Icebreaker

Developer Guide

Icebreaker: Developer Guide

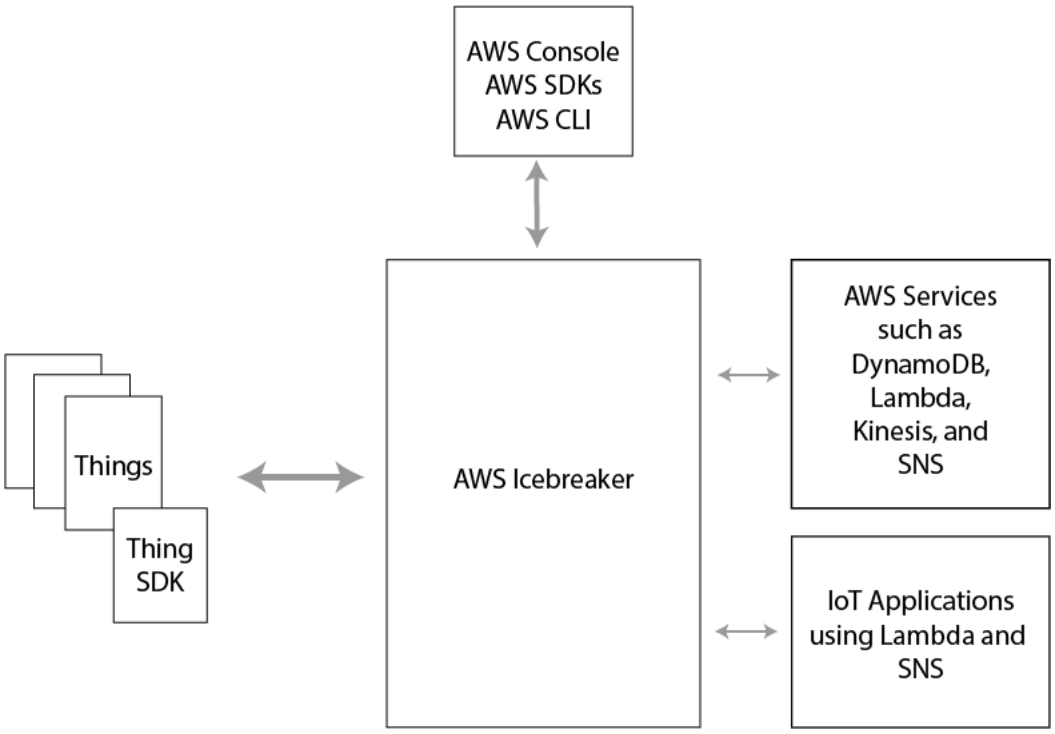
Table of Contents

What Is AWS Icebreaker?

Documentation Version: 0.39

AWS Icebreaker is a service that enables secure, bi-directional communication between internet-connected things (sensors, actuators, devices, applications, etc.) and the cloud over MQTT and HTTP. You can think of Icebreaker as a message processing engine. It receives messages from internet connected "things" and processes those messages. This includes recording, transforming, augmenting, or routing messages to AWS, other web services and applications. Manufacturers, application developers, and enterprises can use Icebreaker to extend the onboard capabilities of physical products by using the cloud to execute logic, communicate with other products/services, and process telemetry data. End users can control their physical devices from smart phone apps.

The following diagram illustrates a high-level view of the Icebreaker service:



You can interact with Icebreaker in a number of ways:

- The Icebreaker Console allows you to configure AWS Icebreaker services within a graphical environment
- The Icebreaker Command Line Interface (CLI) allows you to configure AWS Icebreaker services from the command line
- The Icebreaker SDKs allow you to write applications on top of Icebreaker
- The Icebreaker Thing SDK allows you to write applications in C that run on internet-connected things

Things are any clients such as micro controllers, sensors, actuators, mobile devices, or applications that use Icebreaker to connect to the AWS cloud. The Thing SDK makes it simple to write code running on Internet connected things to communicate with the Icebreaker service.

There are essentially three types of client applications that interact with Icebreaker:

- Embedded applications running on Internet connected devices
- Companion applications running on mobile devices or on the web.
- Server applications

Embedded applications are written in C with the Icebreaker thing SDK. They enable your device to send MQTT messages to and recieve MQTT messages from Icebreaker. They define what information your devices send to Icebreaker and how they respond to messages recieved from Icebreaker.

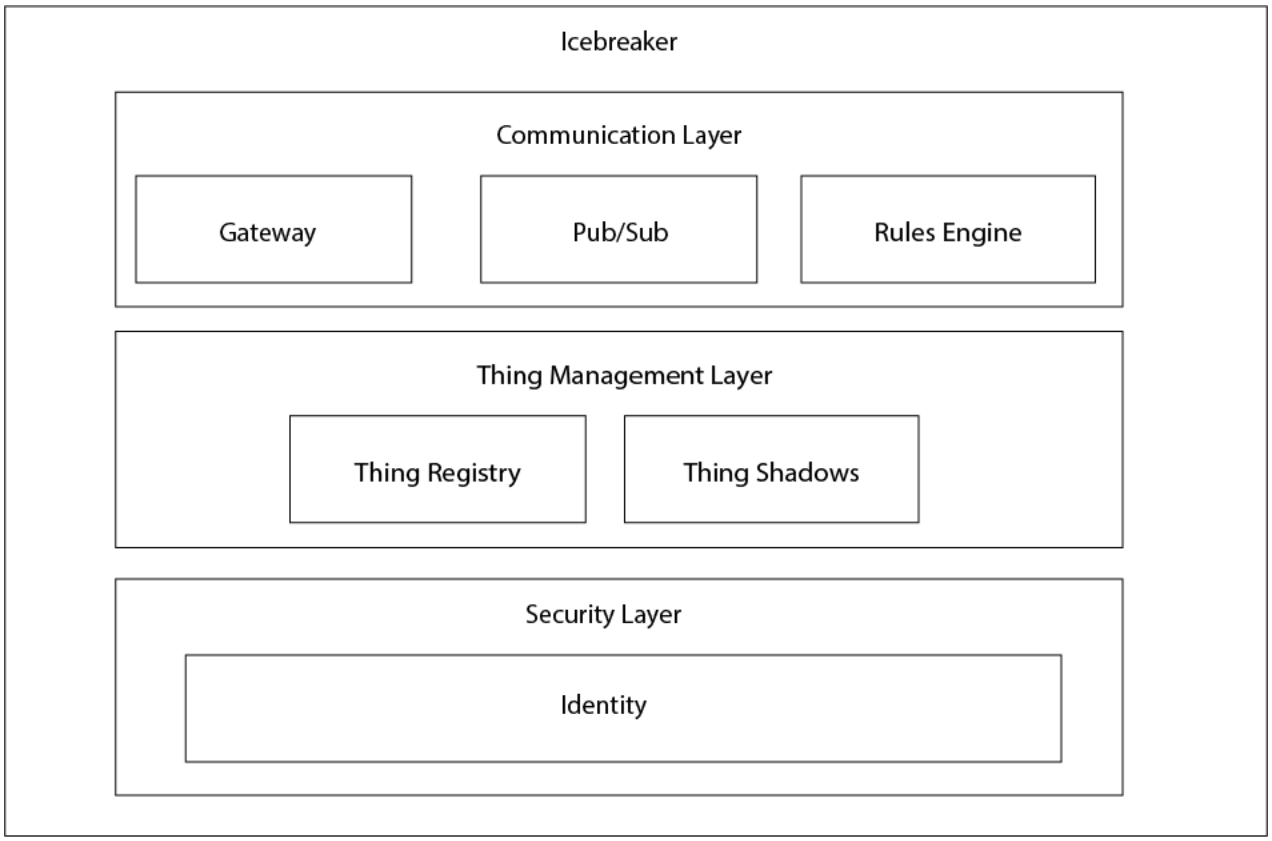
Companion applications are written with the Icebreaker SDKs. These applications allow you to remotely control your devices.

Server applications query Icebreaker for information about your things and process and display the information. A device dashboard showing all active devices is an example of a server application.

Authentication is provided by X509 certificates or AWS Cognito Identities. Authorization is provided by Icebreaker roles and IAM roles.

Icebreaker Components

The following diagram illustrates the structure of the AWS Icebreaker service:



The Icebreaker service is composed of several "layers":

- Communication Layer
- Thing Management Layer
- Security Layer

The communication layer is composed of the Gateway, Pub/Sub Broker, and the Rules Engine. The Gateway is the main entry point into Icebreaker that allows things to securely connect to Icebreaker using MQTT or HTTP. The Pub/Sub Broker is an MQTT broker that enables things to publish and/or subscribe to MQTT messages. The Rules Engine allows you to write rules in an SQL-like syntax. Rules enable filtering, aggregating and forwarding messages, as well as taking appropriate actions on receiving relevant data, such as invoking Lambda functions, or writing message data to DynamoDB tables.

The thing management layer is composed of the Thing Registry and thing shadows. The Thing Registry is a list of all things (devices, sensors, apps, etc) that have been registered with Icebreaker. It provides operations to add, update, retrieve (describe), delete, attach and detach credentials (principals). Thing Shadows provide a virtual representation of a thing. Things can be any type of Icebreaker clients such as micro controllers, sensors, actuators, mobile devices, or applications. Thing Shadows enable you to request the state or update the state of a thing using a REST API or AWS CLI. The state of a thing is represented as a JSON object.

The Security Layer contains the Identity services which performs authentication and authorization operations. Things are authenticated using X.509 certificates or AWS Cognito credentials. Authorization is configured by defining roles.

Getting Started with AWS Icebreaker

This topic will help you get started with Icebreaker by walking you through the following:

- Using the Icebreaker Console
- Install and configure the AWS Icebreaker CLI
- Create a thing in the Thing Registry
- Configuring Authentication and Authorization
- Verify Connectivity and MQTT Publish and Subscribe
- Configure and Test Rules
- Using the Thing Registry and Thing Shadows

There are three ways to interact with the Icebreaker service:

- Using the Icebreaker Console
- Using the Icebreaker CLI
- Using the Icebreaker SDKs

The following sections will describe each of these in more detail.

Using the Icebreaker Console

The Icebreaker console can be found at: Icebreaker Console. The console is divided up into three sections:

- Certificates
- Rules and Integrations
- Access and Policies

These sections are selectable by clicking on the appropriate icon in the upper left hand corner of the console.

Certificates

The certificates section allows you to submit a certificate signing request to generate a new certificate. It also allows you to activate, transfer, deactivate, or revoke and existing certificate.

Rules and Integrations

The rules and integrations section allows you to add a new rule and view your existing rules.

Access and Policies

The access and policies section allows you to add new Icebreaker policies and view existing Icebreaker policies.

Installing AWS CLI

To interact with Icebreaker from the command line you can use the AWS CLI. To install latest version of AWS CLI, see Install AWS CLI. Make sure you have configured your AWS credentials with the AWS CLI. For more information, see Getting Started with AWS CLI.

Download AWS Icebreaker CLI models using the following AWS CLI command:

```
aws s3 cp s3://icebreaker-private-beta/aws-iot-cli.zip aws-iot-cli.zip
```

Extract the .zip file. It will create a directory with the same name as the zip file. That directory will contain an `iot` directory. Copy the `iot` and `iot-data` directories to `~/.aws/models` (on Unix or Linux) or `%UserProfile%\.aws\models` (on Windows).

Open a command prompt/shell, run `aws iot help` and `aws iot-data help` to learn more about Icebreaker commands. Run `aws <service name> <command name> help`"" to learn more about any specific command.

Note
When your account is white listed, by default, the root user in your account has access to the `icebreaker-private-beta` bucket in S3. To enable other IAM users in your account to access the bucket, attach the following IAM policy to the IAM users:

```
{
       "Version": "2012-10-17",
        "Statement": [{
             "Sid": "icebreaker1",
             "Effect": "Allow",
             "Action": [
                  "s3:List*",
                  "s3:Get*"
             ],
             "Resource": [
                  "arn:aws:s3:::icebreaker-private-beta",
                  "arn:aws:s3:::icebreaker-private-beta/*"
             ]
        }]
}
```

Getting Help and Working with Different Parameter Types

Open a command prompt/shell, run "aws iot help" and "aws iot-data help" to learn more about Icebreaker commands. Run "aws *<service-name>* *<command-name>* help" to learn more about any specific command.

If you are having trouble with the formatting of a parameter for a specific command, check the manual by typing help after the command name, for example:

```
$ aws iot <command> help
```

The help for each subcommand describes its function, options, output, and examples.

The sections and examples in this User Guide are verified for Windows OS command prompt. If you are using Linux, OS X, Unix or Windows Power Shell, note that there may be variations in the formats of commands, JSON and other parameter types.

For example, to load parameters from a local file by providing the path to the file the commands will be:

Linux, OS X, or Unix

```
// Read from a file in the current directory
$ aws iot --endpoint-url https://t71u6yob51.execute-api.us-east-1.amazon
aws.com/beta create-certificate-from-csr --certificate-signing-request
file://csrfile.pem --set-as-active

// Read from a file in /path
$ aws iot --endpoint-url https://t71u6yob51.execute-api.us-east-1.amazon
aws.com/beta create-certificate-from-csr --certificate-signing-request
file:///path/csrfile.pem --set-as-active
```

Windows

```
// Read from a file in C:\temp
$ aws iot --endpoint-url https://t71u6yob51.execute-api.us-east-1.amazon
aws.com/beta create-certificate-from-csr --certificate-signing-request
file://C:\temp\csrfile.pem --set-as-active
```

For additional references on formatting commands, JSON parameters, etc., see Specifying Parameter Values for the AWS Command Line Interface

Create a thing in the Thing Registry

To connect a thing to Icebreaker you must first create a thing in the Thing Registry. The Thing Registry allows you to keep a simple record of all things connected to Icebreaker.You can create a thing by using the `create-thing` CLI command or by using the Icebreaker console.

In a command prompt/terminal, run the following command:

```
aws iot --endpoint-url
https://t71u6yob51.execute-api.us-east-1.amazonaws.com/beta create-thing
--thing-name <thing-name>
```

This command takes a thing-name, creates a new thing, and displays the thing ARN and name:

```
{
    "thingArn": "arn:aws:iot:us-east-1:<aws-account-id>:thing/lightbulb",
    "thingName": "lightbulb"
}
```

You can use the `list-things` command to confirm that the thing is created in the Thing Registry:

```
aws iot --endpoint-url
https://t71u6yob51.execute-api.us-east-1.amazonaws.com/beta list-things
```

This command lists all things in the Thing Registry for your AWS account:

```
{
    "things": [
        {
            "attributes": {},
            "thingName": "lightbulb"
        }
    ]
}
```

Configuring Authentication and Authorization

Icebreaker requires an X509 certificate for thing authentication. You can have Icebreaker create a certificate and key pair for you or you can create a keypair and a Certificate Signing Request (CSR) and submit them to Icebreaker to generate a certificate. For this tutorial we will let Icebreaker create the key pair and certificate:

To create a certificate, use the `create-keys-and-certificate` CLI command:

```
aws iot --endpoint https://t71u6yob51.execute-api.us-east-1.amazonaws.com/beta
create-keys-and-certificate --set-as-active
```

The command output contains the key pair and certificate. Notice the certificate ARN you will use it when saving the certificate to a file and when attaching the certificate to your thing.

Save the certificate to a file using the `describe-certificate` command:

```
aws iot --endpoint-url
https://t71u6yob51.execute-api.us-east-1.amazonaws.com/beta describe-certificate
--certificate-id < id> --output text --query
certificateDescription.certificatePem > cert.pem
```

where --certificate-id specifies the part after the last slash (/) in the certificate ARN (`arn:aws:iot:us-east-1:your-aws-account-id:cert/<id-is-found-here>`)

Copy the key pair from the command line and save them into separate .pem files (private-key.pem and public-key.pem).

> **Note**
> When copying the keys and certificate into a text file make sure to remove the embedded newlines ('\n').

Now that you have the certificate you can attach it to your thing. You attach a certificate to a thing using the `attach-thing-principal` CLI command which takes two parameters:

- --thing-name - the name of the thing to attach the certificate to
- --principal - the ARN of your certificate

The following shows how to call `attach-thing-principal`:

```
aws iot --endpoint https://t71u6yob51.execute-api.us-east-1.amazonaws.com/beta
attach-thing-principal --thing-name "<thing-name>" --principal
"<certificate-arn>"
```

The certificate will also need to be copied onto your thing and be referenced by the code that sends messages to Icebreaker.

Create an Icebreaker Policy

Now you have created a certificate you need to tell Icebreaker the operations the certificate holder can perform. This is done using an Icebreaker policy. Use the `create-policy` command to create an Icebreaker policy. The --policy-document argument contains JSON that specifies the permissions assigned to the policy:

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

This JSON policy document allows all IoT operations on all resources. Save this text to a file and specify it in the following command line.

```
aws iot --endpoint-url
https://t71u6yob51.execute-api.us-east-1.amazonaws.com/beta create-policy
--policy-name PubSubToAnyTopic --policy-document
file://<path-to-your-policy-document>
```

The policy must be associated to a certificate to be useful, you do this by attaching the policy to the service. Attach the policy to your certificate by using the following command:

```
aws iot --endpoint-url
https://t71u6yob51.execute-api.us-east-1.amazonaws.com/beta
attach-principal-policy --principal "<certificate ARN>" --policy-name
"PubSubToAnyTopic"
```

Verify MQTT Subscribe and Publish

MQTT clients require a root CA certificate to authenticate Icebreaker. Download the root CA certificate file from: root certificate

To test sending and receiving MQTT messages you need a MQTT client. If you do not have one installed, install Mosquitto, a MQTT broker and client. To subscribe to an MQTT topic, use the mosquitto_sub command. Provide the root CA certificate, the Icebreaker issued certificate, and the corresponding private key:

```
mosquitto_sub --cafile path-to-cert\rootCA.pem --cert path-to-cert\cert.pemc
--key path-to-cert\privateKey.pem -h g.us-east-1.pb.iot.amazonaws.com -p 8883
-q 1 -d -t foo/bar -i clientid1
```

where:

--cert : the Icebreaker certificate

--key : your private key

-h :the Icebreaker service host

-p : the port to use on the service host

-q : the MQTT Quality of Service (QoS) level

-d : enable debug messages

-t : the topic

-I : the client ID

> **Note**
> Ensure egress to port 8883 is allowed on your network. The command will remain running and display information when messages are received.

Open a second command prompt/shell and use the mosquitto_pub command to publish a message:

```
mosquitto_pub --cafile certs\rootCA.pem --cert certs\cert.pem --key
certs\privateKey.pem -h g.us-east-1.pb.iot.amazonaws.com -p 8883 -q 1 -d -t
foo/bar -i clientid2 -m "Hello, World"
```

where:

--cert : the Icebreaker certificate

--key : your private key

-h :the Icebreaker service host

-p : the port to use on the service host

-q : the MQTT Quality of Service (QoS) level

-d : enable debug messages

-t : the topic

-i : the client ID

-m : the message text to send

> **Note**
> Ensure egress to port 8883 is allowed on your network.

This command sends a message, waits for acknowledgement and terminates.

Configure and Test Rules

Now that you can send MQTT messages, you can specify what Icebreaker should do with the messages it receives. Rules allow you to specify how Icebreaker handles messages. You can configure Icebreaker rules to continuously process messages published on topics. You can configure the rules to take 'actions' such as inserting into a DynamoDB table or calling a Lambda function. You can configure multiple rules on a single topic.

Create an IAM Role for Icebreaker

Create an IAM role that Icebreaker can assume to perform actions when rules are triggered

Set the following 'assume role policy document' (i.e. "Trust Relationship") to a file:

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

Use the `create-role` to create the IAM role.

```
aws iam create-role --role-name icebreaker;-actions-role
--assume-role-policy-document file://<path-to-file>/trust-policy-file
```

Save the role ARN from the command output, you will need it when creating a rule.

Grant Permissions to the Role

Grant the role permissions to write to DynamoDB and to invoke Lambda functions. Save the following to a file:

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

Call `create-policy` and specify the file path from above:

```
aws iam create-policy --policy-name Icebreaker-actions-policy --policy-document
file://<IAM-policy-document-file-path>
```

Attach the policy you just created to the role using the attach-policy-role command:

```
aws iam attach-role-policy --role-name "Icebreaker-actions-role" --policy-arn
"<policy-ARN>"
```

Create a Rule to Insert a Message into a DynamoDB Table

Create a DynamoDB table using the DynamoDB console

> **Note**
> Icebreaker requires the DynamoDB table to have a Hash Key of type string named “topic”, and a Range Key of type number named “timestamp”. Use the defaults for all other values.

> **Note**
> Currently, the DynamoDB table must be in the AWS region US-EAST-1.

Create a rule to trigger on a topic and insert an item into the sample DynamoDB table. Rules are specified in JSON using an SQL-like syntax. The following JSON shows how to specify rule that will write all messages sent to the foo/bar topic to the sampleTable DynamoDB table:

```
{
  "sql": "SELECT * FROM 'foo/bar'",
  "ruleDisabled": false,
  "actions": [{
      "dynamoDB": {
        "roleArn": "arn:aws:iam::your-aws-account-id:role/icebreaker-actions-
role",
        "tableName": "sampleTable"
      }
    }]
}
```

Save this text to a file (for example DynamoDbRule) and specify it in the create-topic-rule command:

```
aws iot --endpoint-url
https://t71u6yob51.execute-api.us-east-1.amazonaws.com/beta create-topic-rule
--rule-name saveToDynamoDB --topic-rule-payload
file://<path-to-file>/DynamoDbRule.
```

Publish a message on the foo/bar topic using mosquito_pub to invoke the rule:

```
mosquitto_pub --cafile certs\rootCA.pem --cert certs\cert.pem --key
certs\privateKey.pem -h g.us-east-1.pb.iot.amazonaws.com -p 8883 -q 1 -d -t
foo/bar -i clientid2 -m "{\"msg\" : \"Hello, World\"}"
```

> **Note**
> Ensure egress to port 8883 is allowed on your network.

Verify the data is written to the DynamoDB table by going to the DynamoDB console and double-clicking on sampleTable. The contents of the table is displayed.

Create a Rule to Invoke a Lambda Function

Create a Lambda function using the Lambda console . From the Lambda console, click the **Create a Lambda Function** button. Select the **hello-world** blueprint. Name the function “myHelloWorld”. Scroll down to the **Lambda function handler and role** section. Under **Handler** leave the default value (index.handler) and select **Basic Execution Role** in the **Role** dropdown box. In the Role page click **Allow** button. Back in the new function page, click the **Next** button and then the **Create function** button. On the page displayed, note the function ARN on the right-hand side of the screen, you will need to specify it when creating a rule.

Create a rule to trigger on a topic call the myHelloWorld Lambda function. Rules are specified in JSON using an SQL-like syntax. The following JSON shows how to specify rule that will call the Lambda function when any message is sent to the foo/bar topic:

```
{
                                "sql": "SELECT * FROM 'foo/bar'",
                                "ruleDisabled": false,
                                "actions": [{
                                    "lambda": {
                                        "roleArn": "arn:aws:iam::your-aws-ac
count-id:role/icebreaker-actions-role",
                                       "functionName": "arn:aws:lambda:us-east-
1:your-aws-account-id:function:myHelloWorld"
                                    }
                                 }]
                            }
```

Note
The function name ARN can be found in the Lambda Console on the page displayed immediately after you create the function.

Save this text to a file (for example LambdaRule) and specify it in the create-topic-rule command:

```
aws iot --endpoint-url
https://t71u6yob51.execute-api.us-east-1.amazonaws.com/beta create-topic-rule
--rule-name invokeLambda --topic-rule-payload file://<path-to-file>/LambdaRule.
```

Invoke the rule by publishing an MQTT message on the `foo/bar` topic using `mosquitto_pub` command:

```
mosquitto_pub --cafile certs\rootCA.pem --cert certs\cert.pem --key
certs\privateKey.pem -h g.us-east-1.pb.iot.amazonaws.com -p 8883 -q 1 -d -t
foo/bar -i clientid2 -m "{\"key1\" : \"Hello, World\"}"
```

Note
Ensure egress to port 8883 is allowed on your network.

Go to the Lambda console and observe the CloudWatch Logs generated by the Lambda function to verify that it was invoked and the message was received.

Using Thing Registry and Thing Shadows

The Thing Registry allows you keep a simple record of all things connected to Icebreaker. Thing Shadows allow applications to easily interact with the things connected to Icebreaker. The general data flow of using Thing Registry and Thing Shadows is as follows:

- A thing, such as an internet connected light bulb, is registered in the Thing Registry.
- The light bulb publishes it's current state to Icebreaker. For example "power = on" and "color = green". Icebreaker stores the state in Thing Shadows.
- An application, such as a mobile app controlling the light bulb, uses a RESTful API to query Icebreaker for the last reported state of the light bulb, without the complexity of communicating directly to the light bulb.
- An application uses a RESTful API to request a change in thing state. For example, a mobile app requests the light bulb change it's color to green. The application does not have to deal with the complexity of communicating directly with the thing or being resilient to issues such as intermittent connectivity. Icebreaker handles synchronizing the desired state with the thing the next time the it is connected.

For quick tour of Thing Shadows, we will use the Mosquitto client to simulate the light bulb in our example, and use the AWS CLI commands to simulate the mobile application.

Create a Thing

On a command prompt, run the following command:

```
aws iot --endpoint-url
https://t71u6yob51.execute-api.us-east-1.amazonaws.com/beta create-thing
--thing-name lightbulb1
```

Use the following command to confirm that the thing is created in the Thing Registry:

```
aws iot --endpoint-url
https://t71u6yob51.execute-api.us-east-1.amazonaws.com/beta list-things
```

Simulate a Thing

A thing synchronizes with its shadow in Icebreaker using MQTT pub/sub (or using a RESTful API). To report its state over MQTT, the thing publishes on topic $aws/things/<thing-name>>. If there is an error such as a version conflict when merging the reported state with the shadow, Icebreaker publishes a message on topic $aws/things/{thingName}/shadow/update/rejected. The thing should subscribe on this topic, if it needs to be notified of any such error. To receive updates from the shadow, the thing should subscribe to topic $aws/things/{thingName}/shadow/update.

To simulate the thing using the Mosquitto client, open a new command prompt/shell and subscribe to the rejected topic with this command:

```
mosquitto_sub –cafile <rootCA-cert> --cert <thing-cert> --key <thing-private-key>
-h g.us-east-1.pb.iot.amazonaws.com -p 8883 -q 1 -d -t
$aws/things/lightbulb1/shadow/update/rejected
```

Open another command prompt/shell and subscribe to the update/accepted topic using the following command, to simulate receiving updates from Icebreaker:

```
mosquitto_sub –cafile <rootCA-cert> --cert <thing-cert> --key <thing-private-key>
-h g.us-east-1.pb.iot.amazonaws.com -p 8883 -q 1 -d -t
$aws/things/lightbulb1/shadow/update/accepted
```

Open another command prompt/shell and publish a message on the state topic using the following command, to simulate reporting a state to Icebreaker from the thing. In this command, the example ‘light-bulb’ thing is reporting that its current color is red:

```
mosquitto_pub –cafile <rootCA-cert> --cert <thing-cert> --key <thing-private-key>
-h g.us-east-1.pb.iot.amazonaws.com -p 8883 -q 1 -d -t
$aws/things/lightbulb1/shadow/update -m "{ \"state\": {\"reported\": { \"color\":
\"RED\" } } }"
```

Simulate an App Controlling a Thing

An application such as a mobile app or a web application can authenticate using AWS credentials and use the Icebreaker RESTful API to get the reported state of a thing or set a desired state. While applications typically use the AWS APIs or SDKs, we are using the AWS CLI to simulate the behavior of an application.

To get the last reported state of a thing, use the following command:

```
aws iot-data --endpoint-url https://g.us-east-1.pb.iot.amazonaws.com
get-thing-state --thing-name lightbulb1 output.txt && cat output.txt
```

Note
use '&& type output.txt' on Windows

To request an update, for example to set a desired state on a thing, use the following command:

```
aws iot-data --endpoint-url https://g.us-east-1.pb.iot.amazonaws.com
update-thing-state --thing-name lightbulb1 --payload "{ \"state\": {\"desired\":
{ \"color\": \"GREEN\" } } }"  output.txt && cat output.txt
```

This example sets the light bulb's color to green.

Note
use '&& type output.txt' on Windows

Go back to the command prompt where you have the Mosquitto client subscribed to the sync topic. Verify that the client received the desired state in a message from Icebreaker.

Delete Thing

Delete the thing shadow using the following command:

```
mosquitto_pub –cafile <rootCA-cert> --cert <thing-cert> --key <thing-private-key>
-h g.us-east-1.pb.iot.amazonaws.com -p 8883 -q 1 -d -m "{ \"state\": null }"
```

Delete the thing from the Thing Registry using the following command:

```
aws iot --endpoint-url
https://t71u6yob51.execute-api.us-east-1.amazonaws.com/beta delete-thing
--thing-name lightbulb1
```

AWS Icebreaker Authentication

AWS Icebreaker performs authentication using X509 certificates or Amazon Cognito Identities.

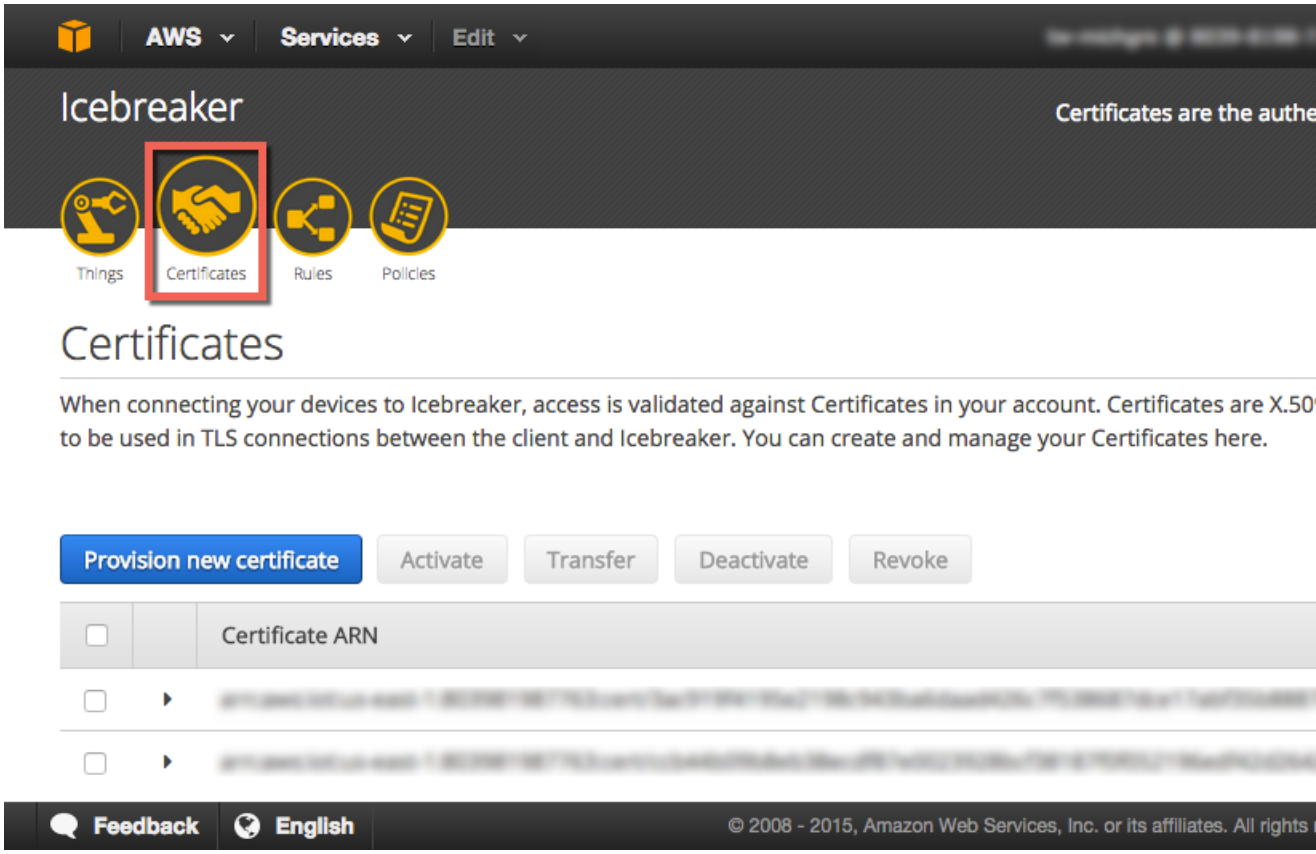
Certificates

Internet connected things need a certificate to authenticate with Icebreaker. You can create and manage certificates using the Icebreaker console or the Icebreaker CLI. The following operations are available:

- Provision a new certificate
- Activate an existing certificate
- Deactivate an existing certificate
- Revoke an existing certificate
- Transfer a certificate to another AWS account

Certificates and the Icebreaker Console

Click the Certificates button as shown in the following screen shot:



Click the Provision new certificate button to create a new certificate. Otherwise, select the certificate you wish to manage and click the appropriate button: Active, Transfer, Deactivate, or Revoke.

Certificates and the Icebreaker CLI

The Icebreaker CLI provides a set of commands for managing certifcates:

- create-keys-and-certificate - creates a new key pair and certificate
- create-certificate-from-csr - creates a new certificate from the specified CSR and private key.
- describe-certificate - returns information about a certificate
- update-certificate - updates a certificate
- delete-certificate - deletes a certificate
- list-certificates - list all certificates in your AWS account
- transfer-certificate - transfer a certificate to another AWS account
- cancel-certificate-transfer - cancel a pending certificate transfer initiated by the caller.
- reject-certificate-transfer - Reject a pending certificate transfer targeted at the caller.

For more information on any of these commands, type "aws iot *<command>* help" in a command-prompt/terminal window.

Amazon Cognito Identities

Amamzon Cognito allows you to create temporary, scoped credentials for accessing AWS services. The most common scenario for using Cognito credentials is when you are writing a mobile or web application that communicates with Icebreaker to control an internet connected thing. For example, you may have an app on your phone that lets you change the color of an internet connected lightbulb. The Cognito identity allows your mobile app to connect to AWS without having to hard-code your AWS credentials. For more information see, Amazon Cognito.

AWS Icebreaker Authorization

Icebreaker uses Icebreaker policies, IAM roles, and IAM policies files to authorize access.

Icebreaker Policy

After creating a certificate for your internet connected thing, you must create and attach an Icebreaker policy that will determine what Icebreaker operations the thing may perform. Icebreaker policies are JSON documents and they follow the same conventions as IAM policies. For more information, see Overview of IAM Policies. An Icebreaker policy looks like the following:

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

This Icebreaker policy allows all Icebreaker actions on all Icebreaker resources. You can specify specific actions:

- iot:Publish
- iot:Subscribe
- iot:UpdateThingShadow
- iot:GetThingShadow

You can also specify specific resources. The resource you can specify depends on the action:

Icebreaker Resources

Action	Resource
iot:DeleteThingShadow	thing name

Action	Resource
iot:Publish	topic ARN
iot:Subscribe	topic filter ARN
iot:UpdateThingShadow	thing name
iot:GetThingShadow	thing name

For more information see Icebreaker Policy Samples (p. 21).

IAM Roles and Policies

IAM roles and policies are used to allow Icebreaker to perform actions on your behalf when a rule is triggered by an incoming MQTT or HTTP message. IAM roles determine "who" can perform the actions specified in the IAM policy. The IAM policy specifies "what" actions the member of a role can perform.

IAM roles are created with the AWS CLI `create-role` command:

```
aws iam create-role --role-name icebreaker-actions-role --assume-
role-policy-document file://<path-to-file>/trust-policy-file
```

The trust-policy-file is a JSON document that allows Icebreaker to assume the role:

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

Once you create the IAM role, you create a policy that specifies what the members of the role are allowed to do. The following policy allows access to invoke Lambda functions and to access DynamoDB databases:

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

To create the policy, use the AWS CLI create-policy command:

```
aws iam create-policy --policy-name icebreaker-actions-policy -- policy-document
file://<file path>/dynamo-and-lambda-policy
```

In this example the policy document described above is called dyanmo-and-lambda-policy.

You must now attach the policy to the IAM role by using the AWS CLI `attach-role-policy` command:

```
aws iam attach-role-policy --role-name "icebreaker-actions-role" --policy-arn
"<policy-ARN>"
```

The policy is specified with an ARN which is in the output from the `create-policy` command.

IAM roles and policies can also be created using the IAM console, for more information, see IAM Policies and IAM Roles.

Icebreaker Policy Samples

Certificates require a named policy which is specified in a JSON document. There are four components in an Icebreaker 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 - Deletes a thing shadow
- Resource - must be set to a MQTT topic ARN. a MQTT topic filter ARN, or a thing-name
  -

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

Example 1

This policy allows the certificate holder to publish and subscribe to all topics in the specified AWS account.

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

Example 2

This policy allows the certificate holder to publish to all topics in the specified AWS account.

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

Example 3

This policy allows the certificate holder to publish to the topic "foo/bar" in the specified AWS account

```
{
    "Version": "2012-10-17",
    "Statement": [{
         "Effect": "Allow",
         "Action": ["iot:Publish"],
         "Resource": ["arn:aws:iot:us-east-1:420622145616:topic/foo/bar"]
     }]
}
```

Example 4

This policy allows the certificate holder to publish to the topics "foo/bar" and "foo/baz" in the specified AWS account

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": ["iot:Publish"],
        "Resource": [
                "arn:aws:iot:us-east-1:420622145616:topic/foo/bar",
                "arn:aws:iot:us-east-1:420622145616:topic/foo/baz"
        ]
    }]
}
```

Example 5

This policy prevents the certificate holder from publishing to the topic "foo/bar" in your AWS account

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Deny",
        "Action": ["iot:Publish"],
        "Resource": ["arn:aws:iot:us-east-1:420622145616:topic/foo/bar"]
     }]
}
```

Example 6

This policy allows the certificate holder to subscribe to the topic filter "foo/*" in the specified AWS account

```
{
    "Version": "2012-10-17",
    "Statement": [{
         "Effect": "Allow",
         "Action": ["iot:Subscribe"],
         "Resource": ["arn:aws:iot:us-east-1:420622145616:topicfilter/foo/*"]
     }]
}
```

Example 7

This policy allows the certificate holder to subscribe to the topic filter "foo/+/bar" in the specified AWS account

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": ["iot:Subscribe"],
       "Resource": ["arn:aws:iot:us-east-1:420622145616:topicfilter/foo/+/bar"]

     }]
}
```

Example 8

This policy allows the certificate holder to publish to the topic "foo" and to subscribe to the topic filter "foo/bar/*" in the specified AWS account

```
{
    "Version": "2012-10-17",
    "Statement": [{
         "Effect": "Allow",
         "Action": ["iot:Publish"],
         "Resource": ["arn:aws:iot:us-east-1:420622145616:topic/foo"]
          },
         {
         "Effect": "Allow",
         "Action": ["iot:Subscribe"],
        "Resource": ["arn:aws:iot:us-east-1:420622145616:topicfilter/foo/bar/*"]

    }]
}
```

Example 9

This policy allows the certificate holder to publish to the topic "foo" and prevents it from publishing to the topic "bar"

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": ["iot:Publish"],
        "Resource": ["arn:aws:iot:us-east-1:420622145616:topic/foo"]
         },
         {
         "Effect": "Deny",
         "Action": ["iot:Publish"],
         "Resource": ["arn:aws:iot:us-east-1:420622145616:topic/bar"]
    }]
}
```

Example 10

This policy allows the certificate holder to subscribe to the topic filter "foo/bar?

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

```
    "Statement": [{
        "Effect": "Allow",
        "Action": ["iot:Subscribe"],
        "Resource": ["arn:aws:iot:us-east-1:420622145616:topicfilter/foo/bar"]

    }]
}
```

Example 2

This policy allows the certificate holder to delete all thing shadows in the specified AWS account.

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

Working with Rules

Use aws iot create-topic-rule to create a rule. create-topic-rule takes the following parameters:

- --rule-name : name of the rule to create
- --topic-rule-payload : the JSON document (string) that describes the rule to create

The following JSON object shows the structure of the topic rule payload document:

```
{
"sql": "string",
     "description": "string",
     "actions": [{
       "dynamoDB": {
                 "tableName": "string",
                 "roleArn": "string"
            },
            "lambda": {
                "functionName": "string",
                "roleArn": "string"
            }
     }]
}
```

For each action element in the JSON document, specify the role ARN for the role that Icebreaker will assume when executing the rule.

Create Role for Icebreaker to Assume When Executing the Rule

Use the `create-role` command to create a role that Icebreaker will assume when executing the rule:

```
aws iam create-role --role-name role-name --assume-role-policy-document
role-policy-document
```

The `create-role` command takes the following arguments:

--role-name - the name of the role
--assume-role-policy-document - the JSON trust relationship policy document

Use the following trust relationship policy:

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

Save this text to a file named assume-role-policy and specify it when calling `create-role`.

```
aws iam create-role --role-name my-iam-role --assume-role-policy-document
file://<path-to-file>/assume-role-policy
```

Create Policy for Role

A policy document describes what principal that assumes the role is allowed to do. Use the `create-policy` command to create the policy:

```
aws iam create-policy --policy-name myPolicy --policy-document
file://<path-to-file>/policy-document
```

--policy-name - the policy name
--policy-document - a JSON document that describes the permissions granted

Note the policy ARN from the command output, you will need it to attach the policy to a role:

```
{
    "Policy": {
        "PolicyName": "my-policy",
        "CreateDate": "2015-07-28T23:59:02.629Z",
        "AttachmentCount": 0,
        "IsAttachable": true,
        "PolicyId": "ANPAJW6A5F3MTDUCMEDKA",
        "DefaultVersionId": "v1",
        "Path": "/",
        "Arn": "arn:aws:iam::your-aws-account-id:policy/my-policy",
        "UpdateDate": "2015-07-28T23:59:02.629Z"
    }
}
```

Attach the Policy to a Role

Use the `attach-role-policy` command to attach your policy to your role.

```
aws iam attach-role-policy --role-name "my-role" --policy-arn
"arn:aws:iam::your-aws-account-id:policy/my-policy"
```

Updating a Rule

Use the `replace-topic-rule` command to update a rule:

```
aws iot replace-topic-rule --rule-name “rule-to-replace” --topic-rule-payload
file://<path-to-file>/updated-rule-payload
```

Deleting a Rule

Use the `delete-topic-rule` command to delete a rule:

```
aws iot delete-topic-rule --rule-name “rule-to-delete”
```

SQL Syntax Reference

A simplified SQL-like syntax is used to filter messages arriving on an MQTT topic and push the data elsewhere. In this document we assume you understand the structure of basic SQL syntax (the language for querying relational databases) and will instead focus on the differences. If you are unfamiliar with SQL, please visit W3Schools SQL Tutorial.

Note
SQL queries can be multi-line, just make sure newlines are escaped properly when posting the rule.

FROM Clause - uses MQTT Topics Instead of Tables

In this new technology we apply a familiar technology (SQL) to a newer domain (real-time data feeds from sensors).

ANSI SQL	Icebreaker SQL	Notes

Table	Data source/MQTT topic	A data source is written as a function like:
		`SELECT * FROM mqtt('com.example/sensors/+')`
		However, it defaults to MQTT so you can simply write it as a string:
		`SELECT * FROM 'com.example/sensors'`

Column	JSON property in a message	All data processed by a query is assumed to be JSON. A flat JSON document should require SQL that is identical to traditional SQL. If the JSON is nested, please reference the section titled "JSON Extensions".

Like traditional ANSI SQL, white-space is insignificant and capitalization for SQL keywords is not important. However, capitalization in strings and JSON properties is still significant. For the purposes of this document we capitalize all keywords, as is common practice with SQL. Example:

```
                        SELECT *
FROM 'example/sensor1'
WHERE temperature > 70
```

WHERE Clause

Most of the expressions that are allowed in ANSI SQL are also allowed here.

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)
+	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
>=	Greater than or equal, comparison	6 >= 5
CASE ... WHEN ... THEN ... ELSE ... END	Case statement	CASE location WHEN 'home' THEN 'off' WHEN 'work' THEN 'on' ELSE 'silent' END

Actions

In a rule, you can specify the actions you want Icebreaker to take on executing the rule. Currently supported actions are:

- inserting into a DynamoDB table
- invoking a Lambda function
- writing to an S3 bucket
- publishing on an SNS topic
- publishing to an SQS queue
- publishing to a Kinesis stream
- republishing on a different topic in Icebreaker

The following JSON document describes a rule that writes the message data to DynamoDB and invokes a Lambda function.

```
{
  "ruleName": "Test Multiple Rules",
  "topicRulePayload": {
    "sql": "Select * from 'topic/test'",
    "description": "test rule for all supported actions ",
    "actions": [
      {
        "lambda": {
          "roleArn": "arn:aws:iam::your-aws-account-id:role/rulesengine_ac
tion_lambda",`
          "functionName": "lambda function arn"
        },
        "dynamoDB": {
         "roleArn": "arn:aws:iam::your-aws-account-id:role/rulesengine_action_dy
namoDB",
          "tableName": "table name"
        },
        "s3": {
          "roleArn": "arn:aws:iam::your-aws-account-id:role/rulesengine_ac
tion_s3",
          "bucketName": "rulesengineactions3",
          "key": "s3 file url",
          "region": "us-east-1"
        },
        "sns": {
         "roleArn": "arn:aws:iam::your-aws-account-id:role/rulesengine_action_dy
namoDB",
          "topicArn": "arn:aws:sns:us-west-2:your-aws-account-id:MyTopic"
        },
        "sqs": {
          "roleArn": "arn:aws:iam::your-aws-account-id:role/rulesengine_ac
tion_sqs",
          "queueUrl": "<queue name>",
          "useBase64" : "false"
        },
        "kinesis": {
          "roleArn": "arn:aws:iam::your-aws-account-id:role/rulesengine_ac
tion_kinesis",
          "streamName": "rulesengine_action_kinesis",
```

```
          "partitionKey": "haha",
          "endpoint": "kinesis.us-east-1.amazonaws.com",
          "region": "us-east-1"
        },
        "republish": {
         "roleArn": "arn:aws:iam::your-aws-account-id:role/rulesengine_action_dy
namoDB",
          "topic": "my_topic_rule_name"
        }
      }
    ],
    "ruleDisabled": false
  }
}
```

Functions

There are several built-in functions that you can use in the select or where clauses of SQL expressions

Function	Description
abs(number)	Returns the absolute value of a number
accountId()	Returns the account ID of the MQTT client sending the message Returns undefined if message didn't come from MQTT
cos(number)	Returns the cosine of a number
asin(number)	Returns the arcsine of a number
atan(number)	Returns the arctangent of a number
bitand(number1, number2)	Returns the value of the parameters after a bitwise AND operation is applied
ceil(number)	Returns a number rounded up to the nearest whole integer
chr(number)	Returns the ASCII character represent by a number
clientId()	Returns the client ID of the MQTT client sending the message Returns undefined if message didn't come from MQTT
concat(string1, string2)	Returns the concatenation of the two strings
cosh(number)	Returns the hyperbolic cosine of a number
endswith(input, suffix)	Returns true if *`input`* ends with substring *`suffix`*
exp(number)	Returns e to the power of a number
floor(number)	Returns a number rounded down to the nearest whole integer

Function	Description
ln(number)	Returns the natural logarithm of a number
log(n, m)	Returns the logarithm of n base m
lower(string)	Returns string with all characters converted to lower case
lpad(string, n)	Adds n spaces to the left side of string
ltrim(string)	Removes all white-space from the left side of string
md2(string)	Returns the MD2 hash value of string
md5(string)	Returns the MD5 hash value of string
mod(m, n)	Returns the remainder of m divided by n
nanvl(value, default)	Returns the value if it's non-null, otherwise it returns default
power(m, n)	Returns m raised to the nth power
regexp_matches(input, pattern	Returns true if regular expression *pattern* is matched by *input*
regexp_replace(source, pattern, replacement)	Performs a regular expression replacement on source
regexp_substr(source, pattern)	Returns the first sub-string of source matched by pattern
remainder(m, n)	Returns the remainder of m divided by n
replace(source, sub-string, replacement)	Returns source with all occurrences of sub-string replaced by replacement
round(number, precision)	Returns a number rounded to precision decimal places If *precision* is 0, then this rounds to the nearest whole number
rpad(string, n)	Adds n spaces to the right side of string
rtrim(string)	Removes all whitespace from the right side of string
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
sin(number)	Returns the sine of a number
sinh(number)	Returns the hyperbolic sine of a number
sqrt(number)	Returns the square root of a number
startswith(input, prefix	Returns true if *input* starts with *prefix*.
tan(number)	Returns the tangent of *number*

Function	Description
tanh(number)	Returns the hyperbolic tangent of *number*
traceId()	Returns the trace ID of the MQTT the message The trace ID is used internally and should be considered to uniquely identify a message within an account Returns undefined if message didn't come from MQTT
trunc(number, precision)	Returns the number truncated to precision decimal places
upper(string)	Returns string with all characters converted to upper case
sha1(string)	Returns the sha1 hash value of string
sha224(string)	Returns the sha224 hash value of string
sha256(string)	Returns the sha256 hash value of string
sha512(string)	Returns the sha512 hash value of string
rand()	Returns a random number between 0 and 1
newuuid()	Returns a new random 20-byte UUID
timestamp()	Returns the current Unix timestamp as observed by the current server

JSON Extensions

We created some extensions to standard SQL syntax so that it's easier to work with nested JSON objects easier.

The '.' Operator

The '.' operator functions identically to standard SQL and JavaScript.

The ".." Operator

Like the single dot ("."), the double dot ("..") accesses members in inner objects. However, unlike the single dot, the double dot searches arbitrarily deep to find a match.

Query	JSON	Matches?
SELECT * FROM 'topic' Where a..b = 3	{"a":{"b":3}}	Yes

Query	JSON	Matches?

SELECT * FROM 'topic'
 	
Where a..b = 3
	{
	
	"a":{
	
	"foo":{
	
	"bar":{"b":3}
	
	}
	}
	
	}
		Yes

SELECT * FROM 'topic'
 	
Where a..b = 3
	{
	
	{"a":[
	
	{"b":3},{
	
	{"b":4}
	
	]}
	}
		Yes

Caveats When Multiple Matches Are Found

When double dots ".." are used inside the WHERE clause, it means "if any matching value satisfies this boolean expression". For instance, in the third row of the example table above, the expression "a..b" matches 2 values (3 and 4). The full expression "a..b = 3" is evaluated as "result1 = 3 OR result2 = 3". When used inside the SELECT clause as a single value, only the last match is used. Again, in the third row of the example table above, if the SQL were rewritten as

```
SELECT a..b FROM 'topic'
```

This would produce this JSON result:

```
{"b":3}
```

The "*" Wild card Operator

This functions exactly like the * in SQL. It's only used in the SELECT clause and creates a new JSON object.

Action Templates

Action templates are strings that supply data at runtime. They can be used when specifing the values in an action string. The syntax is "${<any SQL expression>>}" and can contain any expression that's valid inside a WHERE or SELECT clause like JSON properties, comparisons, calculations and functions. For instance, a JSON snippet:

```
"hashKeyValue":"${jsonProperty1}-${topic()}"
```

Working with the AWS Icebreaker Thing Registry

AWS Icebreaker enables bi-directional communication between Internet connected things (sensors, actuators, devices, applications, etc.) and the cloud. These things may be connected to the Internet intermittently or not at all. The AWS Icebreaker Thing Registry is a list of all things (devices, sensors, apps, etc) that have been registered with AWS Icebreaker. It provides operations to add, update, retrieve (describe), delete, attach and detach credentials (principals). In addition to principals the AWS Thing Registry contains metadata about your things. This could be a model number, serial number, or any kind of string data. In general the data stored about a thing in the registry will be static or change infrequently. You can interact with the AWS Icebreaker Thing Registry by the RESTful API or the AWS CLI. Metadata is added to a thing using the --attribute-payload option with the create-thing or update-thing commands. The --attribute-payload option is followed by a JSON string containing up to 3 key/value pairs. You can use attributes to filter or search the registry for specific things or types of things.

Create a Thing

To add a thing to the registry use the create-thing command as shown in the following example command:

```
aws iot --endpoint-url
https://t71u6yob51.execute-api.us-east-1.amazonaws.com/beta create-thing
--thing-name  "thingName"--attribute-payload
"{\"attributes\":{\"key1\":\"value1\”}, {\"key2\":\"value2\"},
{\"key3\":\"value3\"}}"
```

Describe a Thing

To retrieve the information about a thing, use the describe-thing command as shown in the following example command:

```
aws iot --endpoint-url
https://t71u6yob51.execute-api.us-east-1.amazonaws.com/beta describe-thing
--thing-name "thingName"
```

Update a Thing

To update an existing thing, use the update-thing command. Currently you can add string information to a thing using key value pairs 3 arguments as shown in the following example command:

```
aws iot --endpoint-url
https://t71u6yob51.execute-api.us-east-1.amazonaws.com/beta update-thing
--thing-name “thingName” --attribute-payload
"{\"attributes\":{\"key1\":\"value1\”}, {\"key2\":\"value2\"},
{\"key3\":\"value3\"}}"
```

List all Things in an AWS Account

To list all things in your AWS account, use the list-things command as shown in the following example command:

```
aws iot --endpoint-url
https://t71u6yob51.execute-api.us-east-1.amazonaws.com/beta list-things
```

You can filter things based on attributes by using the `--attribute-name` and `--attribute-value` options.

Attach a Thing to a Principal

To attach a credential to a thing, use the `attach-thing-principal` command as shown in the following example command:

```
aws iot --endpoint-url
https://t71u6yob51.execute-api.us-east-1.amazonaws.com/beta
attach-thing-principal --thing-name "thingName" --principal "principal-arn"
```

Detach a Thing from a Principal

To detach a credential from a thing, use the detach-thing-principal command as shown in the following example command:

```
aws iot --endpoint-url
https://t71u6yob51.execute-api.us-east-1.amazonaws.com/beta
detach-thing-principal --thing-name "thingName" --principal "principal"
```

List Principals Attached to a Thing

To list the credentials attached to a thing, use the `list-thing-principals` command as shown in the following example command:

```
aws iot --endpoint-url
https://t71u6yob51.execute-api.us-east-1.amazonaws.com/beta list-thing-principals
--thing-name "thingName"
```

List Things Attached to a Principal

To list the things attached to a credential, use the `list-principal-things` command as shown in the following example command:

```
aws iot --endpoint-url
https://t71u6yob51.execute-api.us-east-1.amazonaws.com/beta list-principal-things
--principal "principal"
```

Delete a Thing

To delete a thing from the registry use the delete-thing command as shown in the following example command:

```
aws iot --endpoint-url
https://t71u6yob51.execute-api.us-east-1.amazonaws.com/beta delete-thing
--thing-name "thingName"
```

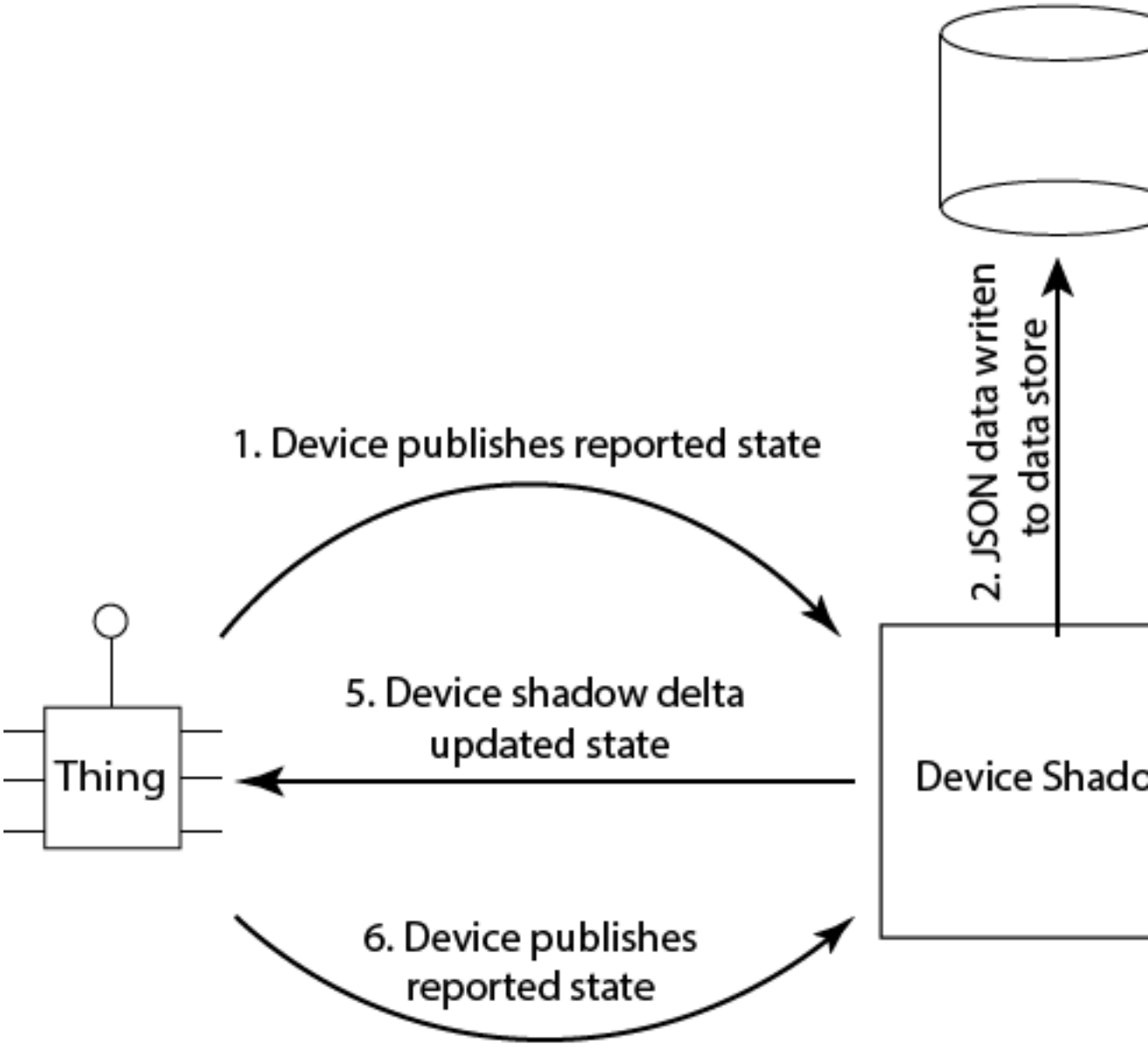
Working with Thing Shadows

AWS Icebreaker enables bi-directional communication between Internet connected things (sensors, actuators, devices, applications, etc.) and the cloud. These things may be connected to the Internet intermittently or not at all. Thing Shadows allow applications to easily interact with the things connected to Icebreaker.

Thing Shadows is a JSON document store that contains the state of a thing. The documents stored are identified and accessed by a developer provided “thing name”, a string identifier that is unique to your AWS account. You can update the document using the Thing Shadow RESTful API or publishing MQTT messages to Thing Shadows special $shadow topics. Devices and applications can subscribe to Thing Shadows special $shadow topics to be notified when a Thing Shadows document is updated. In this way, applications can update the state of a document and devices can immediately sync those changes.

Data-flow in Thing Shadows

To understand how data flows into and out of a thing shadow, think about the following scenario: You have a resource-constrained device (in this case an Internet connected light bulb) that communicates with thing shadows over MQTT. You also have an app running on a mobile phone that controls the light bulb and communicates with thing shadows over an REST API. The following diagram shows the flow of data in this type of scenario.



1. After your light bulb is registered with Icebreaker it can start sending MQTT messages containing its current state. These messages are published on a topic called $shadow/beta/state/thing-name. For example, say we have an Internet connected light bulb called mylight. The light bulb would publish a message to $shadow/beta/state/mylight. The contents of the message would look like the following:

```
{
     "state": {
         "reported": {
             "color":"red"
         }
     }
}
```

2. Thing shadows receives and stores this information:

```
{
    "state": {
       "reported":{
          "color":"red"
       }
    },
    "version":”1”,
    "metadata": {
        "reported": {
            "color":<time-stamp>
        }
    }
}
```

The state data is given a version number and metadata is generated. The metadata contains a timestamp indicating when each JSON property was last updated.

3. Now your mobile device app needs to display the current state of the light bulb. So it makes an HTTP request to get the current state of the light:

```
GET /things/mylight/state
```

Thing shadows returns the entire JSON document (containing thing state, thing metadata, and version):

```
{
    "state": {
        "reported":{
            "color":"red"
        }
    },
    "version":”1”,
    "metadata": {
        "reported": {
            "color":<time-stamp>
        }
    }
}
```

4. Now you use the app to change the color of the light bulb to green. The app makes an HTTP request setting the desired color of the light to green:

```
POST /things/mylight/state
```

```
{
    "state":{
        "desired":{
            "color":"green"
        }
    }
}
```

Thing shadows receives this information and updates the JSON in the shadow store for the light bulb which now looks like this:

```
{
    "state": {
        "desired": {
            "color":"green"
        },
        "reported":{
             "color":"red"
        },
        "version":”1”,
        "metadata": {
            "desired": {
                 "color":<time-stamp>
            },
            "reported": {
                 "color":<time-stamp>
            }
        }
}
```

Because the shadow for the light bulb has changed, thing shadows publishes a message on `$aws/things/mylight/shadow/update/accepted`

```
{
     "state":{
         "desired":{
             "color":"green"
         },
         "reported":{
             "color":"red"
         },
         "version":"2",
         "metadata": {
             "reported":{
                  "color":<time-stamp>
             },
             "desired":{
                 "color":<time-stamp>
             }
         }
    }
}
```

5. Thing shadows detects the difference between the reported and desired states and publishes a message on `$aws/things/mylight/shadow/update/delta`:

```
{
     "state":{
         "color":"green"
     },
     "version":"3",
     "metadata":{
         "color":<time-stamp>
     }
}
```

This information only contains the difference between the reported and desired states.

6. The light bulb receives the MQTT message because it is subscribed to `$aws/things/mylight/shadow/update/delta` and changes the color of the light. When that occurs, it publishes its current state to `$aws/things/{thingName}/shadow/update` :

```
{
     "state": {
          "reported":{
              "color":"green"
          }
     }
}
```

7. Since the data in thing shadows has changed, a message is published to `$aws/things/mylight/shadow/update/delta`:

```
{
     "state":{
         "desired":{
             "color":"green"
         },
         "reported":{
             "color":"green"
         },
         "version":"3",
         "metadata": {
             "reported":{
                   "color":<time-stamp>
              },
              "desired":{
                  "color":<time-stamp>
              }
          }
     }
}
```

The phone app can subscribe to this topic to receive confirmation that its request was successfully completed. Additionally thing shadows publishes a message to `$aws/things/mylight/shadow/update/delta`:

```
{
    "state":{
        "version":"3",
```

```
    }
}
```

This only contains the version number because there is no difference between the reported and desired states within the thing shadow.

Thing Shadow Document

A Thing Shadows state document contains the following parts:

- state: the state of the thing. State has two sub-nodes:
  - desired: the desired state of the thing. Applications can write to this portion of the document to update the state of a thing
  - reported: the reported state of the thing. Devices write to this portion of the document to report their new state
- metadata: This section contains information about the data stored in the state section of the document. Right now we are just storing timestamps of each attribute in the state section. This allows developers to know when the state was updated.
- version: the document version. Every time the document is updated this version number is incremented. This version can be used for optimistic locking by developers to ensure the document they are updating is equal to the document they last read.
- clientToken: a string sent by the client and echoed by Icebreaker. You can use this to determine whether a recieved response was triggered by your own request or someone else publishing on a topic.

Thing Shadows supports full JSON so a developer can store values, objects, and arrays in the reported and desired sections of the thing shadow document.

The following is an example thing shadow document:

```
{
     "state" : {
         "desired" : {
             "color" : "RED",
             "sequence" : [ "RED", "GREEN", "BLUE" ]
         },
         "reported" : {
              "color" : "GREEN"
         }
      },
     "metadata" : {
          "desired" : {
              "color" : {
                  "timestamp" : <time-stamp>
              },
              "sequence" : {
                   "timestamp" : <time-stamp>
              }
          },
          "reported" : {
              "color" : {
                   "timestamp" : <time-stamp>
              }
          }
```

```
     },
     "version" : 10
}
```

Retrieving a Thing Shadow

You can retrieve the current, complete thing shadow document by issuing a HTTP GET to the documents RESTful URL, publishing an empty message to the ./get topic, or using the AWS CLI `get-thing-shadow`. This returns the entire document plus the delta between the desired and reported state. For example:

Using HTTP GET

```
{
    "state" : {
        "desired" : {
          "lights": { "color": "RED" },
            "engine" : "ON"
        },
        "reported" : {
             "lights" : { "color": "GREEN"  },
        "engine" : "ON"
        }
   },
   "metadata" : {
        "desired" : {
             "lights": { "color": { "timestamp" : <time-stamp> },
             "engine" : { "timestamp" : <time-stamp> }
        },
       "reported" : {
             "lights" : { "color" : { "timestamp" : <time-stamp> }  },
             "engine" : { "timestamp" : <time-stamp> }
       }
   },
   "version" : 10
}
```

HTTP request:

```
GET https://g.us-east-1.pb.iot.amazonaws.com/things/{thingName}/shadow
```

HTTP response:

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

```
            "lights" : { "color": "RED"  }
        }
    },
    "metadata" : {
        "desired" : {
            "lights" : { "color": { "timestamp" : <time-stamp> },
            "engine" : { "timestamp" : <time-stamp> }
         },
         "reported" : {
             "lights" : { "color" : { "timestamp" : <time-stamp> }  },
             "engine":{ "timestamp" : <time-stamp> }
         },
         "delta" : {
             "lights" : { "color": { "timestamp" : <time-stamp> }  }
         }
    },
    "version" : 10
}
```

Retrieving a Thing Shadows document requires a policy that allows the calling principal to perform the `"iot:GetThingShadow"` action. Thing Shadows accept two forms of authentication: sigv4 with IAM credentials or TLS mutual authentication with an Icebreaker certificate.

The following snippet is an example policy that allows a caller to retrieve a Thing Shadows document using an Icebreaker certificate for authentication:

```
{
   "Version": "2012-10-17",
   "Statement": [{
      "Effect": "Allow",
      "Action": "iot:GetThingShadow",
      "Resource": ["arn:aws:iot:us-east-1:<accountId>:thing/<thing>"]
   }]
}
```

The following snippet is an example policy that allows a caller to update a Thing Shadows document using IAM credentials for authentication:

```
{
   "Version": "2012-10-17",
   "Statement": [{
      "Effect": "Allow",
      "Action": ["iot:GetThingShadow"],
      "Resource": ["arn:aws:iot:us-east-1:<accountId>:thing/<thing>"]
   }]
}
```

Using MQTT

If Icebreaker recieves an empty message on the ./get topic, it publishes the entire state document to the $aws/things/{thing-name}/shadow/get/accepted topic.

Using AWS CLI

You can also retrieve the Thing Shadows document using the `get-thing-shadow` CLI command. The command takes a single parameter: `--thing-name` that specifies the thing you are retrieving.

Example command line:

```
aws iot-data --endpoint-url https://g.us-east-1.pb.iot.amazonaws.com
get-thing-shadow --thing-name myRGBLight output.txt
```

Updating Thing Shadows

You can update a Thing Shadows document by using the AWS CLI, the REST API or by publishing a message on the `$aws/things/{thingName}/shadow/update` topic. If the specified thing does not exist, it is created.

Using the AWS CLI

You update the Thing Shadows document using the `update-thing-shadow` CLI command. The command parameters are:

--thing-name : the thing name
--payload : the JSON object that specifies the new state

Example command line:

```
aws iot-data --endpoint-url https://g.us-east-1.pb.iot.amazonaws.com
update-thing-shadow --thing-name myRGBLight --payload "{ \"state\": {
\"desired\": { \"color\": \"RED\" }, \"reported\": { \"color\": \"GREEN\" } }
}" output.txt
```

Using the REST API

To update a Thing Shadow using the REST API, send an HTTP POST to: `https://g.us-east-1.pb.iot.amazonaws.com/things/{thingName}/shadow` with the JSON document in the HTTP request body. The JSON document should contain only the values you are adding or changing. For example:

```
{
   state:{
      "desired" : {
        "color" : {  "r" : 10 },
        "engine" : "ON"
   }
}
```

Using MQTT

Publish a message on the `$aws/things/{thingName}/shadow/update` topic.

For example:

```
{
   "state" : { "desired" : {  "color" : "RED"  }, "reported" : { "color" :
"GREEN" } }
}
```

Update topics

Icebreaker publishes on the following special topics when updating shadows:

- $aws/things/{thingName}/shadow/update/accepted - Icebreaker publishes a message on this topic when updating a thing shadow
- $aws/things/{thingName}/shadow/update/rejected - Icebreaker publishes a message on this topic when a thing shadow cannot be updated

Deleting a Thing Shadow

You can delete a thing using the AWS CLI `delete-thing-shadow` command, sending an HTTP DELETE request, or publishing an empty message or a JSON message with “clientToken” set on the AWS/things/{thing-name}/shadow/delete topic.

clientToken is set if the sender is interested in getting a response that contains which client deleted the message. If the publisher choses to publish JSON, it must be valid JSON.

Using the AWS CLI

Use the delete-thing-shadow command:

```
aws iot-data delete-thing-shadow --thing-name {thingName} output.txt
```

Sending an HTTP Request

Send an HTTP DELETE request to
https://https://g.us-east-1.pb.iot.amazonaws.com/things/{thing-name}/shadow

Using MQTT

Publish either an empty message or a message with clientToken set on the $aws/things{thingName}/shadow/delete topic. The message should look like the following. An empty message:

```
{}
```

Or a message with clientToken set

```
{
    "clientToken":"<unique-client-token>:
}
```

Delete topics

Icebreaker publishes on the following special topics when deleting shadows:

- $aws/things/{thingName}/shadow/delete/accepted - Icebreaker publishes a message on this topic when deleting a thing shadow
- $aws/things/{thingName}/shadow/delete/rejected - Icebreaker publishes a message on this topic when a thing shadow cannot be deleted

Deleting an attribute from a Thing Shadow

You can delete an attribute from a thing shadow by setting the associated field to null. Null is treated as delete by Thing Shadows and any field with a value of null is removed from the document. For example:

Initial state:

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

Message payload:

```
{
    "state" : {
         "desired" : null,
         "reported: {
             "engine" : null
         }
    }
}
```

Result:

```
{
    "reported" : {
        "lights" : { "color" : "GREEN" }
    }
}
```

MQTT Topics Used by Thing Shadows

Thing Shadows uses the following MQTT topics to allow things and applications to publish their current state and request the current state be updated:

update

`$aws/things/{thingName}/shadow/update` : a thing (device or app) can publish on this topic to update the JSON state document

You can specify a version attribute in the JSON document to specify which version of the JSON document you are attempting to update. When a version attribute is present in an MQTT message, Icebreaker only processes the update if the message version matches the latest version of the state document in Icebreaker. The following example shows a message with a version attribute:

```
{
   "state" : {
      "desired" : {
         "color" : "RED",
         "temperature" : 55
      },
      "reported" : {
         "color" : "GREEN",
         "temperature" : 35
      }
   },
   "version" : 42,
   "clientToken" : "UnqiueForTheClient"
}
```

accepted

`$aws/things/{thingName}/update` : Icebreaker publishes to this topic when it accepts a change. The message body contains the updated state. For example, if the following JSON document was published on `$aws/things/{thingName}/update`:

```
{
   “state” : { “desired”: { “color” : “RED” } }
}
```

`Icebreaker publishes the following JSON on $aws/things/{thingName}/update/accepted`:

```
{
   “state” : { “desired”: { “color” : “RED” } }
   “version” : 43,
   “metadata” : { “desired” : { “color” : { “timestamp” : 100 } } }
}
```

rejected

`$aws/things/{thingName}/shadow/update/rejected` : Icebreaker publishes on this topic when it rejects a change. For example:

```
{
   “code” : 400,
   “message” : “invalid json",
```

```
   "timestamp" : 100123,
   "clientToken" : "1234",
   "version": 12
}
```

The following parameters are used in every ./rejected message:

- `code` - an HTTP response code that indicates specific types of errors
- `message` - a non-standardized text message that provides additional human readable error information
- `timestamp` - indicates the time when the message was generated by Icebreaker
- `clientToken` - only present if a client token was used in the preceding publish to ./update
- `version` - sent to allow for devices to re-sync on the current shadow document version

delta

`$aws/things/{thingName}/shadow/update/delta` : Icebreaker publishes on this topic when it accepts a change and the JSON state document contains different values for desired and reported states.

The following rules determine when messages are published to .shadow/update/delta and what is included in the messages:

- Messages published on the ./update/delta topic contain only the desired attributes that differ between the reported and desired sections. The messages contains all of these attributes regardless of whether these attributes were contained in the current ./update message or whether they were already stored in Icebreaker. Attributes that do not differ between reported and desired are not included in messages published on ./update/delta
- If an attribute is in the reported section but has no equivalent in the desired section, it is not included in messages published on .update/delta
- If an attribute is in the desired section but has no equivalent in the reported section, it is not included in messages published on ./update/delta
- If an attribute is deleted from the reported section but still exists in the desired section it is included in messages published on ./update/delta

> **Note**
> The state and reported sections are attributes contained within the JSON document

The following examples illustrates a scenario where a device publishes its state and then an app updates the device state:

Icebreaker contains to following JSON document for a thing:

```
{
   "state" : { "color" : "RED" }
   "version" : 10,
   "timestamp" : 1234567,
   "metadata" : { "color" : { "timestamp" : 100 } }
}
```

A thing publishes the following message on ./update:

```
{
   "state" : { "reported": { "color" : "RED" } }
}
```

A controller app attempts to update the thing state:

```
{
   "state" : { "desired": { "color" : "GREEN" } }
}
```

Icebreaker publishes the following on ./update/delta:

```
{
   "state" : { "color" : "GREEN" }
   "version" : 12,
   "timestamp" : 2345679
   "metadata" : { "color" : { "timestamp" : 2345678 } }
}
```

delete accepted

$aws/things/{thingName}/shadow/delete/accepted - Icebreaker publishes a message on this topic when deleting a thing shadow

delete rejected

$aws/things/{thingName}/shadow/delete/rejected - Icebreaker publishes a message on this topic when a thing shadow cannot be deleted

Using the Thing SDK

Download AWS Icebreaker Thing SDK for C using the following AWS CLI command:

```
aws s3 cp s3://Icebreaker-private-beta/aws-iot-thing-sdk-c.tar
aws-iot-thing-sdk-c.tar
```

Identify the thing or device you want to connect to Icebreaker and confirm that you can SSH into it or otherwise access it for development purposes.

Copy the tarball on to the thing and expand it using command:

```
tar -xf tarball-path
```

This will create 4 directories:

1. *iot_src* - the AWS IoT SDK source files
2. *sample_app* - the sample application targeted to an embedded Linux platform with access to OpenSSL
3. *mqtt_paho_emb* - the Paho MQTT Embedded C client
4. *tls* - TLS certificates directory

The SDK contains a sample (subscribe_publish_sample.c) that shows how to write an app that sends and receives MQTT messages to/from Icebreaker. Open the subscribe_publish_sample.c file and set the following variables:

certDirectory – the name of directory where you certificates are located
cafileName - the name of the root CA certificate
clientCRTName – the name of the Icebreaker-generated certificate clientKeyName - the name of your private key

Build the SDK using the included makefile:

```
make –f LinuxPahoOpensslMakefile.mk
```

Run the sample by typing:

```
./subscribe_publish_sample
```

The sample uses default values for host name, port, and cert locations. Optionally, you can specify host name, port, and cert locations on the command line:

```
./subscribe_publish_sample –host “g.us-east-1.pb.iot.amazonaws.com” –p 8883 –c
“cert-dir”
```

Note
Ensure egress to port 8883 is allowed on your network.

Connecting to Icebreaker

The `MQTTConnectParams` struct is used to specify the connection settings. The `ib_mqtt_connect()` function takes a `MQTTConnectParams` instance and connects to Icebreaker.

Server Certificate Validation

As part of the TLS handshake the device (client) needs to perform server certificate validation. One step in this validation is to determine if the current time (on the device) falls within the current time range of the certificate. Devices must have an accurate time set to properly validate the server certificate. We do not recommend disabling server certificate validation as a workaround for devices that do not have an accurate time set. Time can be determined through NTP or other supported mechanism.

Subscribing to a Topic

To subscribe to a topic, create an instance of `MQTTSubscribeParams`, specify the callback handler to call when a message is received, the topic to subscribe to, and the QoS value required. Pass the `MQTTSubscribeParams` to the `ib_mqtt_subscribe()` function.

Publishing a Message to a Topic

Create an instance of `MQTTMessageParams` and specify the quality of service (QoS) requested, whether the message should be retained, and the message payload. Create an instance of `MQTTPublishParams` specifying the topic and the `MQTTMessageParams` struct that contains the message. Call the `ib_mqtt_publish()` function passing in the `MQTTPublishParams` instance. The sample is a simple single-threaded app so periodic calls to `ib_mqtt_yield()` are required to give the SDK and the MQTT client time to process messages.

Thing SDK Porting Guide

The Icebreaker C SDK is designed in layers to allow porting to specific hardware and operating systems. The SDK wraps the Paho embedded MQTT client making communications with the Icebreaker service easier.

Porting Points

The C SDK requires OS/hardware-specific resources including: TLS secured sockets and timers. The Linux example included in the SDK uses OpenSSL and Linux timers. The TLS and timer interfaces are defined in header files included with the SDK. The porting points are below the Paho client.

Threading

The Linux sample is single threaded to make it easier to understand. Your implementation can use multiple threads. While connected the SDK requires periodic calls to `iot_mqtt_yield()`. This can be done from a background thread. MQTT publishing and subscribing can be decoupled from the main application thread using a message queue or mailbox and consumer pattern.

Note
Further details are yet to be added to the Icebreaker C SDK porting guide.

Using Icebreaker with the AWS SDKs for Java and Javascript

Download the Icebreaker SDK for Java using the following AWS CLI commands:

```
aws s3 cp s3://icebreaker-private-beta/aws-java-sdk-1.10.20.zip
aws-java-sdk-1.10.20.zip
```

You can enable AWS SDK for JavaScript support using the AWS Icebreaker service model files.

1. Locate the service model file(s). It is the same file that you use to enable AWS CLI support. For Icebreaker, you can download using the command:

   “aws s3 cp s3://icebreaker-private-beta/aws-iot-cli.zip aws-iot-cli.zip”

   There is a JSON file each under iot and iot-data directories. iot is for control plane operations. iot-data is for data plane operations.
2. In your code,

```
// Enable AWS SDK for JavaScript support using a service model file
var myService = new AWS.Service({apiConfig: require('./path/to/service-mod
el.json'), endpoint: "service enpoint"});

// Look up operation names and parameters in the model file under 'shapes'.
Use camel case.
myService.someOperation(params, function(err, data) {
   console.log(err, data);
});
```

The following code illustrates using JavaScript in a Lambda function. Be sure to allow your Lambda execution role to invoke “iot:*” actions.

```
var aws = require('aws-sdk');
var iot = new aws.Service({
```

```
    apiConfig: require('./iot-service-model.json'),
    endpoint: "t71u6yob51.execute-api.us-east-1.amazonaws.com/beta" });

var iotData = new aws.Service({
    apiConfig: require('./iot-data-service-model.json'),
    endpoint: "g.us-east-1.pb.iot.amazonaws.com" });

exports.handler = function(event, context) {
    // Exercising an IoT Data operation as an example
    var params = { "topic" : "foo/bar", "payload" : "hello world" };
    iotData.publish(params, function(err, data) {
        console.log(err, data); context.done(); });
};
```

Troubleshooting with logs

You can have AWS Icebreaker stream logs to Amazon CloudWatch Logs. The logs are useful for troubleshooting. For example, if a rule did not trigger as expected or a connection error occurs while communicating with the gateway, you can look at the logs and get more information on the root cause such as an authorization failure. Use the following process to enable logging:

Create an IAM Role for Logging

Create an IAM role that AWS Icebreaker can assume to stream logs to CloudWatch Logs.

Set the following 'assume role policy document' (i.e. "Trust Relationship") to a file:

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

Call create-role:

```
aws iam create-role --role-name icebreaker-logging-role
--assume-role-policy-document file:/<path-to-file>/trust-policy-file
```

Save the role ARN from the command output, you will need it when creating a rule.

Grant Permissions to the Role

Grant the role permissions to stream logs to CloudWatch Logs. Save the following to a file:

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

```
    "Statement": [{
        "Effect": "Allow",
        "Action": [
            "logs:CreateLogGroup",
            "logs:CreateLogStream",
            "logs:PutLogEvents",
            "logs:PutMetricFilter",
            "logs:PutRetentionPolicy",
            "logs:GetLogEvents",
            "logs:DeleteLogStream"
        ],
        "Resource": ["*"]
    }]
}
```

Call create-policy and specify the file path from above:

```
aws iam create-policy --policy-name icebreaker-logging-policy --policy-document
file://<file-path>/<IAM-policy-document>
```

Attach the policy you just created to the role using the attach-policy-role command:

```
aws iam attach-role-policy --role-name "icebreaker-logging-role" --policy-arn
"<policy ARN>"
```

Add Logging Role in Icebreaker

Add Logging Role in Icebreaker using the following command:

```
aws iot --endpoint-url
https://t71u6yob51.execute-api.us-east-1.amazonaws.com/beta add-logging-role
--logging-role-payload roleArn=<role ARN>
```

Use Logs

As an example, configure a rule and publish a message that should trigger the rule. Within a few minutes, you will see logs related to the rule in CloudWatch Logs. Go to the CloudWatch Logs console, click on log group "AWSIotLogs", within the log group, click on a log stream with your account id as the log stream name and view the logs.

Icebreaker Appendix

TCP Ports Used

The following table lists the TCP ports used by the various supported protocols

TCP Port Changes

Port	Old Usage	New Usage
443	Secure MQTT	HTTPS sigV4
8443	HTTPS SigV4	HTTPS Client Cert
8192	HTTP Client Cert	HTTPS Client Cert
8883	Secure MQTT	Secure MQTT

MQTT Protocol Implementation Notes

Within Icebreaker, subscribing to a topic with Quality of Service (QoS) 0 means a message will be delivered zero or more times. A message may be delivered more than once. Messages delivered more than once may be sent with a different packet ID. In these cases, the `DUP` flag is not set.

When responding to a connection request, the broker sends a `CONNACK` message. This message contains a flag indicating if the connection is resuming a previous session. The value of this flag may be incorrect when two MQTT clients connect with the same client ID simultaneously.

When a client connects to the broker using a client ID that another client is using, a `CONNACK` will be sent to both clients and then the existing client will be disconnected.

When a client subscribes to a topic, there may be a time delay between the time the broker sends a `SUBACK` and the time the client starts receiving new matching messages.

Authorizing an IAM User to Use Icebreaker

When your account is white listed, by default the root user in your account has access to the icebreaker-private-beta bucket in S3, which contains documentation, CLI models, SDKs, and other utilities. To enable other IAM users in your account to access the bucket, attach the following IAM policy to the IAM users:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Sid": "icebreaker1",
        "Effect": "Allow",
        "Action": [
            "s3:List*",
            "s3:Get*"
         ],
         "Resource": [
            "arn:aws:s3:::icebreaker-private-beta",
            "arn:aws:s3:::icebreaker-private-beta/*"
         ]
     }]
}
```

Moreover, to be able to access Icebreaker commands and configure rules, an IAM user requires the following additional privileges:

```
{
    "Version": "2012-10-17",
    "Statement": [{
         "Effect": "Allow",
          "Action": [
              "iot:*",
              "iam:AttachRolePolicy",
              "iam:CreatePolicy",
              "iam:CreatePolicyVersion",
              "iam:CreateRole",
              "iam:PassRole"
           ],
            "Resource": ["*"]
      }]
}
```