AWS Whitepaper

Designing MQTT Topics for AWS IoT Core

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Designing MQTT Topics for AWS IoT Core: AWS Whitepaper

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

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

Table of Contents

..... v
Abstract and introduction i
Abstract 1
Introduction 1
MQTT communication patterns 2
Point-to-point 2
Broadcast 3
Fan-In 4
Communication workflows 6
MQTT design best practices 7
General best practices 7
Best practices for telemetry 9
Using AWS IoT Basic Ingest for telemetry 9
Using the MQTT topics for telemetry 11
Best practices for commands 12
Using the AWS IoT Shadow for commands 12
Using AWS IoT Jobs for Commands 14
Using the MQTT topics for commands 15
Applications on AWS 18
MQTT command topics example 18
Command request to generate a smart lock code 18
Command processing on the smart lock 19
Command response delivered to the mobile client 21
MQTT telemetry topics example 21
Local telemetry from occupancy sensor to AWS IoT Greengrass 22
AWS IoT Greengrass telemetry from Edge to cloud 23
Best practices for using MQTT topics in the AWS IoT Rules Engine 24
Rules Engine integration with telemetry topics 24
Rules Engine integration with command topics 25
Tracking success of commands 25
Aligning Rules Engine capabilities with MQTT topics 26
Conclusion 28
Further Reading 29
Document History and Contributors 30

Document History 30
Contributors 30
Notices 31

This whitepaper is for historical reference only. Some content might be outdated and some links might not be available.

Designing MQTT Topics for AWS IoT Core

Publication date: **December 8, 2021** (the section called "Document History")

Abstract

This whitepaper focuses on best practices for MQTT topic design in Amazon Web Services (AWS) Internet of Things (IoT). It covers how developing an optimal MQTT topic schema can improve the overall architecture and efficiency of your IoT solutions. It does so by providing greater visibility into cloud to device communication, providing more fine-grained security permissions, and enhancing integration options with other AWS IoT Core services (such as the AWS IoT Rules Engine, AWS IoT Device Shadow, AWS IoT Device Management, and AWS IoT Analytics). This whitepaper is intended for technical architects, IoT cloud engineers, and application architects. This paper assumes that the reader understands fundamental MQTT concepts and terminology.

Introduction

AWS IoT Core supports Message Queuing Telemetry Transport (MQTT), a widely adopted lightweight messaging protocol designed for constrained devices. MQTT participants receive information organized through MQTT topics. An MQTT topic acts as a matching mechanism between publishers and subscribers. Conceptually, an MQTT topic behaves like an ephemeral notification channel.

For AWS IoT, one of the first considerations when using MQTT is the design strategy of your MQTT topics. MQTT topics must balance current device communications, cloud side operations, and future device capabilities. Therefore, it can be challenging to design an ideal MQTT topic structure that creates enough of a schema to enforce least privilege communication but does not create a rigid structure that makes it challenging to support future device deployments.

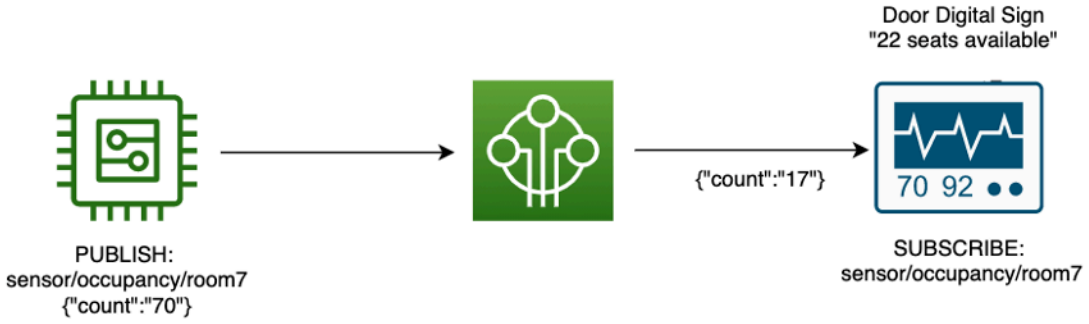
This document provides you with MQTT topic design best practices and guidance. It outlines a set of commonly-used MQTT topic structures that can be implemented to solve various device message patterns, then applies several example design patterns using different AWS IoT services.

MQTT communication patterns

IoT applications support multiple communication scenarios, such as device-to-device, device-to-cloud, cloud-to-device, and device-to-or-from-users. Although the range of patterns can significantly vary, a majority of MQTT communication models derive from three MQTT patterns: point-to-point, broadcast, and fan-in.

Point-to-point

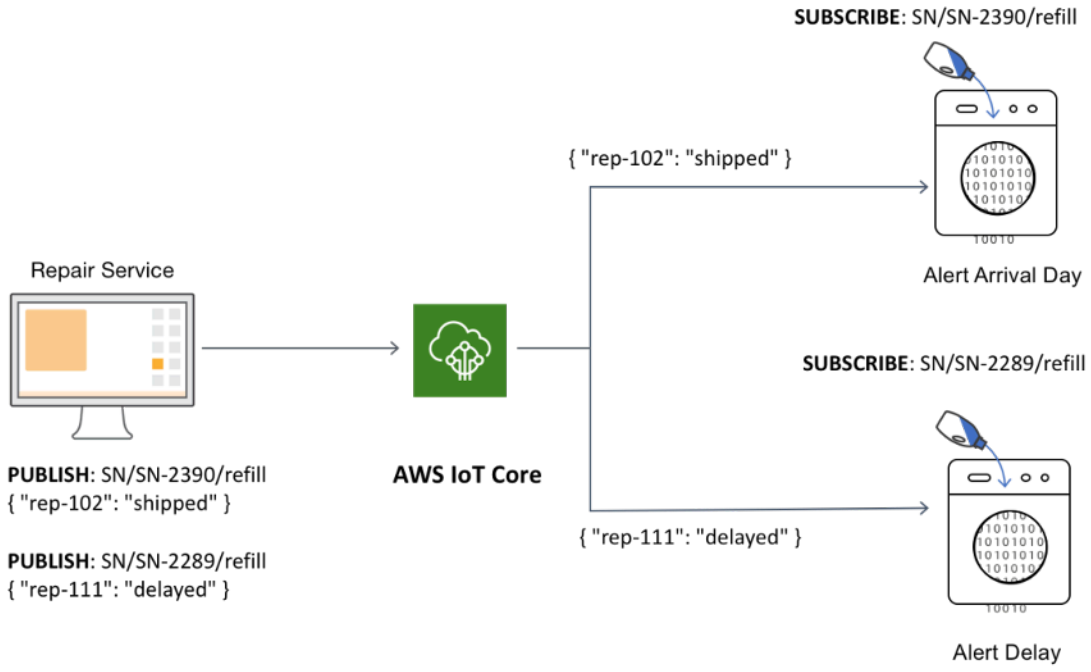
A point-to-point communication pattern is one of the basic building blocks of how devices commonly send and receive messages in MQTT. Two devices use a single MQTT topic as the communication channel. The device that receives the event subscribes to an MQTT topic. The thing that sends the message publishes to the same known MQTT topic. This approach is common in smart home scenarios where an end user receives updates about the thing in the home. In the following example, the room occupancy publishes a message on a topic subscribed to by an application running on the digital display outside the screening room.



One-to-one messaging in point-to-point communication

Point-to-point communication is not limited to one-to-one communication between devices. Point-to-point is also used in one-to-many communication where a single publisher can publish to individual devices using a different MQTT topic per device.

This approach is common in notification scenarios where an administrator sends distinct updates to specific devices. In the following example, the repair service uses a set of point-to-point communications to programmatically loop through a list of appliances and publish a message.

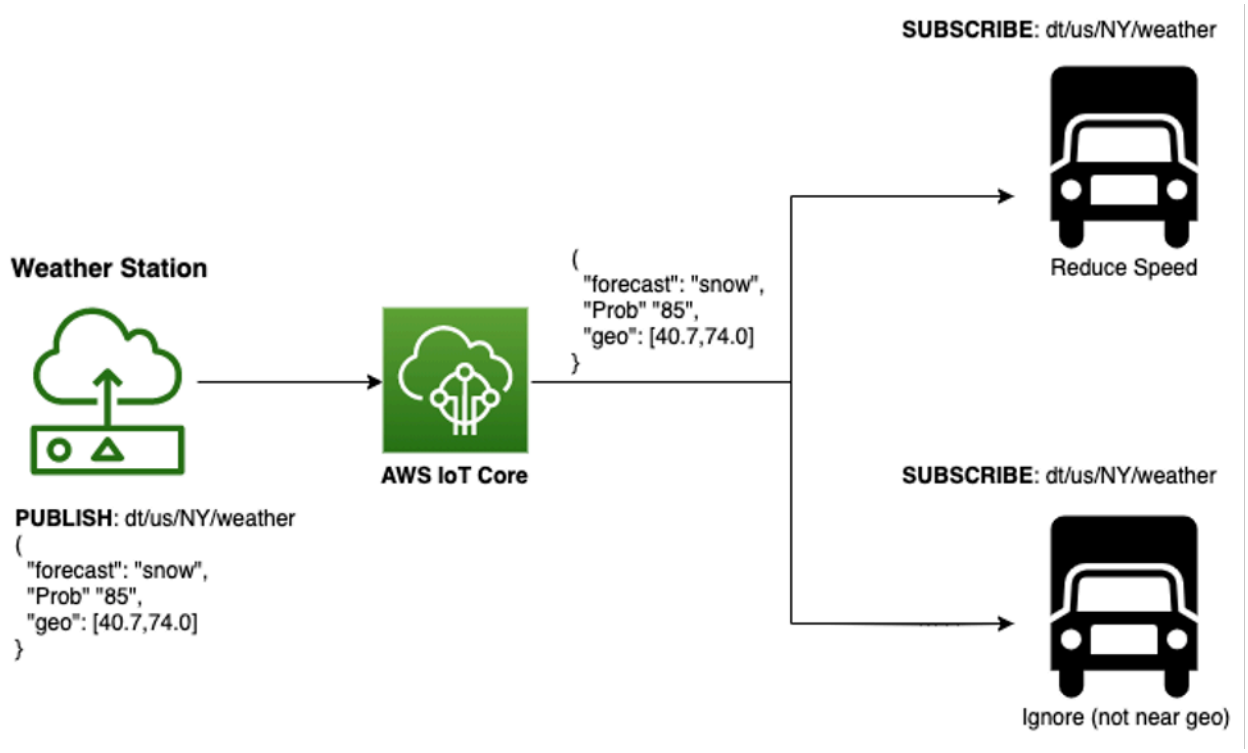


One-to-many messaging in point-to-point communication

Broadcast

Broadcast patterns are used for one-to-many messaging. The broadcast pattern sends the same message to a large fleet of devices. In a broadcast, multiple devices subscribe to the same MQTT topic, and the sender publishes a message to that topic. A typical use of a broadcast pattern is to send a notification to devices based on the category or group of the device. For example, a weather station transmits a broadcast message based on a topic based on its geolocation.

The following illustration depicts an example where a broadcast pattern sends a message on a weather topic that all delivery vehicles in the state subscribe to. The message includes weather conditions and detailed location coordinates. Based on the current location of the vehicle, it can ignore the message or take some action.



One-to-many messaging in broadcast communication

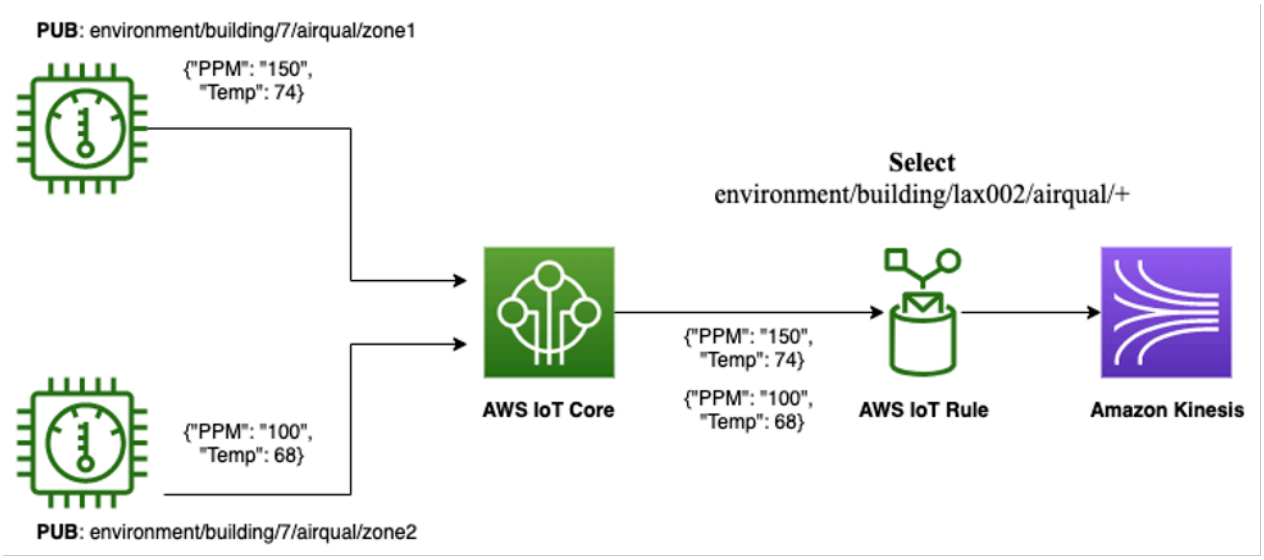
Fan-In

The fan-in pattern is a many-to-one communication pattern and can be thought of as the reverse of the broadcast pattern.

Multiple devices publish on a shared or similar topic with a single subscriber to that topic. With the fan-in pattern, the subscriber may use wildcards as the publishers all use a similar but unique MQTT topics. The fan-in pattern is commonly used to aggregate telemetry data.

In the following example, each device publishes to an MQTT topic containing a known group identifier. The AWS IoT Rules Engine uses a wildcard subscription to receive the messages and route them to an Amazon Kinesis stream. Specifically, the air quality sensors publish on a fan-in topic associated with a specific building (LAX002). The administrative system receives all updates for the building using an MQTT wildcard (+).

Rule: Select environment/building/lax002/airqual/+



Many-to-one communication in a fan-in pattern

When devices communicate via the cloud using MQTT, avoid using the fan-in pattern to a single subscribing end device, because this routing may hit a non-adjustable limit on a single device MQTT connection. Instead, use the fan-in pattern to route a large fleet of messages to your IoT application via the AWS IoT Rules Engine.

For large scale fan-in scenarios, combine the Rules Engine with a wildcard subscription pattern and a Rules Engine action to route to Amazon Kinesis Data Streams, Amazon Data Firehose, or Amazon Simple Queue Service (Amazon SQS).

Communication workflows

The three common communication workflows are device-to-device, device-to-cloud, and cloud-to-device. Each workflow determines the topic structure of topic hierarchy. In the case of device-to-device, MQTT topics should contain identifiers for either the sender or receiver of a message. For device-to-cloud, MQTT messages should include information about the target application. The target application is responsible for augmenting any MQTT messages with internal metadata about the device. Last, for cloud-to-device communication, MQTT messages should contain session information for tracking acknowledgment of any critical messages.

MQTT design best practices

General best practices

Although there are numerous combinations of IoT communication patterns that share common approaches, there are several best practices that apply to any message pattern irrespective of how a device is publishing or receiving a message. This section articulates several overall best practices for you to review and implement as you design your MQTT topic structures.

Review the AWS IoT Core default service limits. Design your communication pattern so that it aligns with any adjustable IoT service limits. AWS IoT has several adjustable and non-adjustable limits associated with using the AWS IoT Core service. As part of your topic review, review the AWS IoT limits, and ensure your MQTT topic and device communication do not conflict with any service limits.

The maximum number of forward slashes (/) in the MQTT topic name for AWS IoT Core is seven. You should not prefix the topic with a forward slash as it counts towards the topic levels and may introduce confusion when building AWS IoT policies. This excludes the first three slashes in the mandatory segments for Basic Ingest topics `$AWS/rules/rule-name/`.

The topic passed to AWS IoT Core when sending a publish request can be no larger than 256 bytes of UTF-8 encoded characters. This excludes the first three mandatory segments for Basic Ingest topics `$AWS/rules/rule-name/`.

Define a consistent naming standard for MQTT topic levels. Since MQTT topics are case sensitive, it is important to use a standard set of naming conventions when designing MQTT topics. For this reason, customers should only use lowercase letters, numbers, and dashes when creating each topic level. Customers should avoid camel casing and using hard to debug characters such as spaces. Publish Topic names cannot contain wildcards (# , +). Topics that start with $ are reserved by AWS IoT Core. They are not supported for publishing and subscribing except for using the specific topic names defined by AWS IoT Core services (for example, the AWS IoT Device Shadow service).

Ensure MQTT topic levels structure follows a general to specific pattern. As topic scheme flows left to right, the topic levels flow general to specific. For example, an HVAC system is associated with an IoT platform named **hv100**, is located in the **basement** of building **bld1518**, and has a Thing Name of **hvac719**. The topic structure begins with the general group, in this case, the name

of the IoT platform, and ends with the most specific identity, the Thing Name. This example creates the following topic level structure:

```
hv100/bld1518/basement/hvac719
```

General ⟶ Specific

Include any relevant routing information in the MQTT topic. Relevant routing information includes, but is not limited to, the IoT application identifier, any groups the device may be a part of, such as installed location, and the unique identity of your IoT device. To continue with the previous HVAC system example, the MQTT topic `hv100/bld1518/basement/hvac719` includes all relevant routing information. Based on this MQTT topic, you can design a system that captures any data related to the entire application using the identifier, `hv100`, but also can target different areas of interest for subscribing to messages, such as the building location.

Prefix your MQTT topics to distinguish data topics from command topics. Make sure that your MQTT topics do not overlap between commands and data messages. By reserving the first topic level to denote data and command topics, you are more easily able to create fine-grained permissions using IoT policies, and monitor the status of commands and command responses separately from passive telemetry commands. For example, use the AWS IoT Device Shadow service for tracking reported and desired states, and use a separate data topic for passive, real-time telemetry data.

Document proposed MQTT topic structures as part of your operations practice. The document should include all topics available for publishing, subscribing, or receiving data, along with the intended producers and consumers of the data. Review the document to ensure it adheres to any AWS IoT limits, internal security requirements, and any application use cases.

Include the Thing Name of the device in any MQTT topic the device uses for publishing or subscribing to its data. To track messages destined for a particular device, include the Thing Name as part of any MQTT message that is published by the device or sent to a specific device. The Thing Name should appear near or at the end of the MQTT topic after any routing topic information.

Include additional contextual information about a specific message in the payload of the MQTT message. This contextual information includes, but is not limited to, a session identifier, the requestor identifier, logging information, or the return topic on which a device is expecting to receive a response. Although the MQTT 3.1.1 specification does not require specific payload attributes, we recommend you include relevant tracking information inside of the MQTT payload. By creating a standard structure including fields such as session identifier and success or error

codes, you can more easily analyze trends in device behavior. Standardizing the communication schema also strengthens a shared vernacular of device use cases across IoT teams.

Avoid MQTT communication patterns that result in a sizeable fan-in scenario to a single device. Some AWS IoT limits cannot be raised as part of a limit increase and frequently correlate to per device actions, such as maximum publish-in on a single MQTT connection. Do not allow a single device to subscribe to a shared topic that is being published to by a large number of other devices. By avoiding this pattern, you are more likely to avoid hitting a single connection device limit, particularly a throughput per connection per second limit.

Never allow a device to subscribe to all topics using #, and only use multi-level wildcard subscriptions in IoT rules. By using multi-level wildcards, you can create unintended consequences when you inadvertently add new topics to the hierarchy that may not be intended for that particular device. Instead, reserve use of multi-level wildcards as part of the IoT rules engine, and use single level wildcards (+) for device subscriptions.

Best practices for telemetry

Telemetry is read-only data that is transmitted by the device and processed in the cloud. It follows the device-to-cloud pattern along with the fan-in pattern for communication.

Telemetry does not require an acknowledge message back from the MQTT broker, beyond optionally setting a higher quality of service (QoS) level. Since telemetry is a passive activity, the MQTT topic for telemetry should not overlap with any MQTT topics for active workflows, such as command and control messages.

A telemetry topic supports more complex devices that publish telemetry on behalf of other devices, such as an edge gateway or a mesh network with a single coordinator.

In AWS IoT, you have the ability to use different AWS IoT services to support telemetry communication patterns. We recommend that you use a combination of AWS IoT Basic Ingest and standard MQTT topics to support your telemetry use cases.

Using AWS IoT Basic Ingest for telemetry

Basic Ingest optimizes data flow for high volume data ingestion workloads by removing the pub/sub Message Broker from the ingestion path. As a result, you have a more cost-effective option to send device data to other AWS services while continuing to benefit from all the security and data processing features of AWS IoT Core.

In cases where devices do not require the publish and subscribe functionality of the Message Broker, Basic Ingest enables you to send data to cloud services through the Rules Engine.

Basic Ingest is an ideal use case for telemetry when the only interested subscriber for an IoT message is your backend IoT application. Basic Ingest uses a reserved MQTT topic structure that is associated to a particular AWS IoT Rule.

A device can publish to the reserved topic associated to a specific AWS IoT Rule, and Basic Ingest will trigger the IoT Rule for the matching Rule Name. The MQTT topic structure for Basic Ingest follows a similar syntax as the following example:

```
$aws/rules/<rule-name>/<optional-customer-defined-segments>
```

Where the field `rule-name` matches the name of the AWS IoT Rule that should be invoked, and `optional-customer-defined-segments` includes any additional topic levels a customer may use for routing or logging as part of the AWS IoT Rule Action.

Best practices for using AWS IoT Basic Ingest

Include any additional routing information after the rule name in the Basic Ingest MQTT Topic. As a best practice, AWS recommends you use the optional segments that can appear after the rule name in the MQTT topic to include relevant additional information that can be used by the AWS IoT Rule for features such as Substitution Templates, IoT Rule SQL Functions, and Where Clauses. Similar to the overall best practice for MQTT topics, any fields that can be used for IoT Rule evaluation, such as application Identifier or device Identifier, should be appended to the end of the Basic Ingest topic. The following example would be publishing to an AWS IoT Rule named BuildingSecurity followed by customer defined segments:

```
$aws/rules/BuildingSecurity/buildings/warehouse4/section6/motion
```

Choose short, descriptive rule names for Basic Ingest. When AWS IoT Rules are used directly by devices via Basic Ingest, AWS recommends that you ensure the rule name follows MQTT topic best practices for consistency. Since the rule will link directly to a reserved MQTT topic, ensure that the rule name is short, descriptive of the underlying use case of the rule, and adheres to the syntax rules described in the section General best practices.

Using the MQTT topics for telemetry

In addition to using Basic Ingest, you can also leverage traditional MQTT topics. These types of MQTT messages are passive AWS IoT data that may be subscribed to by other devices now or in the future.

For example, a device that sends its current status may expect its data to be routed not only to your internal application but also to a user who needs the device's current status. To achieve this level of flexibility, you can use standard MQTT topics for sending and receiving telemetry.

MQTT telemetry topic syntax

The following example and sections provide the MQTT topic structure for telemetry:

```
dt/<application>/<context>/<thing-name>/<dt-type>
```

dt: Set prefix that refers to the type of message. For a telemetry topic, we use `dt`, short for data. All telemetry topics use this top-level prefix for an application. By reusing the same value for telemetry, you can identify the intent of a message by referring to the initial prefixed value. In this case, any `dt` topic is a telemetry topic.

application: Identifies the overall IoT application associated with the device. Commonly used application attributes include device hardware version or an internal identifier for a cloud application that is the primary ingestion point for a message. The IoT application is associated with an internal name for your overarching IoT product or relates specifically to the type of hardware of your device. Because the application topic portion correlates to a group of device messages and is immutable, the application prefix portion of the telemetry MQTT topic is placed immediately after the `dt` message type.

context: Single or multiple levels of additional contextual data about the message a device is publishing. Contextual information is related to information that is set during device provisioning. For example, contextual information in a factory setting could include the current physical location of a device in the facility. Another example of contextual information is a group-id in the MQTT topic. The group-id denotes when multiple devices have an inherent relationship based on specific attributes, such as buying a package of smart light bulbs to control lighting in a room. The group-id enables numerous devices to coordinate activities as a single unit.

thing-name: Identifies which device is transmitting a telemetry message.

dt-type (optional): Associates a message with a particular subcomponent of a device, or for edge gateways, any downstream devices. A complex device often has multiple subcomponents with specific tasks, such as sensors, actuators, or separate system on chips (SOCs). The `dt-type` allows you to associate each subcomponent of a particular device to an individual MQTT topic. One example of this is a subcomponent that measures geolocation and direction of a vehicle. That subcomponent would have a `dt-type` value of `geo` to distinguish its geolocation messages from other components of the car, such as the accelerometer.

Best practices for commands

In IoT applications, command topics are used to control a device remotely and to acknowledge successful command executions. Unlike telemetry, command topics are not read-only. Commands are a back and forth workflow that can occur between two devices or between the cloud and devices. Because commands are actionable messages, isolate the MQTT topic for command messages from telemetry topics.

Several services are available for you to implement command and control operations on AWS IoT. With the capability to store the desired and reported states in the cloud, the AWS IoT Shadow is the preferred AWS IoT service for implementing individual device commands. AWS IoT Device Jobs should be used for fleet-wide operations as it provides extra benefits, such as Amazon CloudWatch metrics for Job tracking, and the ability to track multiple in-transit Jobs for a single device. You can use a combination of the AWS IoT Shadow, AWS IoT Jobs, and standard MQTT topics to support your command use cases.

Using the AWS IoT Shadow for commands

The AWS IoT Device Shadow service acts as a state intermediary, allowing devices and applications to retrieve and update a device's shadow state. You can use the shadow to get and set the state of a device over MQTT or HTTP. The shadow includes the following individual state properties that support command and control:

desired state. Applications that have permissions to send commands to a device can write the requested state changes to the desired portion of the shadow document. By updating the desired state, the AWS IoT Shadow service stores the desired state change in the AWS cloud and then sends an MQTT message to the device using a reserved shadow topic. When a device receives a shadow request, it can execute the changes required from the desired state.

reported state. The reported state of the AWS IoT Thing's shadow stores the last published attributes published by a device. Devices write to this portion of the document to record their

new state while applications read this portion of the Shadow to determine the state of a specific device. Because shadows are stored by AWS in the cloud, they can collect and report device state data from apps and other cloud services whether the device is connected or not. Use the AWS IoT Shadow in situations where a command persists for later use, even if the device is currently offline. For example, if a GPS system is sent a new destination through the Shadow desired state but is not immediately reachable, the new coordinates remain in the GPS IoT Shadow. Once the GPS system regains connectivity, it can actively request its last shadow state and retrieve the new coordinates. The shadow is also ideal for storing the last reported state for attributes of the device.

Best practices for using the AWS IoT Classic Shadow or Named Shadows

The AWS IoT is a mechanism for command and control along with storing the reported state of a specific device.

The following list of best practices offers advice on maximizing the efficiency of commands through the shadow:

IoT devices should not share shadows. To separate commands for each device, make sure that each device has permissions to its own shadow and that devices do not share a single shadow. For complex scenarios, like edge gateways or large device assets with multiple subcomponents, the primary asset should use multiple IoT Things and shadows individually associated with the downstream devices.

Consider using Named Shadows to create logical groups of properties. You can create a unique access policy for each Named Shadow, therefore controlling what applications or services can view or update that group of properties. An example of this would be the device management team viewing the firmware, battery health, or WiFi signal strength but not having access to the data being published by the sensors on said devices.

Use the shadow for state or commands that have a medium to low transaction per second (TPS). The shadow is an ideal fit for infrequent updates that occur in minutes, hours, or days as the shadow publishes on additional topics to acknowledge an action was successful. For a high frequency or throughput commands that do not require the updates to the shadow consider publishing to a MQTT command topic.

Use the shadow for storing status metrics of a device. Store informational data about the current health of the device including, but not limited to, connectivity, the status of device sensors and control units, and any error information about those subcomponents. If you know the current status of the device, you can make actionable decisions during command requests.

Use the AWS IoT Device Shadow service to catalog the current firmware version. The shadow is an ideal location for a device to report the firmware version installed on the hardware. The firmware should be a simple attribute, such as a field that highlights the `major.minor.patch` version of the service.

Use the optional `clientToken` field with AWS IoT Device Shadow service updates to track the sender of a shadow message. The `clientToken` is a field in the Shadow that enables a subscriber to associate the responses with requests in your MQTT application. If a device sets the `clientToken` during a shadow update request, the AWS IoT Shadow service includes that same `clientToken` in the associated shadow output events.

Using AWS IoT Jobs for Commands

AWS IoT Jobs is a service that allows you to define a set of remote operations that are sent to and executed on one or more things connected to AWS IoT. For command use cases, Jobs allows applications to run tasks that require executing multiple steps. An AWS IoT Job contains instructions that the thing must run to complete its transaction. AWS IoT Jobs are the recommended feature for fleet-wide operational tasks, such as software updates, that are only executed by trusted administrators of the entire IoT application.

Best practices for using AWS IoT Jobs

Use thing groups to organize devices for AWS IoT Jobs. Create multiple thing groups organized by common device attributes, such as the current firmware version, hardware version, or deployment environments (for example, staging or production). Thing groups should also have common hierarchical structures, such as business units or locations. During deployments, you can use thing groups as the deployment target for a specific IoT job.

Use staged rollouts to deploy commands using Device Jobs. Device Jobs are the ideal solution for delivering fleet-wide operations to devices. Create multiple smaller deployments first, to subsets of the fleet, letting the devices apply your changes, and then rolling out the commands to a greater number of devices over time. By allowing changes to progress over weeks and months, you can have more confidence that there are fewer unforeseen issues and you can react more quickly if there is an issue during an earlier rollout.

Using the MQTT topics for commands

MQTT command topic syntax

In some scenarios, you may want to design your command communication using the standard MQTT publish and subscribe model. These types of situations may occur when a device must execute a command that is temporal (that is, can only be processed at this current type and should fail if the device is unavailable) or run a single command across multiple devices simultaneously.

It is also possible in a brownfield environment where a device may be incapable of leveraging higher-level AWS IoT services. You may also require the flexibility to choose your own set of MQTT topics to define commands to and responses from devices.

In cases where you are using a separate set of command topics, follow similar best practices for MQTT command topics as described for telemetry. A command topic should have flexibility for complex devices that publish or relay commands to other devices. Command topics should also provide visibility into essential attributes. MQTT command topics should be designed in a way that can answer operational questions based on the MQTT topic and payload:

- Who is the originator of the command?
- Who is the intended receiver of the command?
- Was the command processed successfully?
- What is the current status of the command?
- If the command was not processed successfully, what is the error?

In addition to these questions, you may also want to determine when a command was requested, when a device responded, and to monitor the state of any single request among the fleet in the cloud.

When you design MQTT topics for command requests, follow this structure:

```
cmd/<application>/<context>/<destination-id>/<req-type>
```

Since commands are two-way communication patterns, design a similar MQTT topic structure for responding to commands, such as the following:

```
cmd/<application>/<context>/<destination-id>/<res-type>
```

Because telemetry topic design is similar to command topic design, this section provides only the portions of the IoT topic for command requests and responses that differ.

cmd: Prefix that refers to the type of message. Command topics use `cmd`, which is short for command. By prefixing all commands with `cmd` and all telemetry with `dt`, telemetry and commands are isolated on separate MQTT topics.

req-type: Classifies the command. For simple request and response patterns, the req-type attribute should be a single command request static value such as `req`. In cases of limited command types, the MQTT message includes the additional data in the payload.

In more complex systems, where a device is orchestrating multiple devices, actuators, or subcomponents, the `req-type` attribute relates to each subcomponent available to receive commands. For example, if a device is mobile, you may want to steer the device remotely or receive navigational information about the device's surroundings. This type of subcomponent would have a `req-type` of `nav` where commands are sent steering in single or multiple planes.

destination-id: Identifies the destination device or application for this message. By including the `destination-id`, the target device can subscribe to its own set of command topics and receive any command requests.

res-type: Denotes command responses and identifies responses that are related to a previously sent command. The `res-type` enables a single device to use one single-level wildcard subscription for all incoming command acknowledgments. If a device has limited commands, the response topic can use a static field, such as `res`.

MQTT command payload syntax

In addition to creating a clear MQTT topic structure for commands, make sure that you generate a schema for message payloads. MQTT payload information is parsed by the receiving device or IoT application, to inform it of any additional logic it may need to complete its operation. For MQTT commands, include the following fields with the command message payload:

session-id: Identifies a unique session. The requestor generates the `session-id` for the command and includes it in the request payload. The response topic uses the session-id upon command completion. By using a `session-id`, the AWS IoT Rules Engine can store and track the status of commands and determine if a request is still in transit, successful, or in error. Devices can also keep track of in-transit requests when communicating with multiple devices.

response-topic: In a command, there is a request for an action to happen and then a response that indicates the status of the command (successful or error). To avoid hard-coding response topics, we recommend that for any MQTT command, the command request payload includes a field that has a response topic. The device publishes its response payload using the response topic. For example, consider the following command topic:

```
cmd/security/device-1/cert-rotation
```

In the payload of this request, the IoT application includes a field that denotes where the device (`device-1`) should send its response and a session identifier for tracking. See the following example for this command's payload structure:

```
{
   "session-id":"session-820923084792",
   "res-topic":"cmd/security/app1/res"
}
```

Applications on AWS

The following sections provide use cases for implementing MQTT topic best practices using AWS IoT.

MQTT command topics example

For a smart door lock application, a user must be able to submit a command to the lock that initiates a temporary key to be issued for an approved visitor. The temporary key consists of a TTL, code, and information about the authorized user.

The ability to create a temporary key allows another individual to open the lock for a specified period. This use case would apply in scenarios such as visiting family member arriving at the home while the primary owner is at work.

This scenario assumes the following details:

- The homeowner has a mobile device ID of `mobile-1`
- The approved visitor has a mobile device ID of `mobile-2`
- The smart lock is installed as part of a group of other locks in the home. The set of locks has a context for `groupId` where `groupId` equals `group-3`
- The smart lock for the front door has a `lockId` of `lock-1`
- The smart lock hardware has a series number of `series100`. The series is a unique identifier for this product version.

Command request to generate a smart lock code

The primary owner first publishes a command to the lock requesting a temporary access code created for an approved visitor. The command payload includes the identification of the homeowner's mobile device, a randomly generated session identifier for tracking the current request, an action field that contains the type of command, and a topic field. The smart lock uses the topic in the topic field for publishing its response back to the homeowner.

PUBLISH: cmd/series100/group-3/lock-1/credentials
PAYLOAD:

```
{
   "clientId":"mobile-1",
   "sessionId":"0193-0428",
   "topic":"cmd/series100/mobile-1/res",
   "action": {
             "type":"generate-passcode",
             "uid":"visitor-1"
    }
}
```

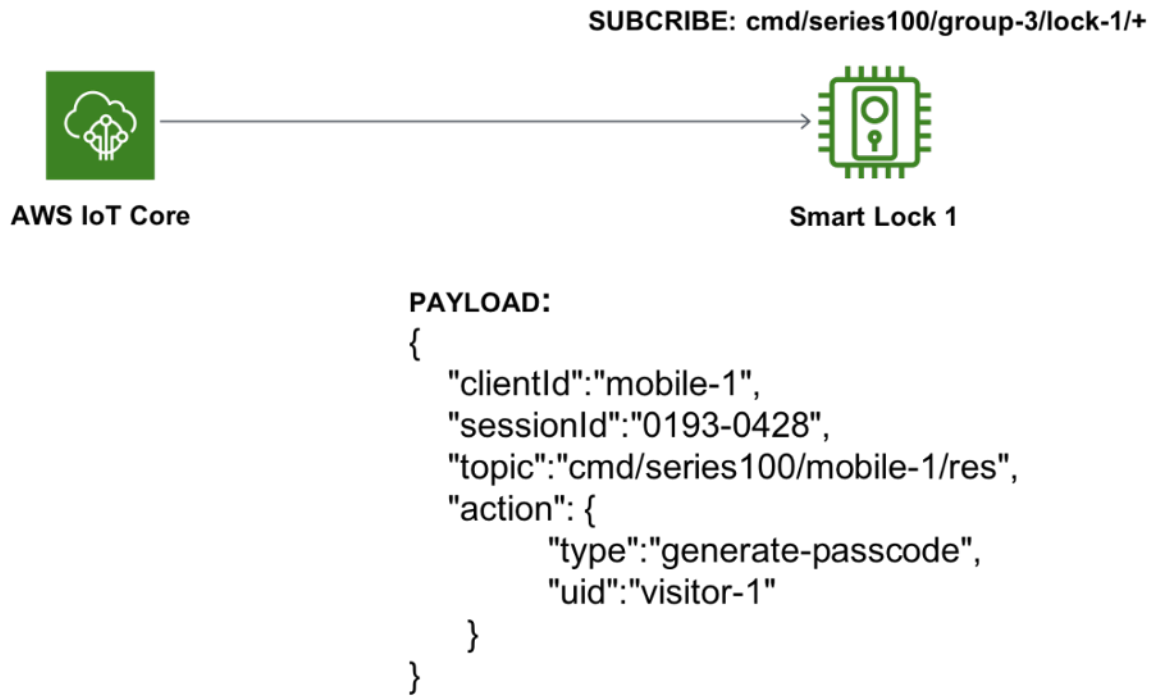
Mobile user requests temporary credentials for the front door

Augment MQTT messages using any internal standards for requests, responses, and telemetry. For example, by adding in the requestor of the command in the payload, applications can specify different response topics based on the use case. If the homeowner requests a temporary lock but needs the response to reach all door locks in the home, the topic field could be changed to send to a group of devices.

Command processing on the smart lock

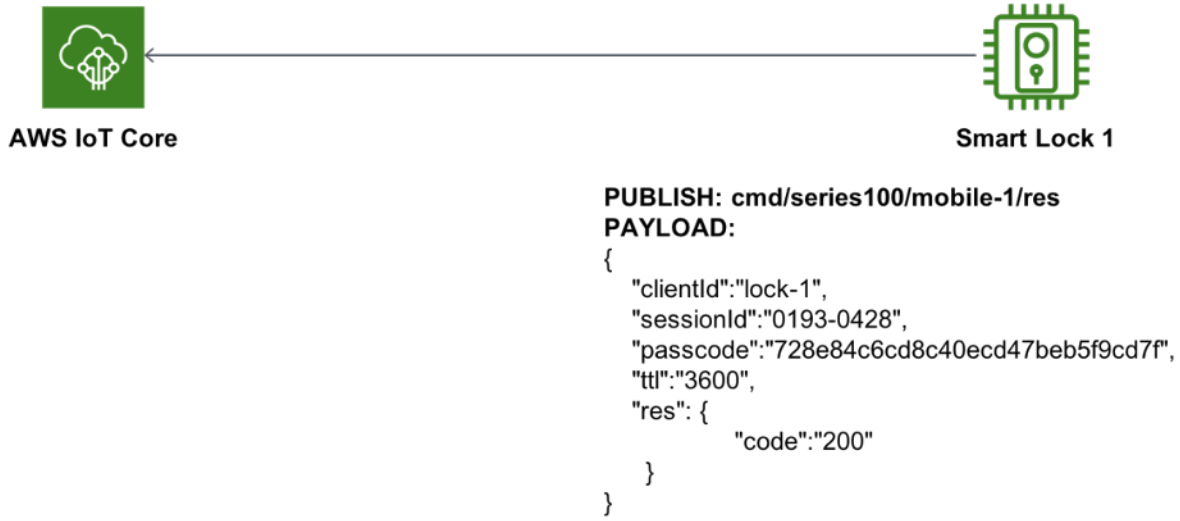
The smart lock receives the command message from the MQTT topic. By leveraging a consistent naming schema for commands in this application, the smart lock can ensure that it only receives commands on its specific command topic. This topic design also makes it possible for the lock to subscribe using a single level MQTT wildcard after its identifier.

The single level wildcard command is backward compatible as the IoT application adds new command types. The following example is a simplified example to show the topic structure. Consider using device attributes and policy variables restricting the topic(s) that a device can subscribe or publish to.



After receiving the MQTT payload, the smart lock parses the command request to determine what type of action to run. In this case, the command is related to credentials. The device also extracts the client ID, the session ID, and the response topic from the command payload.

Because a home can have multiple authorized homeowners, the client ID determines which homeowner has requested this change. In this example, the action field includes the type of credentials request, generate-password, and the associated user for the temporary key. Last, the device obtains the response topic field in the MQTT message and uses this information to publish its response.

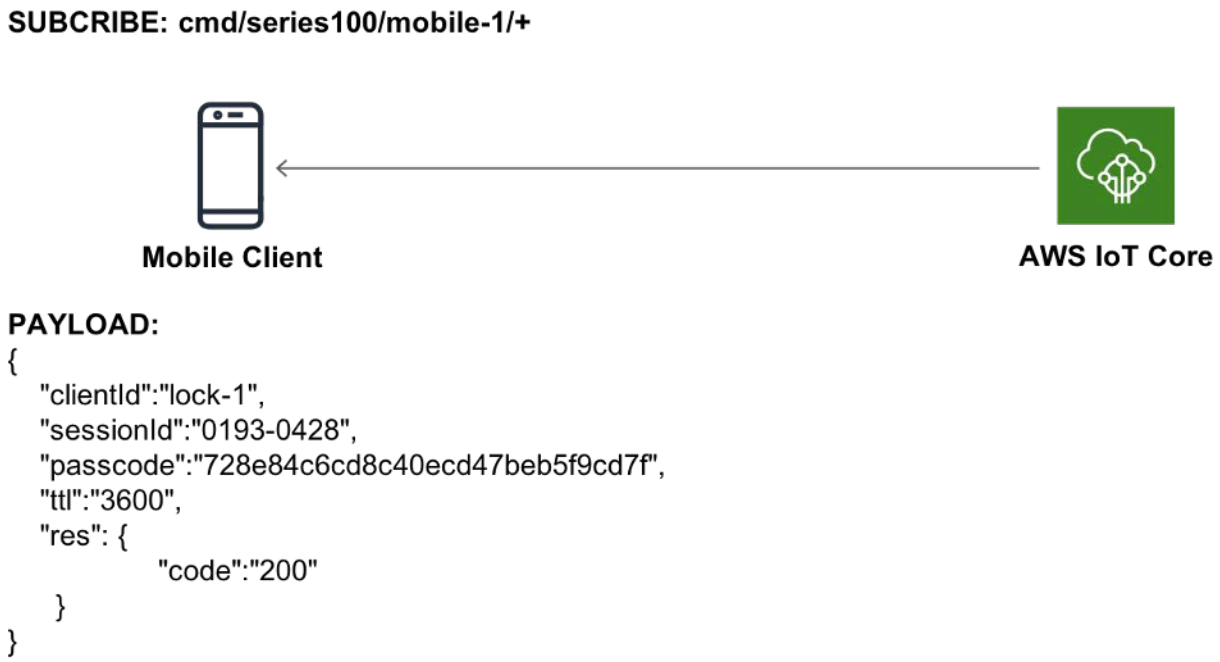


Smart lock response published to AWS IoT Core

Command response delivered to the mobile client

The homeowner's mobile client subscribes to command responses for any smart locks in the group. Whenever the mobile client receives a successful command response, the temporary code along with any additional authorization permissions can be processed on the device and simultaneously stored in the AWS Cloud. Later, the authorized visitor can use the temporary code along with additional security credentials, such as exchanging for OAuth credentials, proving local presence to the door, and so on, to apply the temporary code to the smart lock.

In this workflow, the application used a command topic for a single device. However, a similar workflow may be used to request commands for multiple devices in a group, such as locking all of the doors in the home.



Command response sent from AWS IoT Core to mobile client

MQTT telemetry topics example

This section is an example of aggregating telemetry from a set of occupancy sensors that are placed throughout a building to monitor room usage. The occupancy sensors communicate to a local gateway that is running AWS IoT Greengrass. AWS IoT Greengrass. AWS IoT Greengrass then delivers all sensor metrics on an MQTT topic to AWS IoT Core. Because this use case is focused on

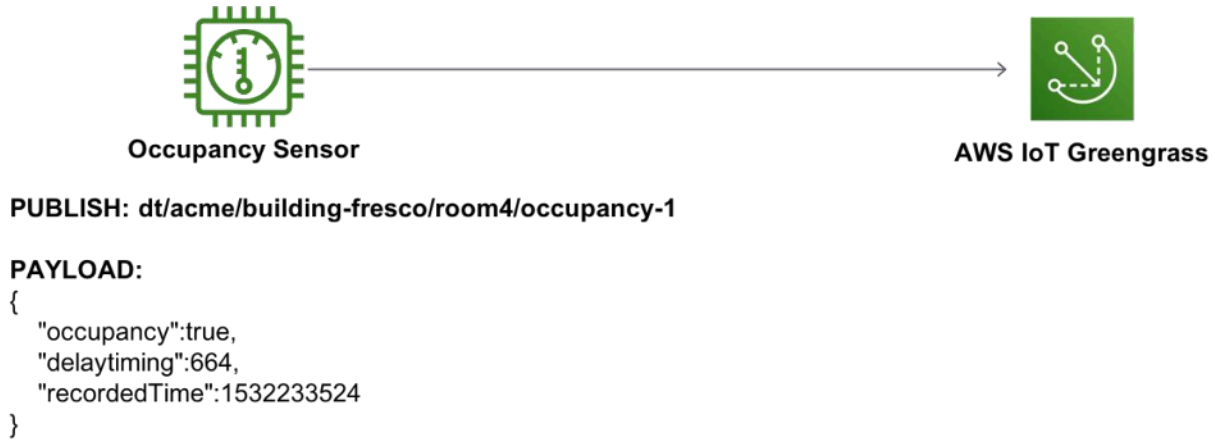
telemetry, a response topic is not needed between AWS IoT Greengrass and AWS IoT Core, either locally or upstream.

This scenario assumes the following details:

- The occupancy sensors have IDs of `occupancy-1` and `occupancy-2`
- The building is called `building-fresco`
- Each sensor is placed on a specific floor and within a specific room name in `building-fresco`.
- The AWS IoT Greengrass local gateway has a unique identifier of `gateway-1`
- The current building automation system correlates to an internal project called `acme`

Local telemetry from occupancy sensor to AWS IoT Greengrass

Each occupancy sensor publishes an occupancy reading once per minute and whenever a person enters or leaves the room. Because the occupancy sensors do not correlate precisely to the state of the device and instead refer to the state of the room, the sensor publishes the room status on a telemetry topic. The payload includes a timestamp, occupancy count, and any efficiency countdown for turning off lights if a room is vacant. The occupancy sensor uses an MQTT topic that includes the contextual information about the position of the sensor within the building and its associated project. A AWS IoT Greengrass core receives all occupancy sensor data locally.



Local occupancy sensor publishes sensor reading to AWS IoT Greengrass

AWS IoT Greengrass telemetry from Edge to cloud

In this example, the primary role of AWS IoT Greengrass is to aggregate the data from multiple occupancy sensors then send the data to AWS IoT Core. Because AWS IoT Greengrass is the local bridge to the cloud, AWS IoT Greengrass adds metadata to each sensor reading.

AWS IoT Greengrass adds building information to each message to show the overall usage of the building in 5-minute increments. AWS IoT Greengrass also augments the MQTT topic by including the appropriate application identifier, `acme`.

PUBLISH: dt/acme/building-fresco/gateway-1

PAYLOAD:

```
[ {
    "occupancy":true,
    "delaytiming":664,
    "recordedTime":1532233524,
    "room":4,
    "sensorId":"occupancy-1"

  },
  ...
]
```

AWS IoT Greengrass aggregates and augments telemetry, then forwards messages to AWS IoT Core

Best practices for using MQTT topics in the AWS IoT Rules Engine

The AWS IoT Rules Engine enables you to define how messages sent to AWS IoT Core can interact with AWS services. An AWS IoT rule consists of a SQL SELECT statement, a topic filter, and a rule action.

The SQL SELECT statement can extract data from incoming MQTT messages. The topic filter of an AWS IoT rule specifies which MQTT topics invoke an AWS IoT Rule Action.

The rules engine plays a pivotal role in intelligently directing messages to other AWS services or republishing to devices. AWS IoT rules support use cases, such as gathering operational metrics, data enrichment, data aggregation of device telemetry for analytics purpose, and for troubleshooting errors.

Rules Engine integration with telemetry topics

We recommend using a topic structure for telemetry similar to the following:

```
dt/<application-prefix>/<context>/<thing-name>/<dt-type>
```

The second field in the MQTT topic telemetry pattern defined as `application-prefix` represents an immutable, natural bifurcation between your devices in a fleet. A common attribute for the application is the device hardware version or the name of the IoT application. Using the telemetry MQTT structure, you can create an IoT rule to capture all telemetry associated with a specific application version:

```
{
    "sql":"SELECT *, topic(2) as applicationVersion, topic(3) as
contextIdentifier FROM 'dt/#'",
    "awsIoTSqlVersion":"2016-03-23",
    "ruleDisabled":false,
    "actions":[{
    ...
    }]
}
```

Because the MQTT topic structure mirrors a hierarchy, this rule can select different parts of the MQTT topic hierarchy and inject it into the new payload. These attributes provide further context as messages are processed and stored in other AWS services.

Rules Engine integration with command topics

The Rules Engine can be used to capture insight into the success or failure of commands, regardless of whether the commands are sent using the AWS IoT Device Shadow service, AWS IoT Jobs, or by using an MQTT command topic.

Tracking success of commands

The AWS IoT Rules Engine can be used to track the success rates of individual commands. The IoT rule extracts payload information, such as the session identifier; generates additional metadata in the rule select statement, such as creating a time to live; and temporarily stores the new message payload into a data store, such as Amazon DynamoDB.

The rule that follows mirrors a common implementation of this use case for an AWS customer. The IoT rule stores each session as an individual DynamoDB record and because the WHERE clause identifies this as an incoming command, the rule adds a literal value named `status` that marks the command as `In progress`.

```
{
"sql": "SELECT sessionId AS token,timestamp()/1000 as ttl, topicId AS responseTopic,
 clientId AS requestorID, action.type AS commandType, 'In PRogress' AS status FROM
 'cmd/series100/#' WHERE topic(5) == 'credentials'”
 "actions": [{
      "dynamoDBv2": {
          "roleArn": "arn:aws:iam::12345678:role/service-role/dynamoDBrole",
       "putItem":{
        "tableName": "command sessions table"
                 }
             }
       }]
```

As command messages are published to the topic matching the rule, the preceding rule maintains a record of all in-transit commands.

By following the MQTT topic best practices, the response topic includes overlapping information as the command itself (for example, the original session ID and the response topic used by the smart lock).

As an added capability, the cloud application may have a second IoT rule that uses the session ID to update the status of a specific command using information from the response metadata.

Refer to the following example:

```
{
      "sql": "SELECT sessionId AS token,timestamp()/1000 as ttl, topic() AS
 responseTopic, clientId AS requestorID, res.code AS response.code, 'Complete' AS
 status FROM 'cmd/series100/#' WHERE topic(5) == 'res'”
       "actions": [{
           "dynamoDBv2": {
               "roleArn": "arn:aws:iam::12345678:role/service-role/dynamoDBrole",
           "putItem":{
               "tableName": "command sessions table"
                     }
                 }
           }]
```

Aligning Rules Engine capabilities with MQTT topics

As you define your use of the IoT rules, review the following recommendations as you relate to MQTT topics and the AWS IoT Rules Engine:

- Use the topic(Decimal) rule function to augment your MQTT messages with contextual information contained in your MQTT topics.
- Use the timestamp() rule function to include a timestamp that correlates the time that a message reached AWS IoT Core.
- If your commands are in JSON, reference any contextual payload metadata, such as session ID, in the SELECT and WHERE clause of the rules engine. The additional payload information can be used to determine if and when a rule should initiate.
- Use Substitution templates in your AWS IoT actions to express variables as part of the AWS IoT Rule action that is initiated. Substitution expressions make it easier to scale and dynamically route to downstream IoT Rule actions.
- To track the completion of a command or request, use the AWS IoT Rules Engine to store the data and status in a service, such as DynamoDB. As messages are processed, data can be

automatically expired from DynamoDB using a TTL field. In cases where commands are sent at a high throughput rate, you can leverage the AWS IoT Rules Engine with Amazon Kinesis to buffer data before DynamoDB storage.

- Use the AWS IoT Rules WHERE clause to filter messages that do not apply to an AWS IoT Action. The WHERE clause can be used with the JSON payload or Rules Engine functions, such as get_thing_shadow(thingName, roleARN) or aws_lambda(functionArn, inputJson).
- After AWS IoT Core receives a message, use AWS services like Amazon Kinesis or Amazon SQS to buffer the message payload along with the MQTT topic the message was published to. Once messages are buffered, you can run your own logic on AWS Lambda or Amazon Elastic Compute Cloud (Amazon EC2) to map fields from the payload or the topic and enrich the payload with additional metadata related to the individual devices, the type of device, or the device group. The `topic(decimal)` rule function can be used to enrich the payload with the entire topic when using `topic()`. If you want to enrich the payload with an serial number that is part of the MQTT topic shown below then you would use `topic(4)`.

```
dt/customer435/hub/745384327
```

Conclusion

MQTT is a simple, secure, flexible, and robust IoT protocol. It allows you to define communication networks between devices and the cloud that can tailor fit an increasingly large number of customer use cases. To support you on your initial steps in using MQTT on AWS IoT, this whitepaper has presented several best practices, guidelines, and considerations that can be used when reviewing how to implement IoT device communications.

AWS IoT enables the definition of several MQTT communication patterns - point-to-point, broadcast, and fan-in — that relate to different use cases. In addition, AWS IoT Services provide you with additional managed services including, but not limited to, AWS IoT Jobs, AWS IoT Device Shadow service, and the AWS IoT Rules Engine.

Further reading

For additional information, see the following:

- AWS Whitepapers
- AWS IoT Core Documentation
- Rules for AWS IoT
- The IoT Atlas

Document History and Contributors

Document History

To be notified about updates to this whitepaper, subscribe to the RSS feed.

Change	Description	Date
Whitepaper updated	Updated for Named Shadows, Sample diagrams and examples	December 8, 2021
Minor updates	Adjusted page layout	April 30, 2021
Whitepaper updated	Updated for Basic Ingest.	May 1, 2019
Initial publication	Whitepaper first published.	October 1, 2018

> **Note**
>
> To subscribe to RSS updates, you must have an RSS plug-in enabled for the browser you are using.

Contributors

The following individuals and organizations contributed to this document:

- Olawale Oladehin, Solutions Architect, AWS IoT
- Steve Krems, Solutions Architect, AWS IoT

Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS products or services are provided “as is” without warranties, representations, or conditions of any kind, whether express or implied. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

© 2020 Amazon Web Services, Inc. or its affiliates. All rights reserved.