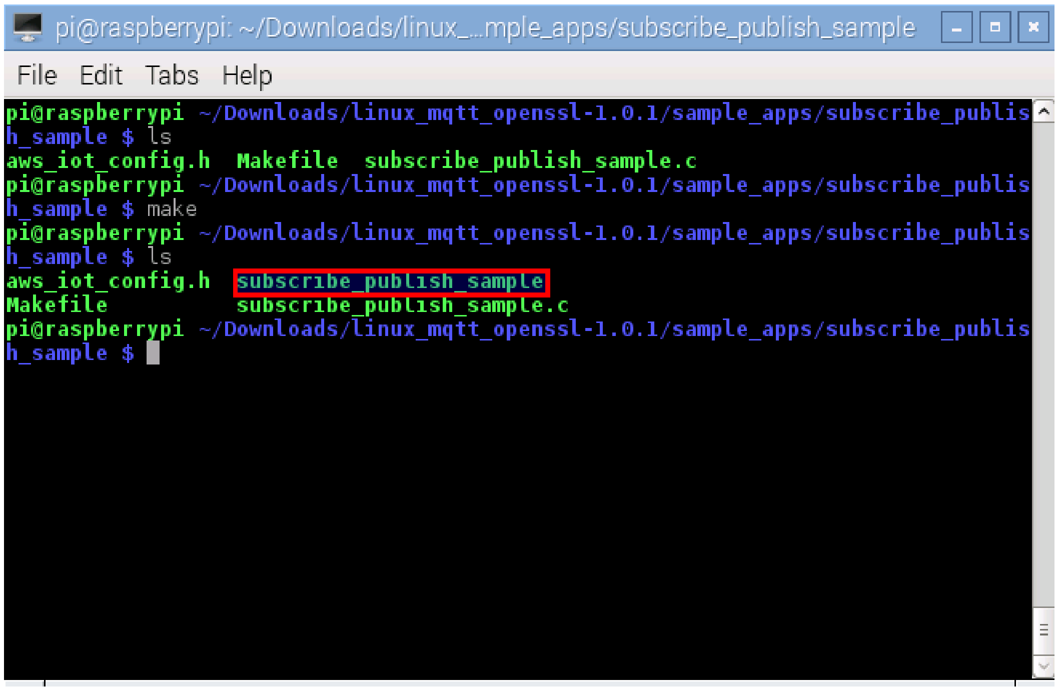
AWS_IOT_PRIVATE_KEY_FILENAME
: Your private key.

Run Sample Applications

Compile the `subscribe_publish_sample_app` using the included makefile.

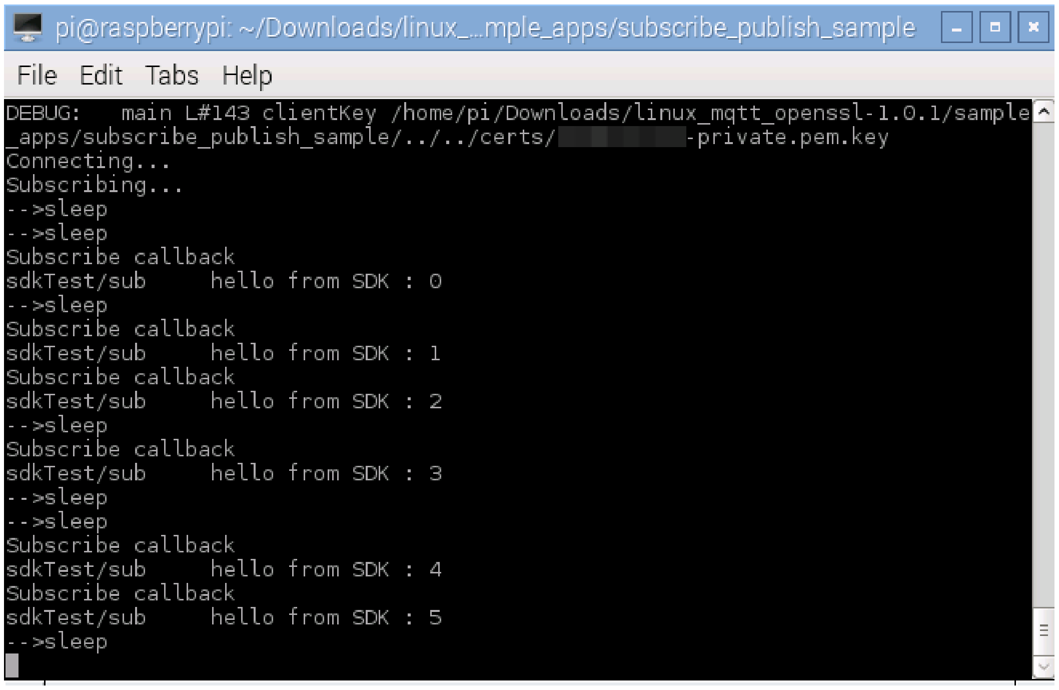
```
make -f Makefile
```

This will generate an executable file.


pi@raspberrypi: ~/Downloads/linux_...mple_apps/subscribe_publish_sample
File Edit Tabs Help

```
pi@raspberrypi ~/Downloads/linux_mqtt_openssl-1.0.1/sample_apps/subscribe_publis
h_sample $ ls
aws_iot_config.h  Makefile  subscribe_publish_sample.c
pi@raspberrypi ~/Downloads/linux_mqtt_openssl-1.0.1/sample_apps/subscribe_publis
h_sample $ make
pi@raspberrypi ~/Downloads/linux_mqtt_openssl-1.0.1/sample_apps/subscribe_publis
h_sample $ ls
aws_iot_config.h  subscribe_publish_sample
Makefile          subscribe_publish_sample.c
pi@raspberrypi ~/Downloads/linux_mqtt_openssl-1.0.1/sample_apps/subscribe_publis
h_sample $
```

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


pi@raspberrypi: ~/Downloads/linux_...mple_apps/subscribe_publish_sample
File Edit Tabs Help

```
DEBUG:   main L#143 clientKey /home/pi/Downloads/linux_mqtt_openssl-1.0.1/sample
_apps/subscribe_publish_sample/../../certs/[redacted]-private.pem.key
Connecting...
Subscribing...
-->sleep
-->sleep
Subscribe callback
sdkTest/sub     hello from SDK : 0
-->sleep
Subscribe callback
sdkTest/sub     hello from SDK : 1
Subscribe callback
sdkTest/sub     hello from SDK : 2
-->sleep
Subscribe callback
sdkTest/sub     hello from SDK : 3
-->sleep
-->sleep
Subscribe callback
sdkTest/sub     hello from SDK : 4
Subscribe callback
sdkTest/sub     hello from SDK : 5
-->sleep
```

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

Getting Started with AWS IoT on Raspberry Pi and the AWS IoT Device SDK for JavaScript

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 Device SDK for JavaScript. After following the steps in this guide, you will be able to get connected to the AWS IoT platform and run sample apps included in the SDK.

Prerequisites

- A fully set up Raspberry Pi board with Internet access

  For information about setting up your Raspberry Pi, see the 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:

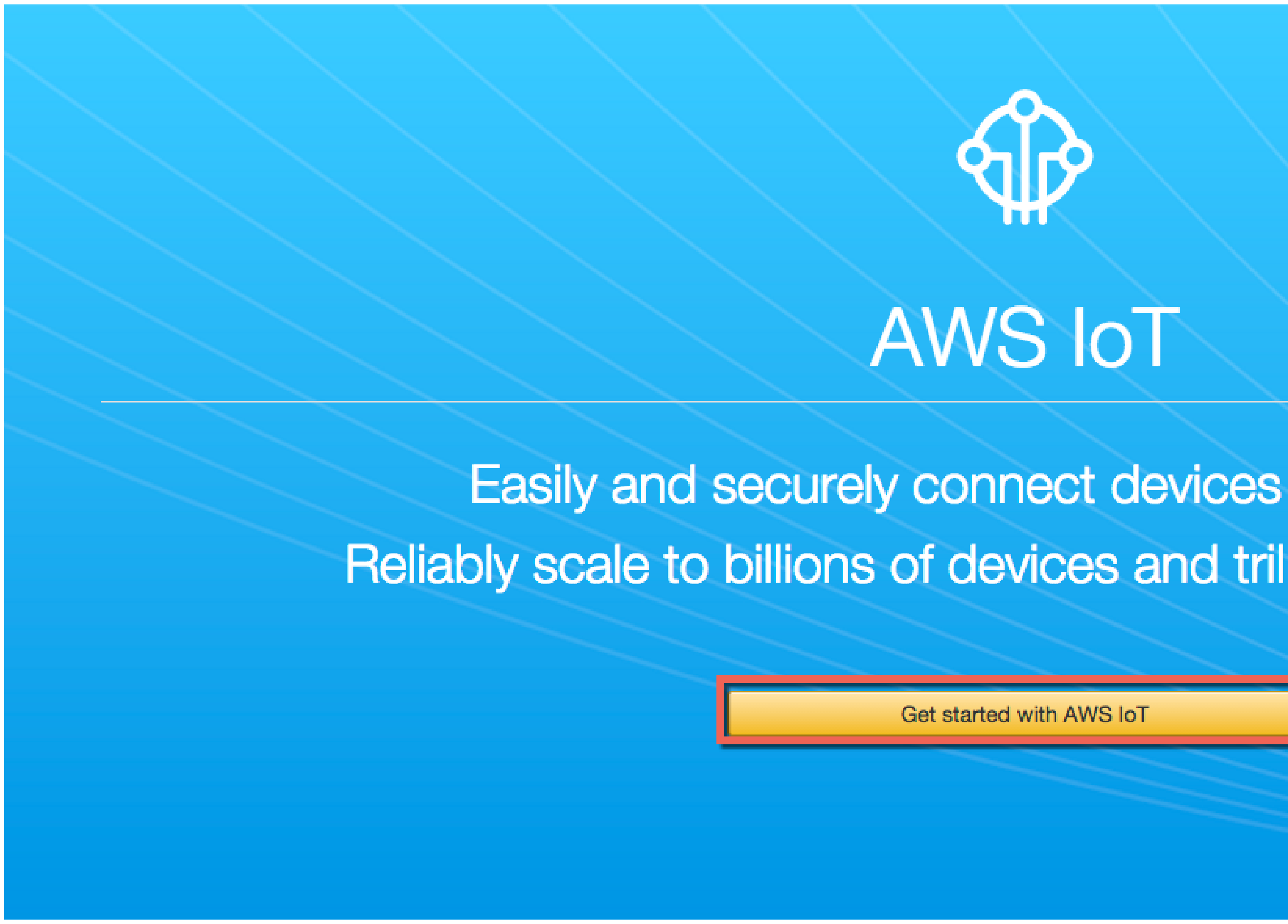
- Raspberry Pi 2 Model B
- Raspbian Jessie
- Iceweasel browser

Connecting Your Raspberry Pi

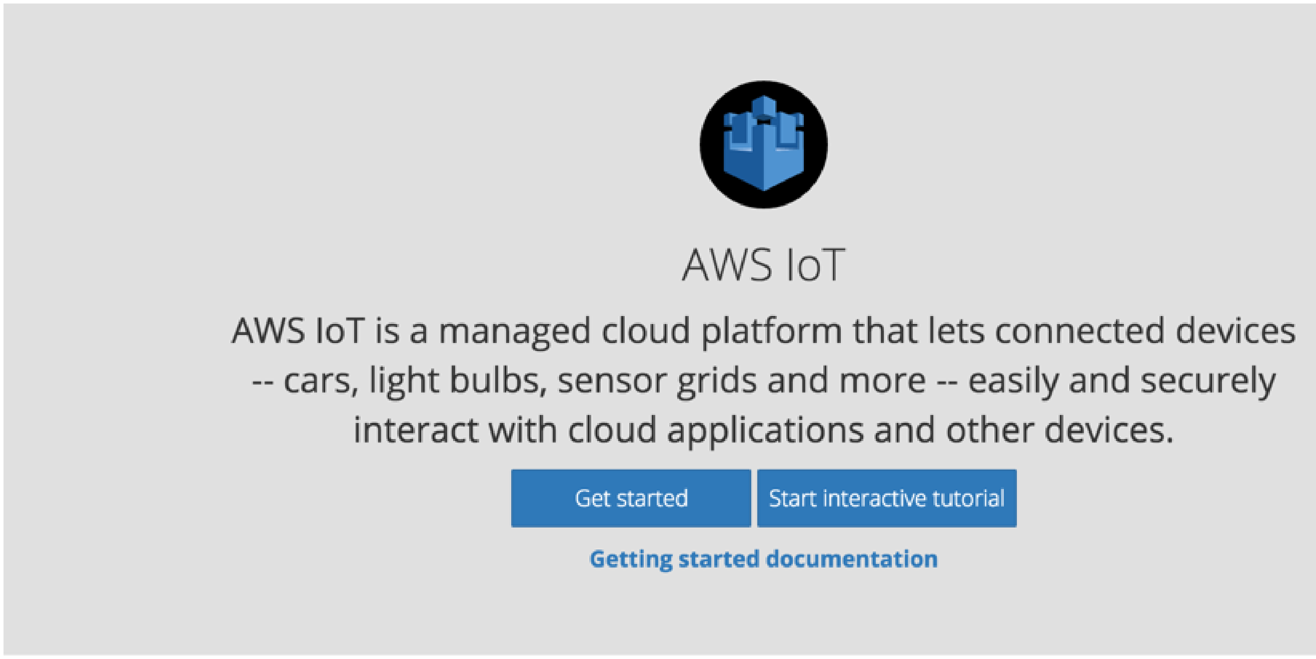
Sign in to the AWS IoT Console

Turn on your Raspberry Pi and confirm you have an Internet connection.

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 with AWS IoT**:

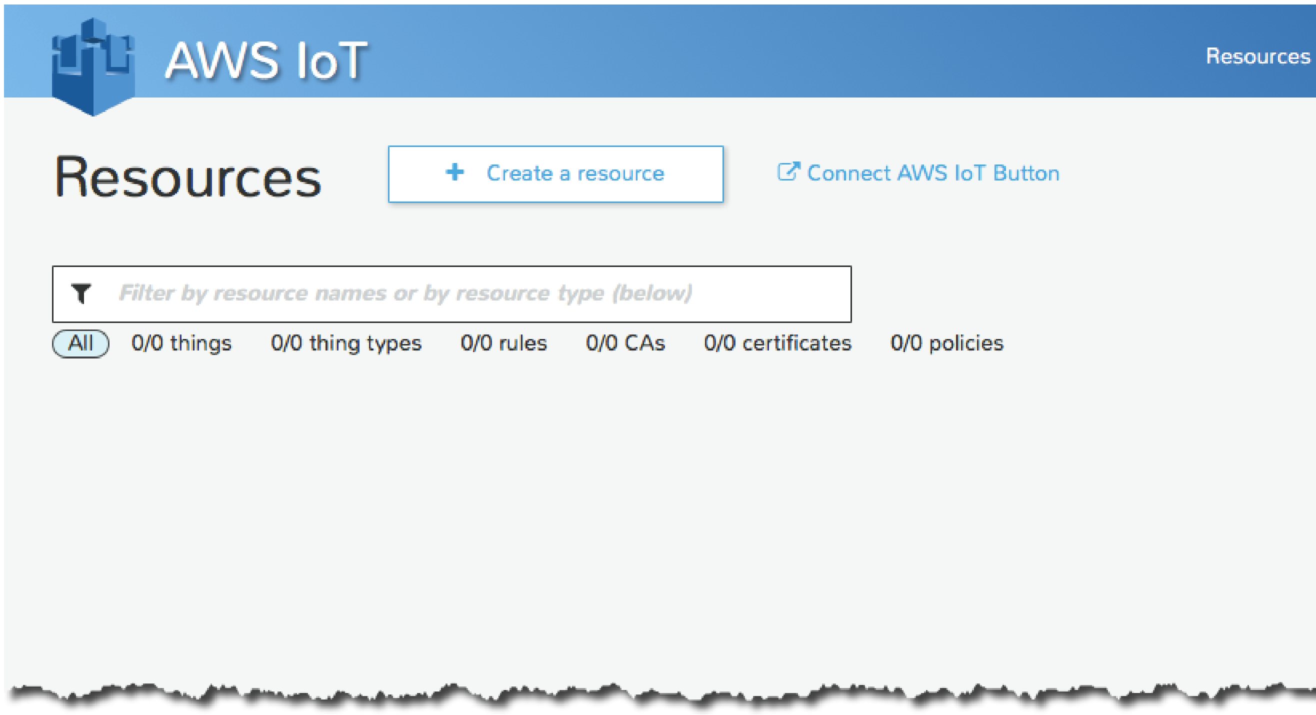

AWS IoT
Easily and securely connect devices
Reliably scale to billions of devices and tril
Get started with AWS IoT

If this is your first time using the AWS IoT console, you will see two buttons: **Get Started** and **Start Interactive Tutorial**.

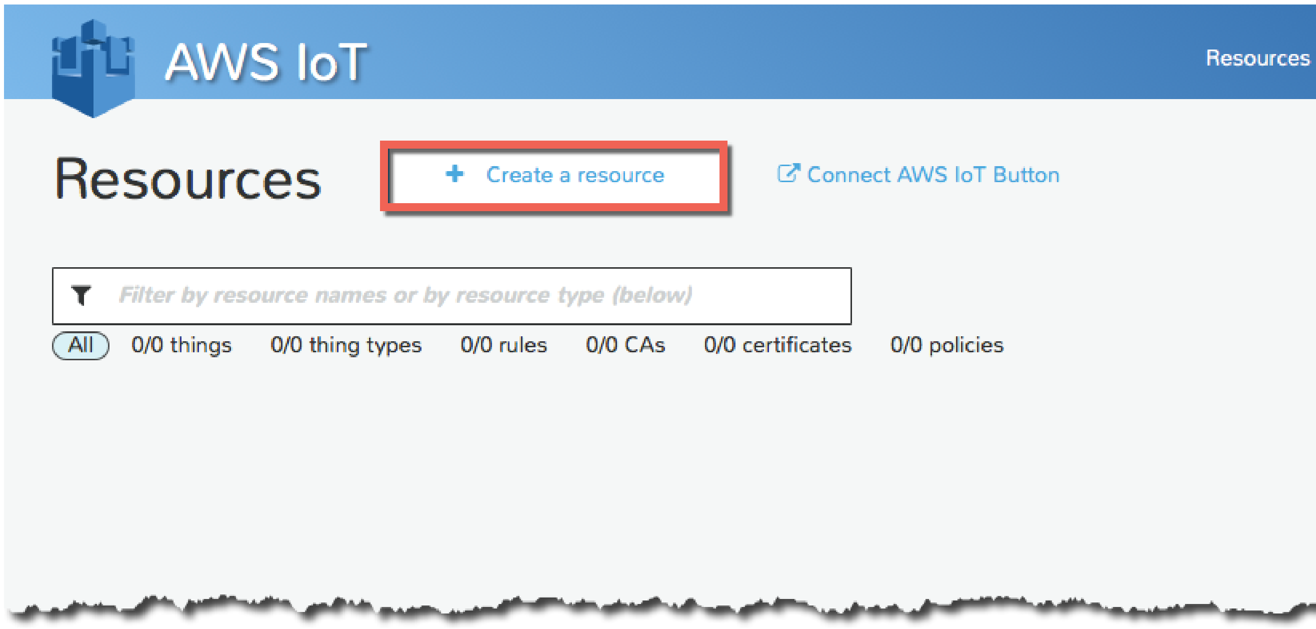


AWS IoT is a managed cloud platform that lets connected devices -- cars, light bulbs, sensor grids and more -- easily and securely interact with cloud applications and other devices.

Choose **Get Started**. The following page should appear.



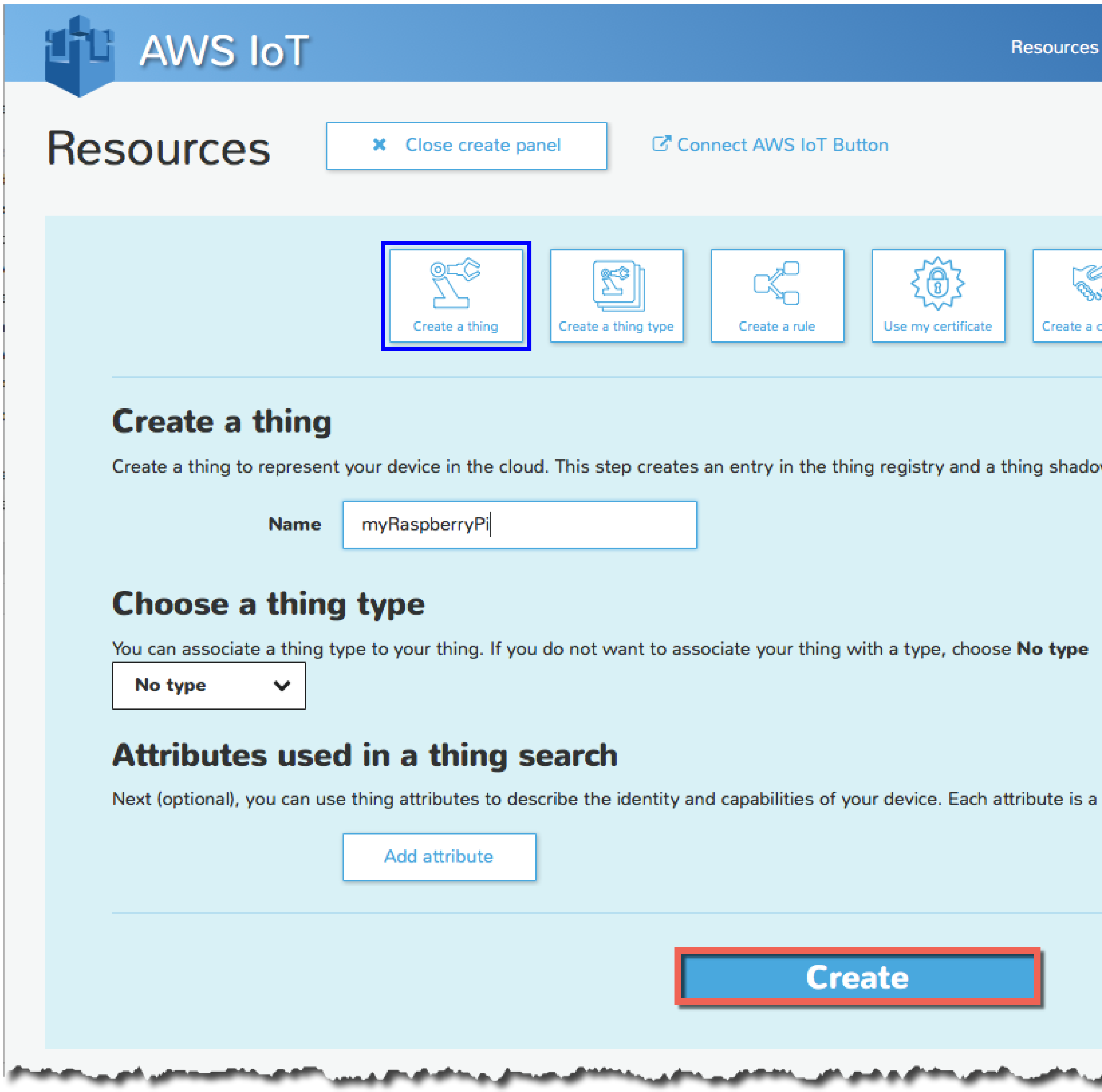
If you don't see a blue banner with **Create a thing**, **Create a rule**, **Create a certificate**, and **Create a policy** buttons, choose the **Create a resource** button:



Create and Attach a Thing (Device)

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

Choose **Create a thing**, type in a name for the thing, and then choose **Create**:



Create a thing

Create a thing to represent your device in the cloud. This step creates an entry in the thing registry and a thing shado

Name: myRaspberryPi

Choose a thing type

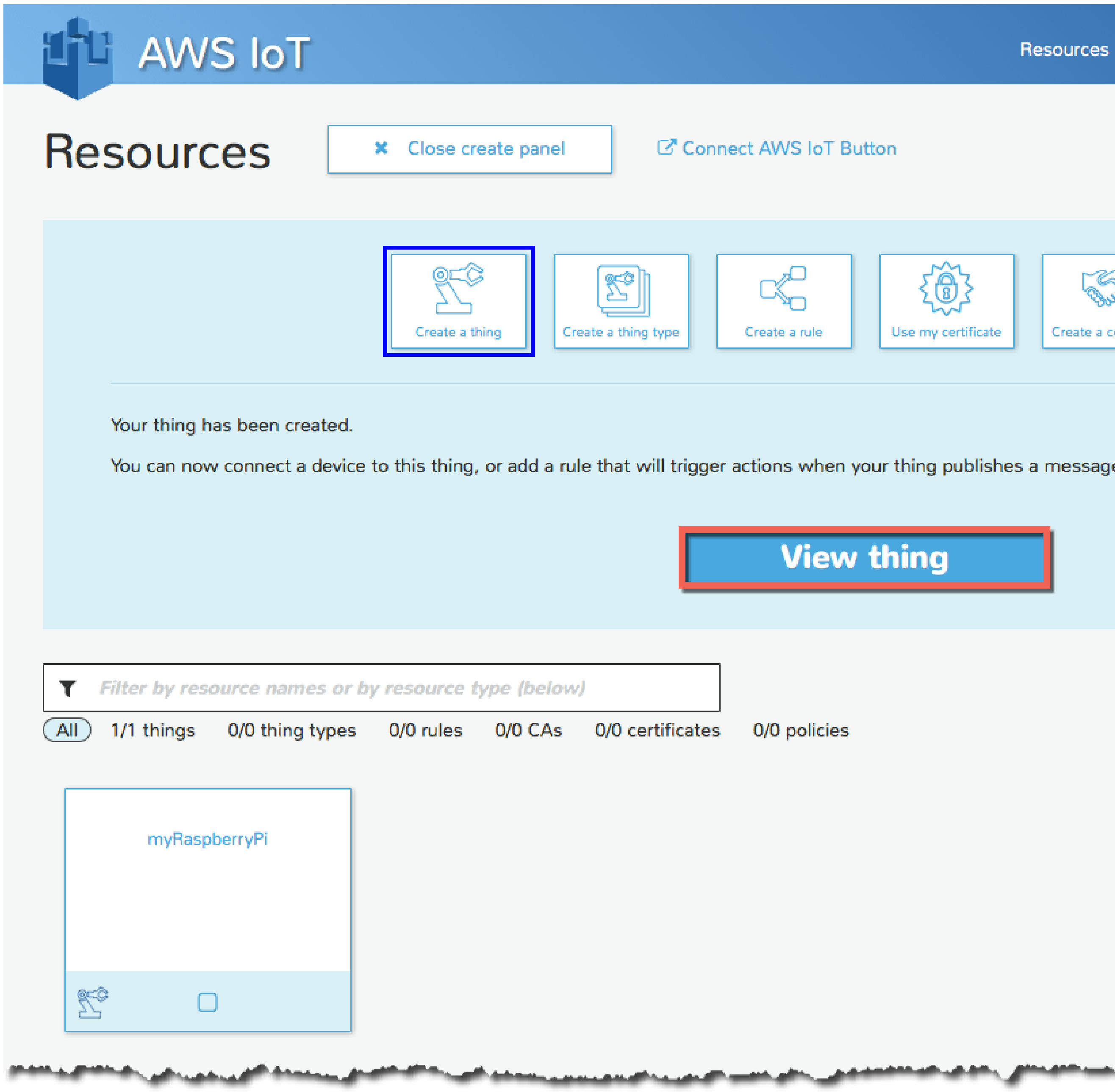
You can associate a thing type to your thing. If you do not want to associate your thing with a type, choose **No type**

No type

Attributes used in a thing search

Next (optional), you can use thing attributes to describe the identity and capabilities of your device. Each attribute is a

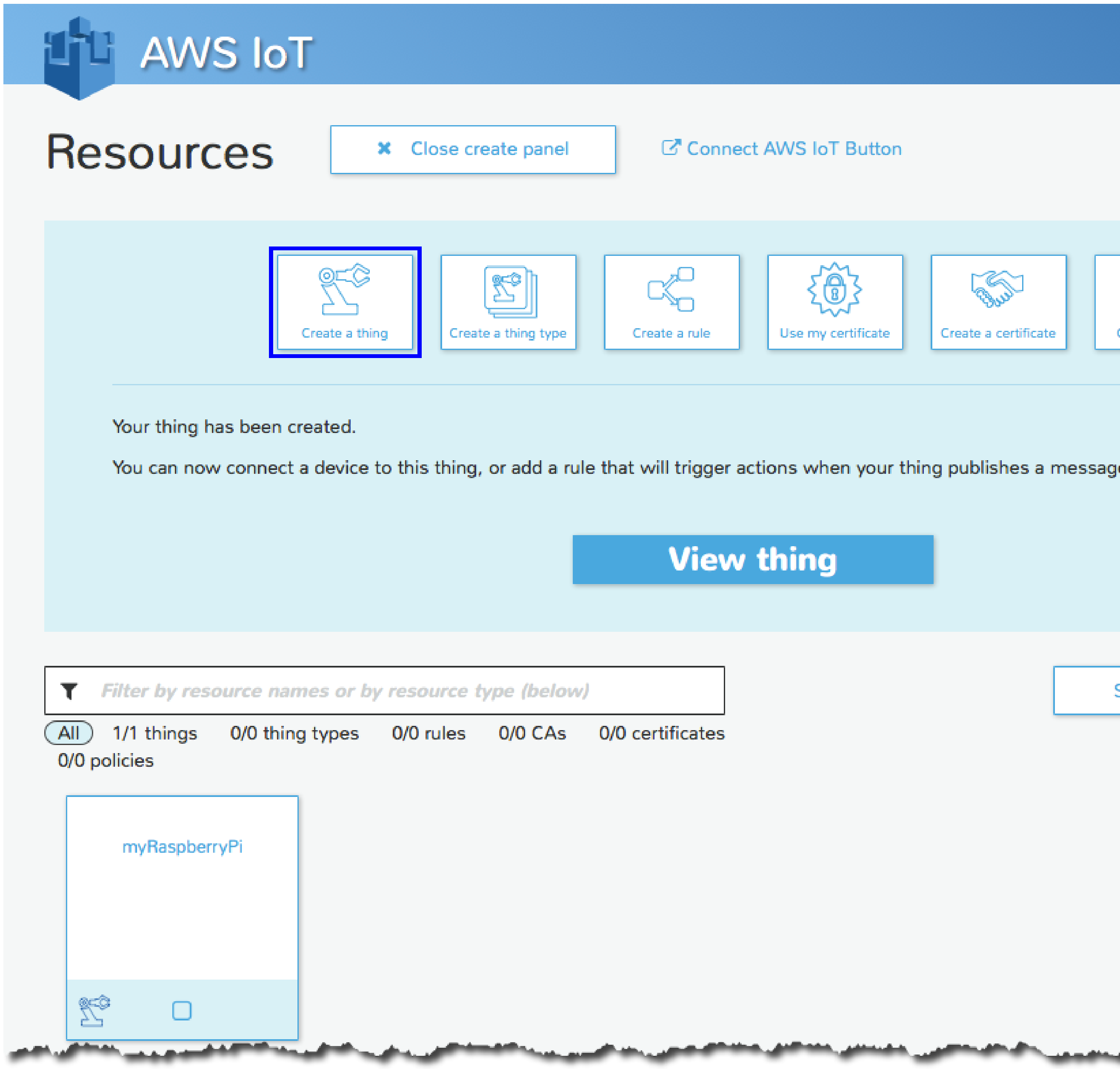
In addition to a confirmation message, the **View thing** button will be displayed:



Your thing has been created.

You can now connect a device to this thing, or add a rule that will trigger actions when your thing publishes a messag

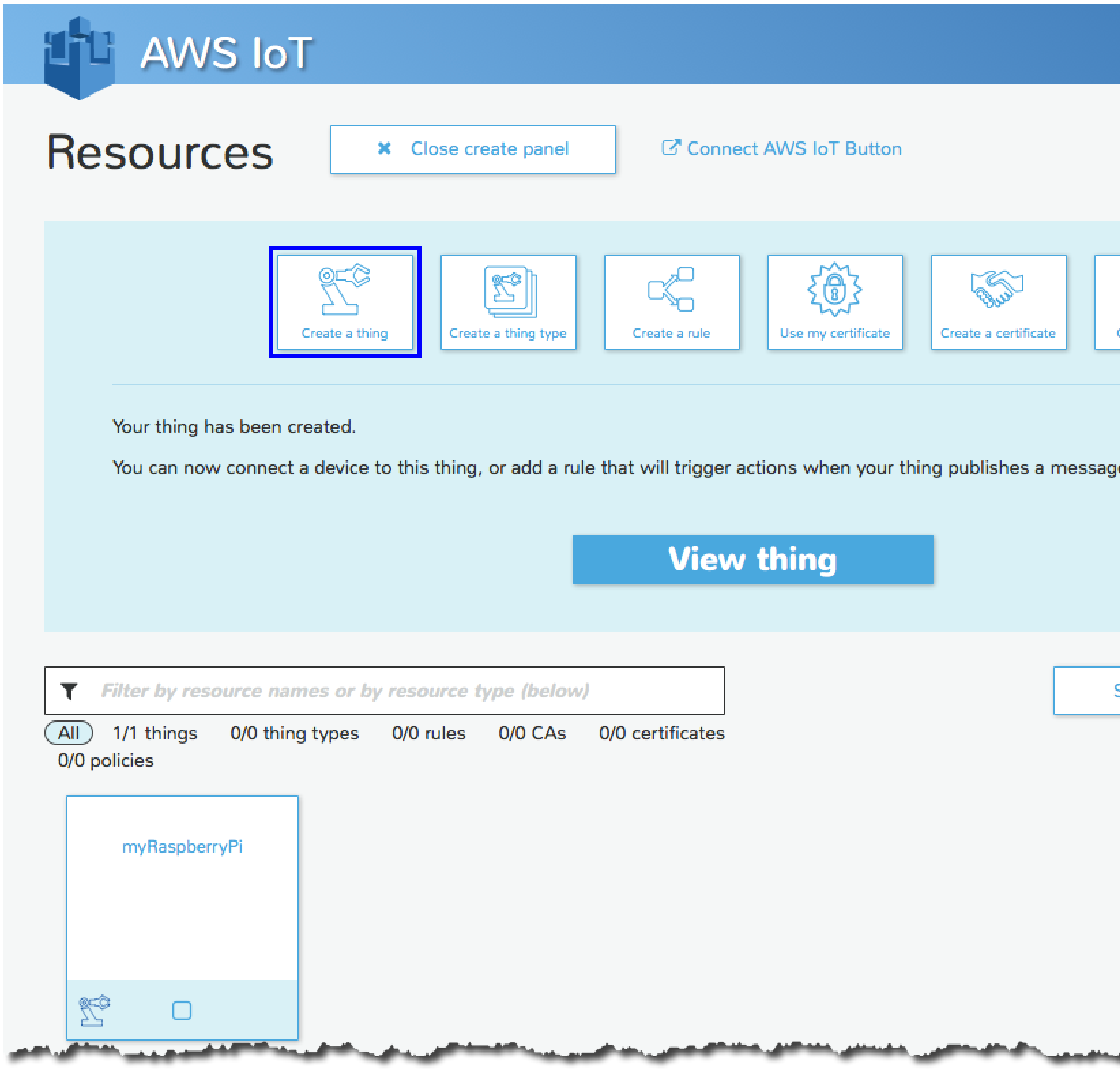
Choose **View thing** to display information about your thing:



Your thing has been created.

You can now connect a device to this thing, or add a rule that will trigger actions when your thing publishes a messag

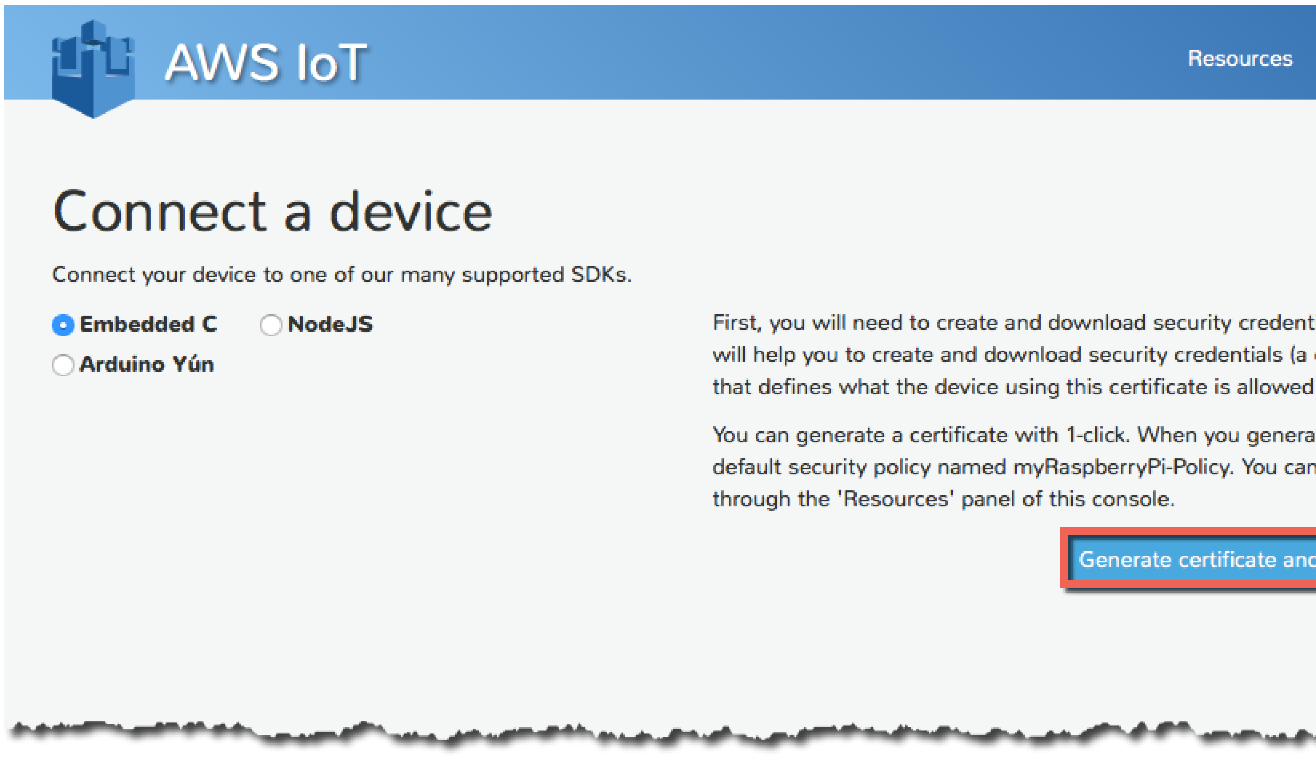
Choose the **Connect a device** button to download a key pair and a certificate generated by AWS IoT :



Your thing has been created.

You can now connect a device to this thing, or add a rule that will trigger actions when your thing publishes a messag

On the **Connect a device** page, select the SDK to use, and then choose**Generate certificate and policy**:



AWS IoT

Resources

Connect a device

Connect your device to one of our many supported SDKs.

[✓ Embedded C] NodeJS

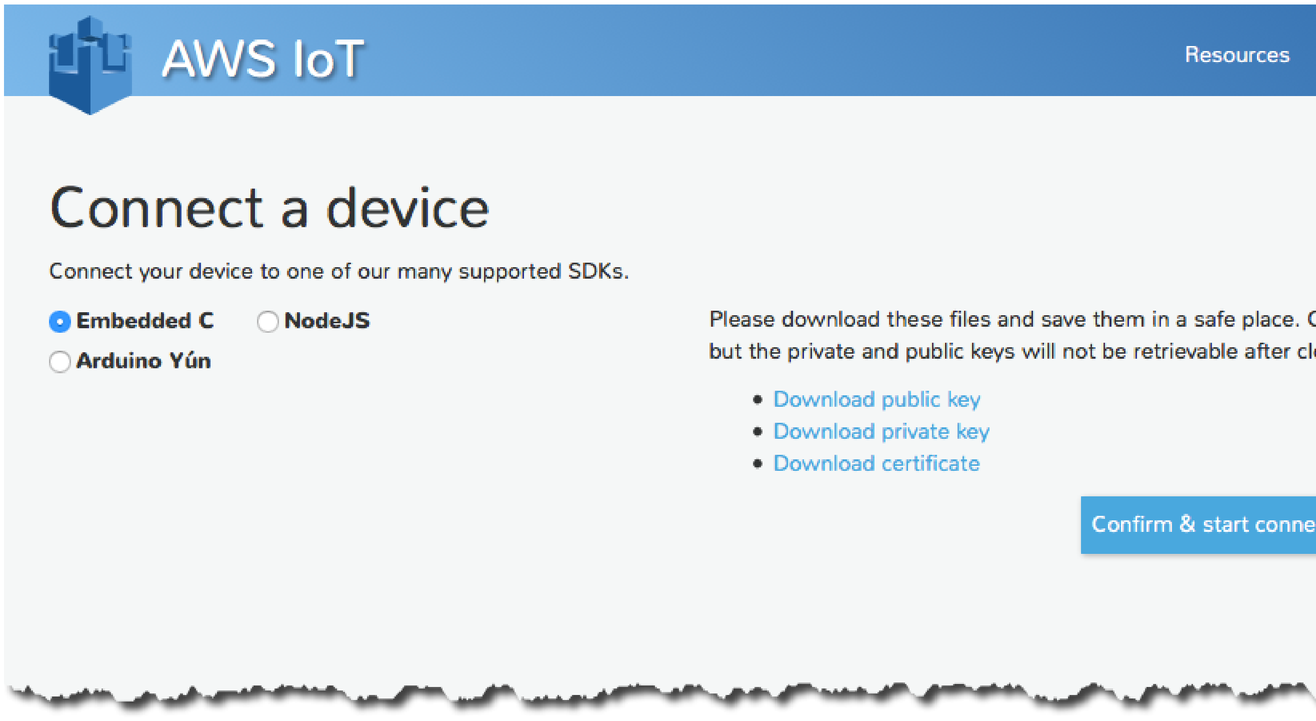
Arduino Yún

First, you will need to create and download security credent
will help you to create and download security credentials (a
that defines what the device using this certificate is allowed

You can generate a certificate with 1-click. When you genera
default security policy named myRaspberryPi-Policy. You can
through the 'Resources' panel of this console.

This will generate an X.509 certificate and key pair; activate the X.509 certificate; and create an AWS IoT policy and attach it to the certificate.

The following page will be displayed:



AWS IoT

Resources

Connect a device

Connect your device to one of our many supported SDKs.

[✓ Embedded C] NodeJS

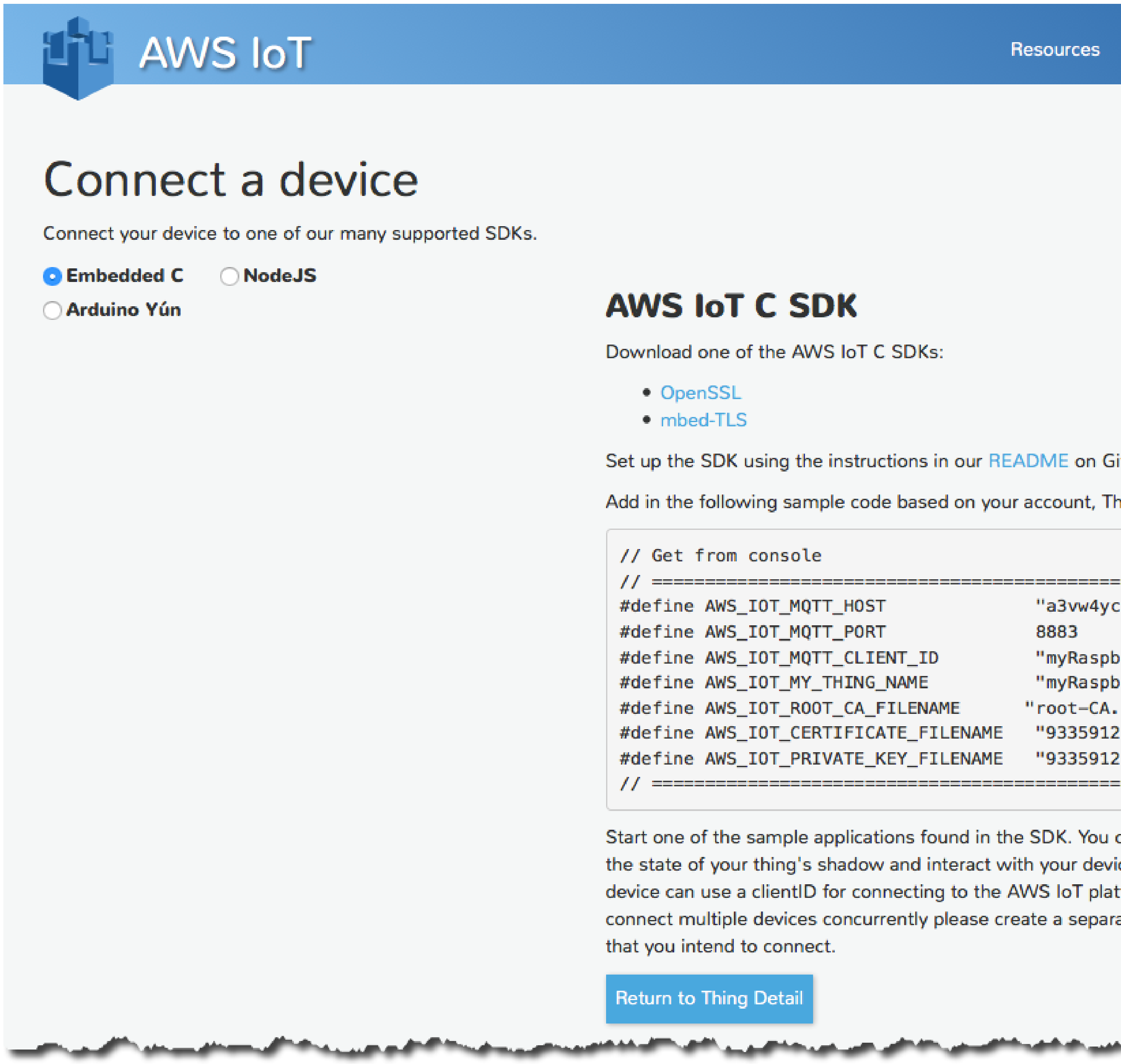
Arduino Yún

Please download these files and save them in a safe place. C
but the private and public keys will not be retrievable after cl

- Download public key
- Download private key
- Download certificate

Create a working directory called `deviceSDK` where your files will be stored. Choose the links to download your public and private keys and certificate, and then save them in the `deviceSDK` directory.

Choose **Confirm & start connecting**. The following page will be displayed:



Connect a device

Connect your device to one of our many supported SDKs.

AWS IoT C SDK

Download one of the AWS IoT C SDKs:

- OpenSSL
- mbed-TLS

Set up the SDK using the instructions in our README on Gi

Add in the following sample code based on your account, Th

```
// Get from console
// =============================================
#define AWS_IOT_MQTT_HOST              "a3vw4yc
#define AWS_IOT_MQTT_PORT              8883
#define AWS_IOT_MQTT_CLIENT_ID         "myRaspb
#define AWS_IOT_MY_THING_NAME          "myRaspb
#define AWS_IOT_ROOT_CA_FILENAME      "root-CA.
#define AWS_IOT_CERTIFICATE_FILENAME   "9335912
#define AWS_IOT_PRIVATE_KEY_FILENAME   "9335912
// =============================================
```

Start one of the sample applications found in the SDK. You c
the state of your thing's shadow and interact with your devic
device can use a clientID for connecting to the AWS IoT plat
connect multiple devices concurrently please create a separa
that you intend to connect.

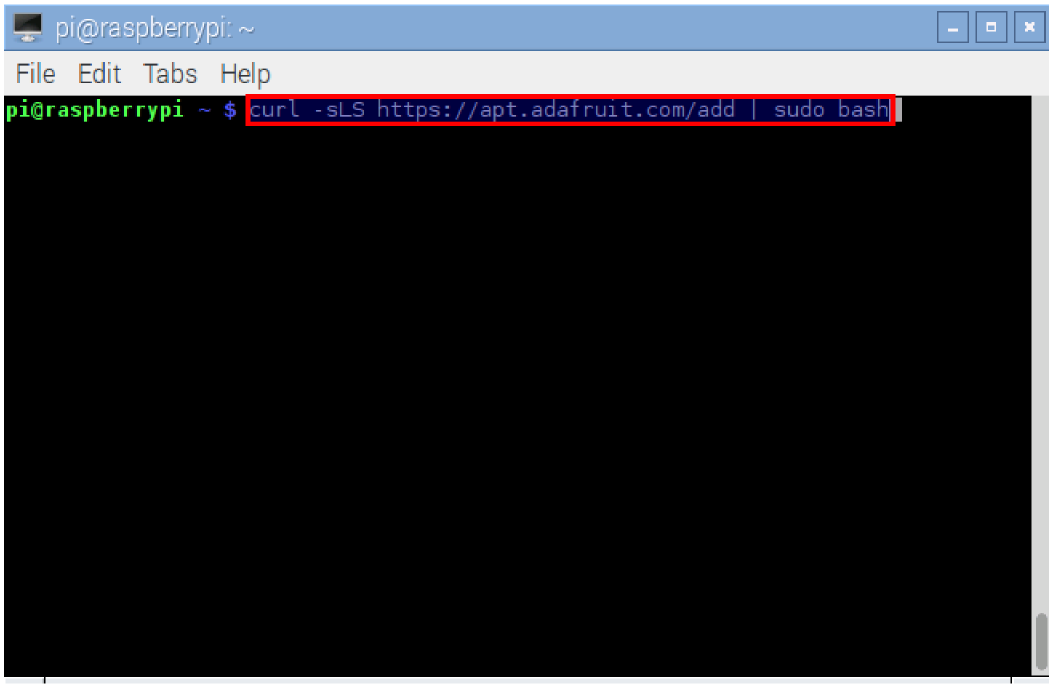
Set Up the Runtime Environment for the AWS IoT Device SDK for JavaScript

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.

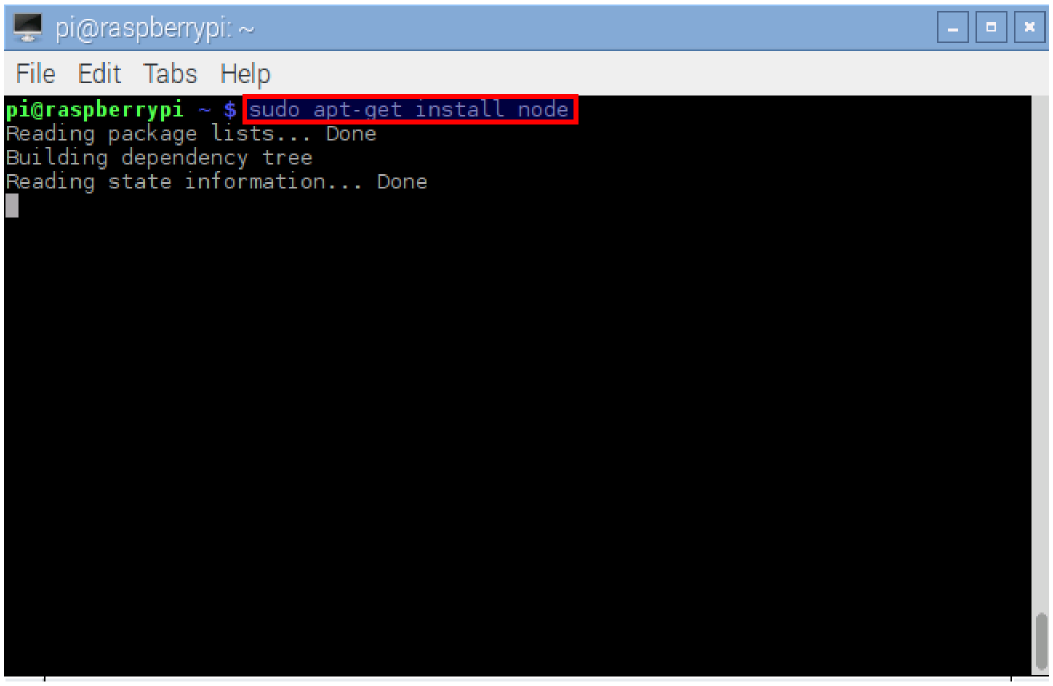
To add the Node repository, open a terminal and run the following command:

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



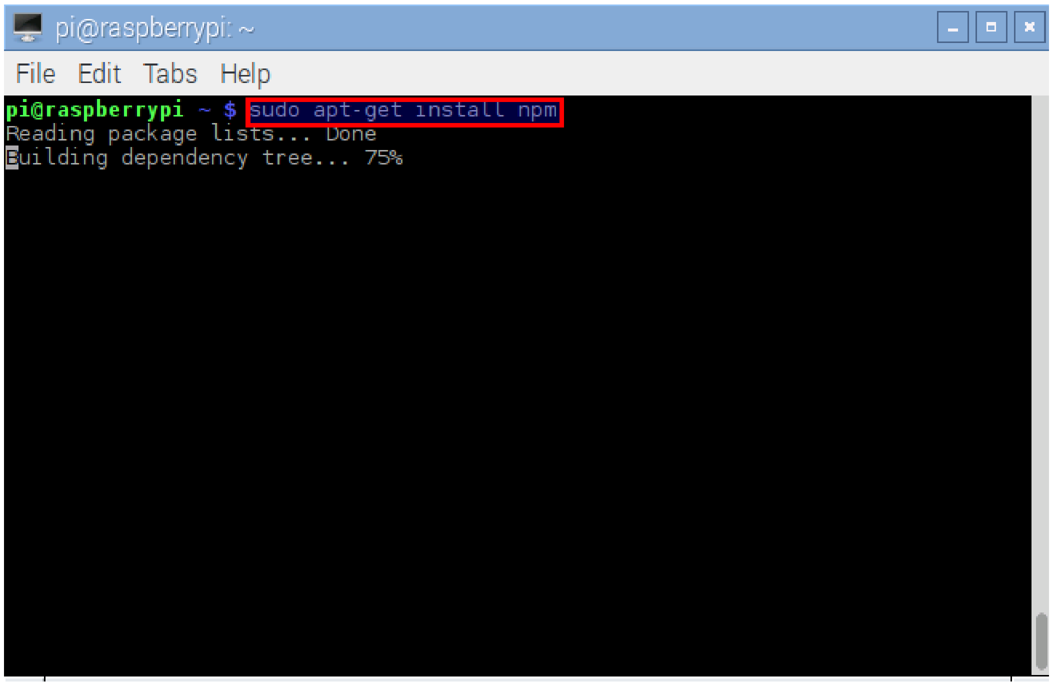
```
pi@raspberrypi ~ $ curl -sLS https://apt.adafruit.com/add | sudo bash
```

To install Node, run `sudo apt-get install node`. You should see output similar to the following:



```
pi@raspberrypi ~ $ sudo apt-get install node
Reading package lists... Done
Building dependency tree
Reading state information... Done
```

To install npm, run `sudo apt-get install npm`. You should see output similar to the following:

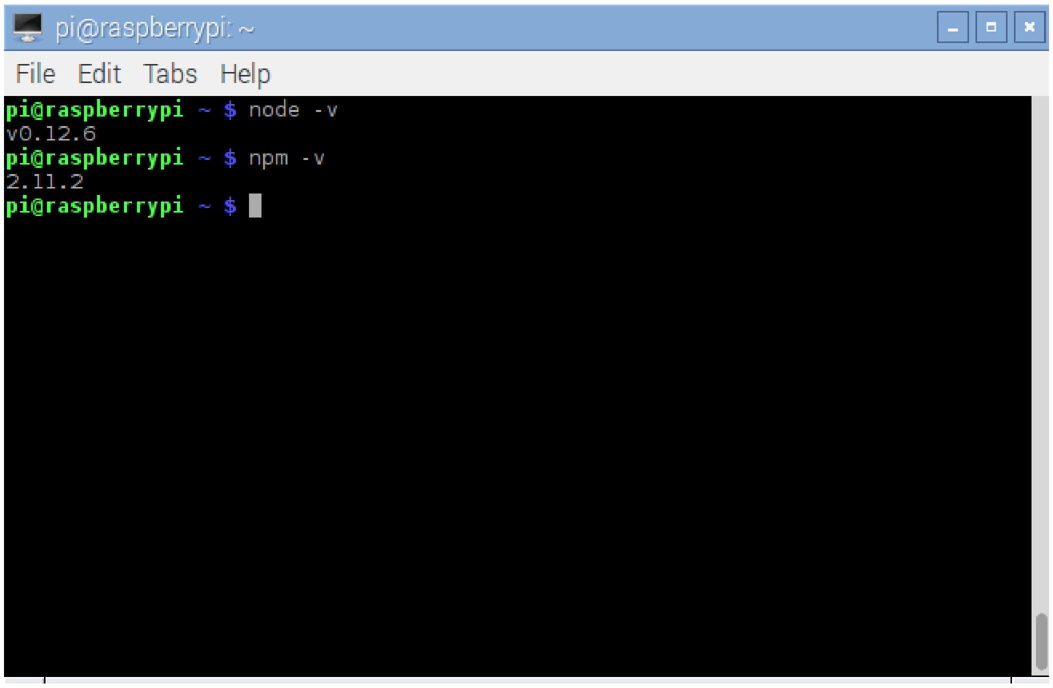


pi@raspberrypi: ~

File Edit Tabs Help

```
pi@raspberrypi ~ $ sudo apt-get install npm
Reading package lists... Done
Building dependency tree... 75%
```

To verify the installation of Node and npm, run `node -v` and `npm -v`. You should see output similar to the following:



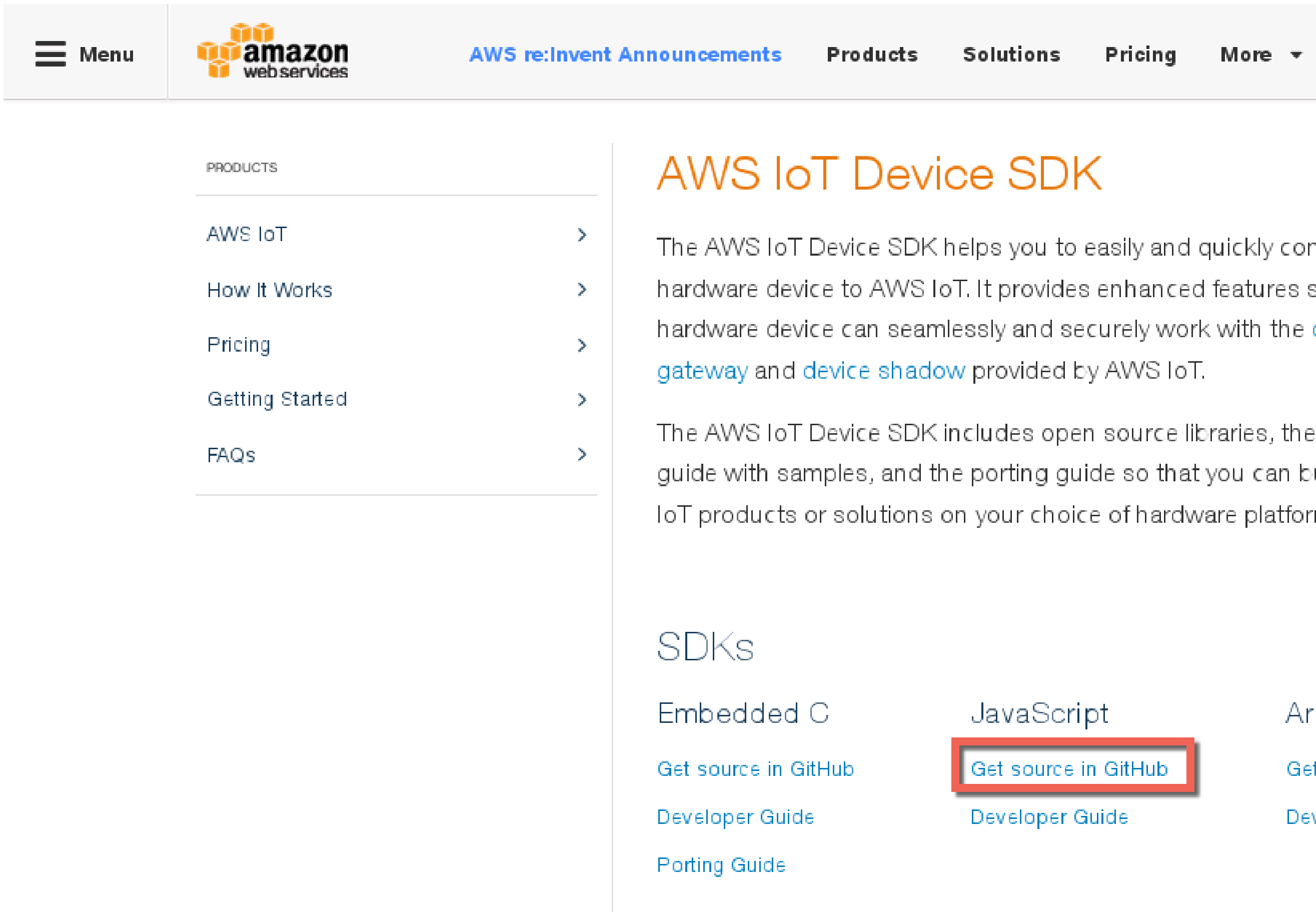
pi@raspberrypi: ~

File Edit Tabs Help

```
pi@raspberrypi ~ $ node -v
v0.12.6
pi@raspberrypi ~ $ npm -v
2.11.2
pi@raspberrypi ~ $ 
```

Install the AWS IoT Device SDK for JavaScript

Now you will port the SDK to the Raspberry Pi. On the AWS IoT Device SDK page, choose the **Get source in GitHub** link:



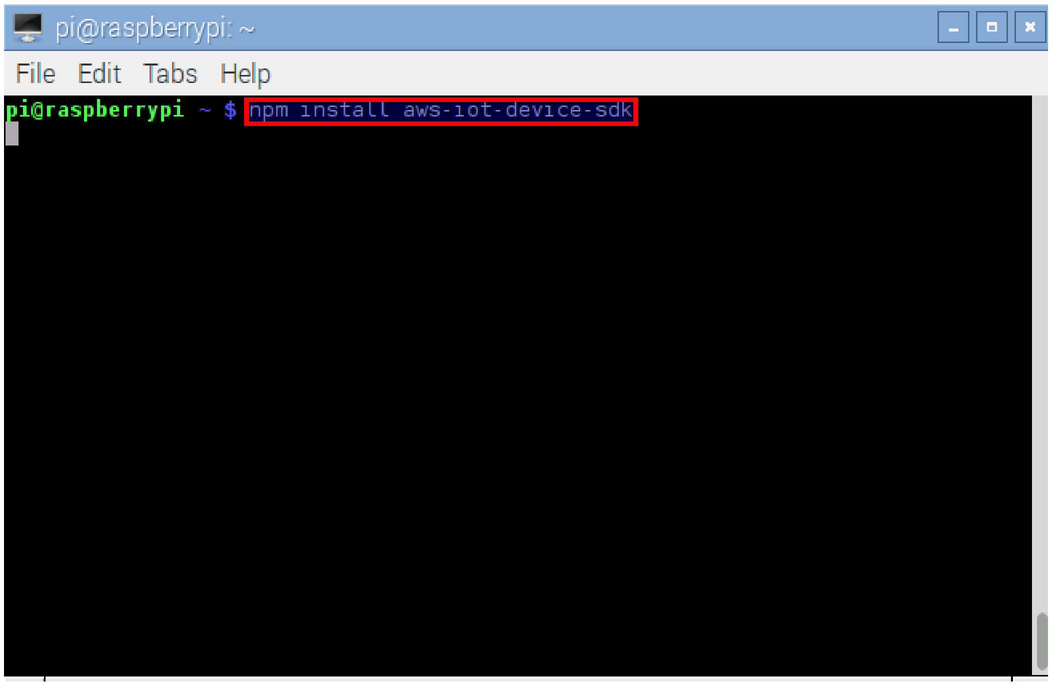
AWS IoT Device SDK

The AWS IoT Device SDK helps you to easily and quickly con
hardware device to AWS IoT. It provides enhanced features s
hardware device can seamlessly and securely work with the
gateway and device shadow provided by AWS IoT.

The AWS IoT Device SDK includes open source libraries, the
guide with samples, and the porting guide so that you can b
IoT products or solutions on your choice of hardware platfor

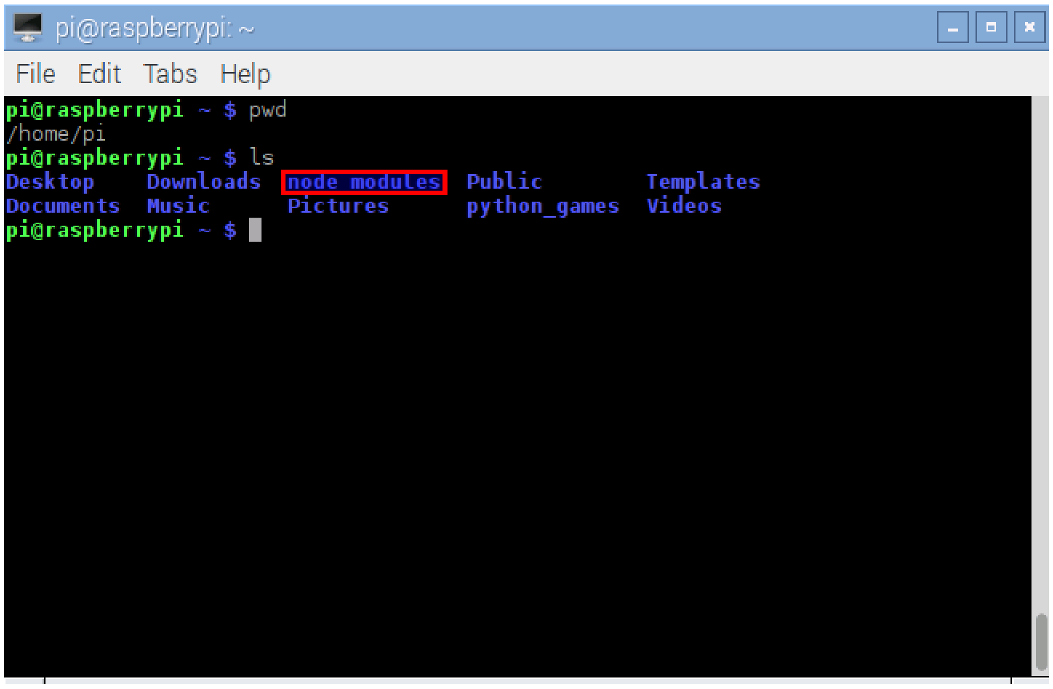
SDKs

Open a console window. To keep things simple, we will use npm to install the SDK from the npm repository:



```
pi@raspberrypi ~ $ npm install aws-iot-device-sdk
```

After the installation is complete, you should find the installed module in ~ directory. (`/home/pi` is the default.)



```
pi@raspberrypi ~ $ pwd
/home/pi
pi@raspberrypi ~ $ ls
Desktop    Downloads  node_modules  Public       Templates
Documents  Music      Pictures      python_games Videos
pi@raspberrypi ~ $
```

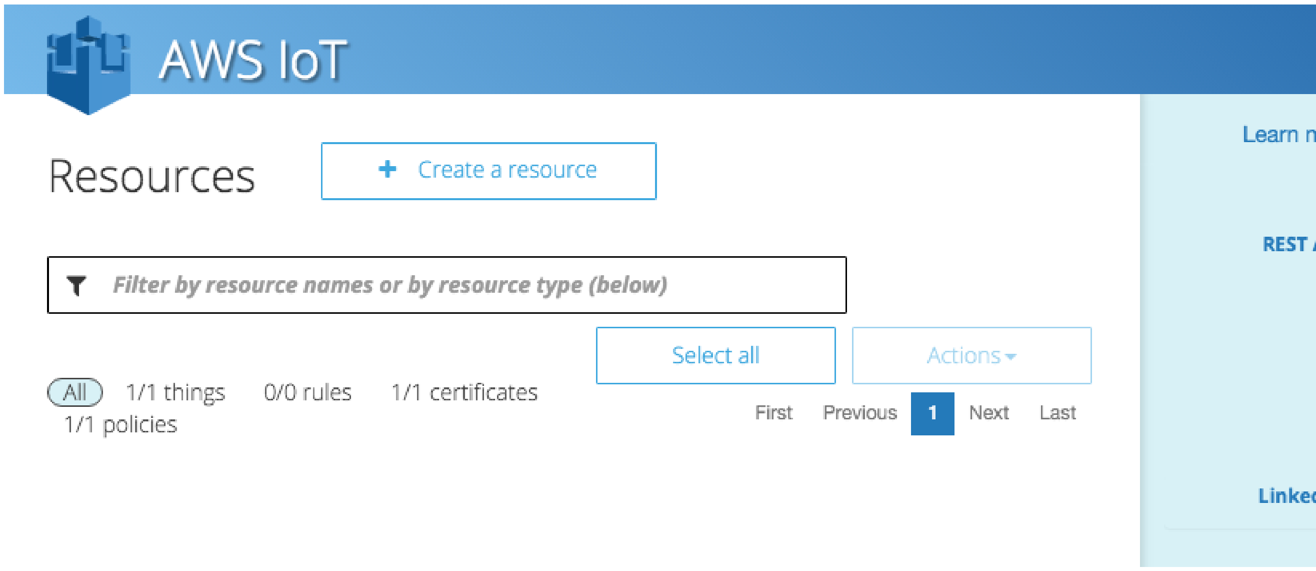
Prepare to Run the Sample Applications

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.

Type `cd ~` to go to your home directory. Create a directory where your certificate, private key, and root CA certificate will be stored. Name the directory `certs`.

Copy your certificate and private key into the directory. Download a root CA certificate from here. Copy the text from the browser, paste it into a file, and then copy it into the `certs` directory.

You will need to configure your personal endpoint, private key, and certificate. If 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. Otherwise, go to the AWS IoT console, double-click **MyNewThing**, and copy everything after "https://" including ".com" from **REST API endpoint**.



By default, the files should be named as follows:

- your private key: `private.pem.key`
- your certificate: `certificate.pem.crt`
- the CA root certificate: `root-CA.crt`

You can edit the cmdline.js file to change the default names used by each sample.

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

Run the Sample Applications

Now you can run examples using node examples/<YourDesiredExample>.js -f <certs location> (assuming you are under `node_modules/aws-iot-device-sdk/`). In this case, the certificates location should be `~/certs/`. You can specify the certificates location and your own host address using command line options. For information, see Certificates.

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

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 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. 196).
- **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 Developer 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 Developer 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 Developer 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.

Metric	Description
PublishIn.Success	A client published on an MQTT topic successfully. Valid Dimensions: Protocol Valid Statistics:1 for success, 0 for failure. Unit: Count
PublishOut.Success	Clients subscribed to an MQTT topic recieved a published message. Valid Dimensions: Protocol Valid Statistics:1 for success, 0 for failure. Unit: Count
Subscribe.Success	AWS IoT message broker received a request to subscribe to an MQTT topic. Valid Dimensions: Protocol Valid Statistics:1 for success, 0 for failure. Unit: Count
Ping.Success	AWS IoT received a Ping message. Valid Dimensions: Protocol Valid Statistics:1 per ping request from the client. Unit: Count
Connect.Success	A client connected to AWS IoT. Valid Dimensions: Protocol Valid Statistics: 1 per successful MQTT connection from the client. Unit: Count
GetThingShadow.Accepted	AWS IoT received a GetThingShadow request. Valid Dimensions: Protocol Valid Statistics:1 for success, 0 for failure. Unit: Count
UpdateThingShadow.Accepted	AWS IoT received a UpdateThingShadow request. Valid Dimensions: Protocol Valid Statistics:1 for success, 0 for failure. Unit: Count

Metric	Description
DeleteThingShadow.Accepted	AWS IoT received a DeleteThingShadow request. Valid Dimensions: Protocol Valid Statistics:1 for success, 0 for failure. Unit: Count
RulesExecuted	AWS IoT executed a rule.. Valid Dimensions: Protocol Valid Statistics:1 for success, 0 for failure. Unit: Count

Dimensions for AWS IoT Metrics

Metrics use the namespace and provide metrics for the following dimension(s):

Dimension	Description
Protocol	The protocol with which the request was made. Valid values are MQTT or HTTP.

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. 199)
- How can I be notified if my things are not publishing data each day? (p. 200)
- How can I be notified if my thing's shadow updates are being rejected each day? (p. 200)

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 UpdateThingShadowSuc
cessAlarm --state-reason "initializing" --state-value OK
```

```
Prompt>aws cloudwatch set-alarm-state --alarm-name UpdateThingShadowSuc
cessAlarm --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/528ce36e8047f6a75ee51ab7bed
db4eb268ad41d2ea881a10b67e8e76924d894",
        "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/iot
monitor-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.

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.

LogLevel
: The logging level. Can be DEBUG, INFO, ERROR, or WARN.

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.

ThingId
: The thing identifier, 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.

DEBUG
: Information that might be helpful when debugging a problem.

  Logs will include DEBUG, INFO, ERROR, and WARN information.

ERROR
: Any error that causes an operation to fail.

  Logs will include ERROR information only.

INFO
: High-level information about the flow of things.

  Logs will include INFO, ERROR, and WARN information.

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.

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.
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.

Operation/Event Name	Description
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.
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.
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.

AWS IoT Limits

The following tables list limits in AWS IoT.

Message Broker Limits

Client ID size	128 bytes of UTF-8 encoded characters.
Connection inactivity (keep-alive interval)	By default, an MQTT client connection is disconnected after 30 minutes of inactivity. When the client sends a PUBLISH, SUBSCRIBE, PING, or PUBACK message, the inactivity timer is reset. A client can request a shorter keep-alive interval by specifying a value between 5-1,200 seconds in the MQTT CONNECT message sent to the server. If a keep-alive value is specified, the server will disconnect the client if it does not receive a PUBLISH, SUBSCRIBE, PINGREQ, or PUBACK message within a period 1.5 times the requested interval. The keep-alive timer starts after the sender sends a CONNACK. If a client sends a keep-alive value of zero, the default keep-alive behavior will remain in place. If a client request a keep-alive shorter than 5 seconds, the server will treat the client as though it requested a keep-alive interval of 5 seconds. The keep-alive timer begins immediately after the server returns a CONNACK to the client. There might be a brief delay between the client's sending of a CONNECT message and the start of keep-alive behavior.
Maximum number of slashes in topic and topic filter	A topic provided while publishing a message or a topic filter provided while subscribing can have no more than eight forward slashes (/).

Maximum inbound unacknowledged messages	The message broker allows 100 in-progress unacknowledged messages per client. (This limit is applied across all messages that require ACK.) When this limit is reached, no new messages will be accepted from this client until an ACK is returned by the server.
Maximum outbound unacknowledged messages	The message broker allows only 100 in-progress unacknowledged messages per client.(This limit is applied across all messages that require ACK.) When this limit is reached, no new messages will be sent to the client until the client acknowledges the in-progress messages.
Maximum retry interval for delivering QoS 1 messages	If a connected client is unable to receive an ACK on a QoS 1 message for one hour, the message broker will drop the message. The client might be unable to receive the message if it has 100 in-flight messages, it is being throttled due to large payloads, or other errors.
Maximum subscriptions per subscribe call	A single SUBSCRIBE call is limited to request a maximum of eight subscriptions.
Message size	The payload for every PUBLISH message is limited to 128 KB. The AWS IoT service will reject messages larger than this size.
Restricted client ID prefix	'$' is reserved for internally generated client IDs.
Restricted topic prefix	Topics beginning with '$' are considered reserved and are not supported for publishing and subscribing except when working with the Thing Shadows service.
Subscriptions per session	The message broker limits each client session to subscribe to up to 50 subscriptions. A SUBSCRIBE request that pushes the total number of subscriptions past 50 will result in the connection being disconnected.
Thing name size	128 bytes of UTF-8 encoded characters. This limit applies for both the thing registry and Thing Shadow services.
Throughput per connection	AWS IoT limits the ingress and egress rate on each client connection to 512 KB/s. Data sent or received at a higher rate will be throttled to this throughput.
Topic size	The topic passed to the message broker when publishing a message cannot exceed 256 bytes of UTF-8 encoded characters.

WebSocket connection duration	WebSocket connections are limited to 24 hours. If the limit is exceeded, the WebSocket connection will automatically be closed when an attempt is made to send a message by the client or server. If you need to maintain an active WebSocket connection for longer than 24 hours, simply close and re-open the WebSocket connection from the client side before the time limit elapses. AWS IoT supports keep-alive values specified in MQTT CONNECT messages. When a client specifies a keep-alive value, the client tells the server to disconnect the client and transmit any last-will message associated with the MQTT session if the server does not receive a message (PUBLISH, SUBSCRIBE, PUBACK, PINGREQ) within 1.5 times the keep-alive period. AWS IoT supports keep-alive values between 5 seconds and 20 minutes. If a client requests no keep-alive (that is, sets the field to 0 in the MQTT CONNECT message), the server will set the keep-alive value to 20 minutes, which corresponds to the maximum idle time supported by AWS IoT of 30 minutes. Most MQTT clients (including the AWS SDK clients) support keep-alive values by sending a PINGREQ if the keep-alive period expires without the transmission of any other message by the client.

Device Shadow Limits

Maximum depth of JSON device state documents	The maximum number of levels in the `"desired"` or `"reported"` section of the JSON device state document is 5. For example:
	
	`"desired": {`
	`    "one": {`
	`        "two": {`
	`            "three": {`
	`                "four": {`
	`                    "five":{`
	`                    }`
	`                }`
	`            }`
	`        }`
	`    }`
	`}`

Maximum number of in-flight, unacknowledged messages	The Thing Shadows service supports up to 10 in-flight unacknowledged messages. When this limit is reached, all new shadow requests will be rejected with a 429 error code.

Maximum number of JSON objects per AWS account	There is no limit on the number of JSON objects per AWS account.
Maximum size of a JSON state document	8 KB.
Maximum size of a thing name	128 bytes of UTF-8 encoded characters.
Shadow lifetime	A thing shadow is deleted by AWS IoT if it has not been updated or retrieved in more than one year.

Security and Identity Limits

Maximum number of policies that can be attached to a certificate	10
Maximum number of named policy versions	5
Maximum policy document size	2048 characters (excluding white space)
Maximum number of device certificates that can be registered per second	15

Throttling Limits

The following table lists the throttling limits for AWS IoT API:

API	Transaction per Second
AcceptCertificateTransfer	10
AttachPrincipalPolicy	15
AttachThingPrincipal	15
CancelCertificateTransfer	10
CreateCertificateFromCsr	15
CreatePolicy	10
CreatePolicyVersion	10
CreateThing	15
CreateThingType	15
DeleteCertificate	10
DeleteCACertificate	10
DeletePolicy	10
DeletePolicyVersion	10
DeleteThing	15
DeleteThingType	15

API	Transaction per Second
DeprecateThingType	15
DescribeCertificate	10
DescribeCACertificate	10
DescribeThing	10
DescribeThingType	10
DetachThingPrincipal	15
DetachPrincipalPolicy	15
DeleteRegistrationCode	10
GetPolicy	10
GetPolicyVersion	15
GetRegistrationCode	10
ListCACertificates	10
ListCertificates	10
ListCertificatesByCA	10
ListOutgoingCertificates	10
ListPolicies	10
ListPolicyPrincipals	10
ListPolicyVersions	10
ListPrincipalPolicies	15
ListPrincipalThings	10
ListThings	10
ListThingPrincipals	10
ListThingTypes	10
RegisterCertificate	10
RegisterCACertificate	10
RejectCertificateTransfer	10
SetDefaultPolicyVersion	10
TransferCertificate	10
UpdateCertificate	10
UpdateCACertificate	10
UpdateThing	10

AWS IoT Rules Engine Limits

Maximum number of rules per AWS account	1000
Actions per rule	A maximum of 10 actions can be defined per rule.
Rule size	Up to 256 KB of UTF-8 encoded characters (including white space).